Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп’ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента	*Цалюка Максима Семеновича*
	(ПІБ)
академічної групи	*122М-22-4*
	(шифр)
спеціальності	*122 Комп’ютерні науки*
	(код і назва спеціальності)
освітньої програми	*«122 Комп’ютерні науки»*
	(назва освітньої програми)
на тему:	*Дослідження продуктивності кросплатформного фреймоворку Flutter в порівнянні з нативними мовами розробки мобільних додатків Swift, Kotlin.*

___________________*М.С. Цалюк*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтин-говою	**інституційною**	
розділів кваліфікаційної роботи				
спеціальний	*доц.Спірінцев В.В.*			
економічний				

Рецензент				

Нормоконтролер	*проф. Лактіонов І.С.*			

Дніпро
2023

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)

_____________ М.О. Алексєєв
(підпис) (прізвище, ініціали)

« » _________ 2023 року

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

студенту *122м-22-4* *Цалюку Максиму Семеновичу*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи *Дослідження продуктивності кросплатформного фреймоворку Flutter в порівнянні з нативними мовами розробки мобільних додатків Swift, Kotlin.*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 09.10.2023 р. № 1227-с

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – процес застосування фреймворку Flutter з метою дослідження продуктивності в порівнянні з нативними фреймворками SwiftUI та Jetpack Compose.

Предмет досліджень – методи та методології побудування мобільних додатків за допомогою фреймворку Flutter на основі єдиної бази коду.

Мета НДР – підвищення ефективності прийняття рішень інженерами та менеджментом IT компаній при плануванні та розробці мобільних додатків на платформах iOS та Android.

Вихідні дані для проведення роботи – теоретичні та експериментальні дослідження, основані на сучасних підходах до розробки мобільних додатків з використанням фреймворку Flutter.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Новизна запропонованих рішень полягає в тому, що отримав подальший розвиток напрямок розробки мобільних додатків, який базується на новій методиці тестування продуктивності додатків в реальному часі, з урахуванням динамічних характеристик та сучасних потреб розробників.

Практична цінність результатів полягає у тому, що запропоновані в роботі моделі і методи дозволяють використовувати знання експерта для вирішення задач розробки мобільного програмного забезпечення.

4 ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Результати досліджень мають бути подані у вигляді, що дозволяє побачити та оцінити безпосереднє використання нечіткої моделі даних. В результаті роботи повинен бути розроблений програмний комплекс для вирішення задачі ідентифікаційної експертизи на основі нечіткої моделі даних.

5 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець)**
Аналіз теми та постановка задачі	09.10.2023-23.10.2023
Аналіз фреймворків Flutter, SwiftUI, Jetpack Compose, для подальшого використання у тестовому середовищі	24.10.2023-10.11.2023
Створення додатків за допомогою фреймворків Flutter, SwiftUI, Jetpack Compose для вирішення задачі порівняння продуктивності усіх фреймворків	10.11.2023-04.12.2023

6 РЕАЛІЗАЦІЯ РЕЗУЛЬТАТІВ ТА ЕФЕКТИВНІСТЬ

Економічний ефект від реалізації результатів роботи очікується позитивним завдяки скорочення витрат на заробітну плату працівникам, які виконують завдання з розробки мобільного програмного забезпечення.

Соціальний ефект від отриманих в ході виконання дипломної роботи результатів очікується позитивним, завдяки удосконаленню та пришвидшенню методів розробки мобільних додатків з використанням фреймворку Flutter.

7 ДОДАТКОВІ ВИМОГИ

Завдання видав ______________ *Спірінцев В.В.*
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання ______________ *Цалюк М.С.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *09.10.2023 р.*

Термін подання кваліфікаційної роботи до ЕК *06.12.2023*

РЕФЕРАТ

Пояснювальна записка: 135 стор., 23 рис., 1 таблиця, 2 додатки, 61 джерело.

Об'єкт дослідження: процес застосування фреймворку Flutter з метою дослідження продуктивності в порівнянні з нативними фреймворками SwiftUI та Jetpack Compose.

Предмет дослідження: методи та методології побудування мобільних додатків за допомогою фреймворку Flutter, на основі єдиної бази коду.

Мета роботи: підвищення ефективності прийняття рішень інженерами та менеджментом IT компаній при плануванні та розробці мобільних додатків на платформах iOS та Android.

Методи дослідження: методи дослідження базуються на основних принципах системного аналізу, крос-платформного програмування, принципах декларативного програмування, об’єктно-орієнтованого програмування, методології використання API у мобільних додатках.

Новизна отриманих результатів: полягає в тому, що отримав подальший розвиток напрямок розробки мобільних додатків, який базується на новій методиці тестування продуктивності додатків в реальному часі, з урахуванням динамічних характеристик та сучасних потреб розробників.

Практична цінність результатів: полягає у тому, що запропоновані в роботі моделі і методи дозволяють використовувати знання експерта для вирішення задач розробки мобільного програмного забезпечення.

Область застосування: Робота про дослідження продуктивності кросплатформного фреймворку Flutter у порівнянні з мовами розробки Swift та Kotlin відкриває можливості оптимізації мобільних додатків через вибір найбільш ефективного інструменту для мультиплатформного розгортання.

Значення роботи та висновки: Робота доводить, що кросплатформний фреймворк Flutter може виявитися важливим інструментом для розробки мобільних додатків, пропонуючи високу продуктивність, ефективність та зручність у порівнянні з нативними мовами розробки Swift та Kotlin.

Прогнози щодо розвитку досліджень: Покращити систему, додавши можливість модифікації профілів користувачів, запис їх до локальної бази даних, оптимізація мобільного додатку. Розширення бази тестових завдань для перевірки продуктивності фреймворку, автоматизація тестів та отримання результатів.

Список ключових слів: Flutter, кроссплатформена розробка, мобільні додатки, Swift, Kotlin, продуктивність, розробка UI, State Management, реактивний інтерфейс, widget.

ABSTRACT

Explanatory note: 135 pages, 23 figures, 1 table, 2 applications, 61 sources.

Object of research: Process of applying the Flutter framework to investigate performance against the native SwiftUI and Jetpack Compose frameworks.

Subject of research: Methods and methodologies for building mobile applications using the Flutter framework, based on a single code base.

Purpose of Master's thesis: Increasing the efficiency of decision-making by engineers and management of IT companies when planning and developing mobile applications on iOS and Android platforms.

Research methods: Research methods are based on the basic principles of system analysis, cross-platform programming, principles of declarative programming, object-oriented programming, methodology of using APIs in mobile applications.

Originality of research is in / consists of / is associated with /: Is that the direction of mobile application development, which is based on a new method of testing the performance of applications in real time, taking into account dynamic characteristics and modern needs of developers, has received further development.

Practical value of the results consists of: Lies in the fact that the models and methods proposed in the work make it possible to use the knowledge of an expert to solve the problems of mobile software development.

Scope of application: The work on the study of the performance of the cross-platform framework Flutter in comparison with the development languages Swift and Kotlin opens the possibility of optimizing mobile applications through the choice of the most effective tool for multi-platform deployment.

The value of the work and conclusions: The work proves that the Flutter cross-platform framework can be an important tool for developing mobile applications, offering high performance, efficiency and convenience compared to native development languages Swift and Kotlin.

Research forecast and development: Improve the system by adding the ability to modify user profiles, save them to the local database, and optimize the mobile application. Expanding the database of test tasks for checking the performance of the framework, automating tests and obtaining results.

Keywords: Flutter, cross-platform development, mobile applications, Swift, Kotlin, productivity, UI development, State Management, reactive interface, widget.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ОС – операційна система;

MVVM – Model View ViewModel;

MVC – Model View Controller;

Bloc – Business Logic Component;

API – Application Interface;

ПЗ – Програмне забезпечення;

SDK – Software Development Kit;

UI – User Interface;

UX – User Expereince;

FPS – Frames Per Second;

ЗМІСТ

РЕФЕ-
РАТ…………………………………………………………………...…**Ошибк а! Закладка не определена.**
ABSTRACT……………………………………………………….……....
Ошибка! Закладка не определена.
ПЕРЕЛІК УМОВНИХ ПОЗНА-
ЧЕНЬ……………………………………….......**Ошибка! Закладка не определена.**
ВСТУП………………………………………………………………...10
РОЗДІЛ 1 АНАЛІЗ ТЕМИ ТА ПОСТАНОВКА ЗА-
ДАЧІ………….........**Ошибка! Закладка не определена.**

1.1. Стан ринку кроссплатформної розробки для мобільних додатків....Ошибка! Закладка не определена.

1.2. Особливості кросплатформної розробкиОшибка! Закладка не определена.

1.2.1. Теоретичні аспекти кроссплатформної розробкиОшибка! Закладка не определена.

1.2.2. Методологічні аспекти кросплатформної розробки ....Ошибка! Закладка не определена.

1.3. Переваги фреймворку Flutter перед іншими засобами кросплатформеної розробки ....Ошибка! Закладка не определена.

1.4. Огляд архітектури кросплатформного фреймворку Flutter....Ошибка! Закладка не определена.

1.4.1. Моделі шарів ....Ошибка! Закладка не определена.

1.4.2. Розбір складових частин додатка розробленого за допомогою Flutter……………………………………………………………..О шибка! Закладка не определена.

1.4.3. Реактивний користувацький інтерфейсОшибка! Закладка не определена.

1.4.4. Widgets ....Ошибка! Закладка не определена.

1.4.5. Побудова віджетів.........................Ошибка! Закладка не определена.

1.4.6. Стан віджету ....Ошибка! Закладка не определена.

1.4.7. Управління станом........................Ошибка! Закладка не определена.

1.4.8. Модель візуалізації Flutter ....Ошибка! Закладка не определена.

1.5. Постановка задачі ....Ошибка! Закладка не определена.

1.6. Висновки до першого розділу ....Ошибка! Закладка не определена.

РОЗДІЛ 2 АНАЛІЗ ТА ДОСЛІДЖЕННЯ РОБОТИ НАТИВНИХ ФРЕЙМ-ВОРКІВ МОБІЛЬНОЇ РОЗБО-РКИ…………………………….................**Ошибка! Закладка не определена.**

2.1. Розбір методів роботи нативного фреймворку розробки iOS додатків SwiftUI ....Ошибка! Закладка не определена.

2.1.1. Огляд основних концепцій SwiftUIОшибка! Закладка не определена.

2.1.2. Мова програмування Swift та його використання в SwiftUI ....Ошибка! Закладка не определена.

2.1.3. Компоненти та структура SwiftUIОшибка! Закладка не определена.

2.1.4. Основні можливості та функціонал SwiftUI для створення інтерфейсів ....Ошибка! Закладка не определена.

2.1.5. Принципи роботи анімацій та графічного візуалу в SwiftUI ....Ошибка! Закладка не определена.

2.1.6. Інструменти та середовище розробки у SwiftUIОшибка! Закладка не определена.

2.1.7. Взаємодія зі SwiftUI через архітектурні патерни та підходи ....Ошибка! Закладка не определена.

2.2. Розбір методів роботи нативного фреймворку розробки Android додатків Jetpack Compose. ....Ошибка! Закладка не определена.

2.2.1. Введення в Jetpack Compose: переваги та концепції...........Ошибка! Закладка не определена.

2.2.2. Основи програмування з Kotlin у Jetpack Compose.............Ошибка! Закладка не определена.

2.2.3. Компоненти та їх функціонал у Jetpack ComposeОшибка! Закладка не определена.

2.2.4. Анімації та інтерактивність у Jetpack ComposeОшибка! Закладка не определена.

2.2.5. Інструменти й середовище розробки для Jetpack Compose.Ошибка! Закладка не определена.

2.2.6. Архітектурні підходи та прийоми роботи з Jetpack Compose ....Ошибка! Закладка не определена.

2.3. Висновки до другого розділу..........Ошибка! Закладка не определена.

РОЗДІЛ 3 ПОРІВНЯННЯ РОБОТИ ФРЕЙМВОРКУ FLUTTER З НАТИВНИМИ ФРЕЙМВОРКАМИ МОБІЛЬНОЇ РОЗРОБКИ……..……**Ошибка! Закладка не определена.**

3.1. Продуктивність та швидкодія Flutter в порівнянні з нативними фреймворками ....Ошибка! Закладка не определена.

3.2. Відмінності у розробці, тестуванні та розгортанні мобільних додатків між Flutter та нативними фреймворкамиОшибка! Закладка не определена.

3.3. Огляд недоліків та можливих обмежень у використанні Flutter в порівнянні з нативними фреймворкамиОшибка! Закладка не определена.

3.4. Розробка додатків на фреймворках Flutter, SwiftUI, Jetpack Compose та практичне порівняння продуктивності.Ошибка! Закладка не определена.

3.4.1. Практичний етап розробки. ....Ошибка! Закладка не определена.

3.4.2. Аналіз продуктивності. ....Ошибка! Закладка не определена.

3.4.3. Оцінка ефективності ресурсів......Ошибка! Закладка не определена.

3.5. Висновки до третього розділу. ....Ошибка! Закладка не определена.

ВИСНО-

ВОК……………………………………………………………**Ошибка! Закладка не определена.**

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕ-

РЕЛ…………………………………...**Ошибка! Закладка не определена.**

Додаток А. ЛІСТИНГ ПРО-

ГРАМИ……………………………………..**Ошибка! Закладка не определена.**

Додаток Б. ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ……….135

ВСТУП

Актуальність теми полягає у тому, що вибір фреймворку для розробки мобільних додатків впливає на якість, швидкість, вартість та конкурентоспроможність продукту. Тому, необхідно мати об'єктивні критерії для порівняння різних фреймворків та визначення їх переваг та недоліків.

Метою роботи є підвищення ефективності прийняття рішень інженерами та менеджментом IT компаній при плануванні та розробці мобільних додатків на платформах iOS та Android. Для досягнення цієї мети було поставлено наступні завдання:

Проаналізувати особливості фреймворку Flutter та його відмінності від нативних фреймворків SwiftUI та Jetpack Compose;

Розробити мобільний додаток за допомогою фреймворку Flutter, який має такі функції: реєстрація та авторизація користувачів, відображення списку товарів, додавання товарів до кошика, оформлення замовлення, відгуки та рейтинги;

Розробити аналогічний мобільний додаток за допомогою нативних фреймворків SwiftUI та Jetpack Compose для платформ iOS та Android відповідно;

Провести тестування розроблених додатків за такими параметрами: час розробки, розмір додатку, швидкість завантаження, використання пам'яті та батареї, зручність інтерфейсу, наявність помилок та збоїв;

Порівняти отримані результати та зробити висновки про продуктивність фреймворку Flutter у порівнянні з нативними фреймворками SwiftUI та Jetpack Compose.

Об'єктом дослідження є процес застосування фреймворку Flutter з метою дослідження продуктивності в порівнянні з нативними фреймвор-

ками SwiftUI та Jetpack Compose. Предметом дослідження є методи та методології побудування мобільних додатків за допомогою фреймворку Flutter, на основі єдиної бази коду.

Методами дослідження є принципи використання засобів крос-платформного програмування, принципи декларативного програмування, об'єктно-орієнтоване програмування, методології використання API у мобільних додатках.

Новизна отриманих результатів полягає у обґрунтованому виборі фреймворку Flutter як інструменту для розробки мобільних додатків для платформ iOS та Android, який може допомогти IT компаніям оптимізувати робочі процеси та витрати.

Практична цінність результатів полягає у тому, що запропоновані в роботі моделі і методи дозволяють використовувати знання експерта для вирішення задач розробки мобільного програмного забезпечення.

Область застосування результатів дослідження є розробка мобільних додатків для різних сфер діяльності, таких як електронна комерція, освіта, розваги, соціальні мережі тощо.

Структура та обсяг кваліфікаційної роботи. Відповідно до мети, задач і предмета дослідження, кваліфікаційна робота складається з реферату, вступу, трьох основних розділів і висновків, списку використаних джерел та 2 додатків. Загальний обсяг роботи містить 135 сторінок друкованого тексту, із них основна частина - 18 сторінки з 19 рис., спеціальна – 89 сторінок, списку використаних джерел з 61 найменувань на 3 сторінках, 2 додатки на 10 сторінках.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1. Стан ринку кросплатформної розробки для мобільних додатків

Ринок кроссплатформеної розробки для мобільних додатків вже був дуже активним і зростав швидко. Однак слід зауважити, що ця область постійно розвивається і змінюється. Ось деякі ключові аспекти ринку кроссплатформеної розробки:

Зростаючий попит на кроссплатформену розробку: За останні кілька років попит на кроссплатформену розробку постійно зростав. Це обумовлено бажанням компаній скоротити витрати та ресурси, необхідні для розробки мобільних додатків для різних платформ (iOS і Android).

Популярність фреймворків: Фреймворки для кроссплатформеної розробки, такі як Flutter (Google), React Native (Facebook), Xamarin (Microsoft), та інші набирали популярність. Flutter, зокрема, став дуже популярним завдяки своєї продуктивності та можливостям для розробки красивих інтерфейсів.

Рост екосистеми: Розвиток кроссплатформених фреймворків призвів до зростання екосистеми і інструментів, доступних для розробників. Це включає в себе сторонні бібліотеки, розширення, плагіни і засоби розробки, які полегшують створення додатків.

Конкуренція з нативною розробкою: Нативна розробка для iOS та Android завжди була основним конкурентом кроссплатформеній розробці. На деяких ринках, таких як фінанси та геймдев, компанії можуть віддавати перевагу нативній розробці для отримання більшої продуктивності і глибокого доступу до функцій пристроїв.

Зростання гібридних додатків: Крім фреймворків, розвивалися і інші підходи, такі як гібридні додатки. Гібридні додатки використовують веб-технології (HTML, CSS, JavaScript) для розробки мобільних додатків, і вони також набули свою аудиторію.

Застосування в корпоративному секторі: Корпорації та підприємства також звертали свою увагу на кроссплатформену розробку для створення корпоративних додатків, оскільки це може значно скоротити витрати на розробку і підтримку.

Питання продуктивності та нативності: Однак існували обмеження щодо продуктивності та нативності додатків, створених з використанням кроссплатформених фреймворків. В деяких випадках, особливо для важких ігор або додатків з високими вимогами до продуктивності, нативна розробка залишалася більш вигідним варіантом.

Зростання спільноти та ресурсів: Велика активна спільнота розробників та наявність безлічі навчальних матеріалів і курсів сприяла зростанню інтересу до кроссплатформеної розробки.

Загалом, станом на зараз, кроссплатформена розробка є живим і динамічним сегментом ринку мобільних додатків, і вона має потенціал для подальшого зростання і розвитку.

1.2. Особливості кросплатформної розробки

У сучасному цифровому світі розробка програмного забезпечення є надзвичайно важливою галуззю індустрії. З поширенням різних платформ, операційних систем, пристроїв та архітектур програмного забезпечення виникає потреба в створенні додатків та програм, які могли б працювати на різних пристроях та платформах без великих зусиль та витрат часу на написання окремого коду для кожної з них. Саме тут і стає актуальною кроссплатформена розробка.

Ми розглянемо концепцію кроссплатформеної розробки в усій її глибині та ширині. Ми охопимо теоретичні та методологічні аспекти цієї галузі, вивчимо аналітичний огляд літературних джерел з предмета дослідження, а також критично проаналізуємо різні погляди та класифікуємо їх. Ми також розглянемо основні фактори, які впливають на стан та розвиток кроссплатформеної розробки, та детально розглянемо її сутність, значення, класифікаційні характеристики, історію та тенденції розвитку.

1.2.1. Теоретичні аспекти кросплатформної розробки

Принцип "Write Once, Run Anywhere" (Пиши раз, запускайте будь-де). Однією з основних ідей кроссплатформеної розробки є можливість написання коду один раз та його використання на різних платформах. Це досягається завдяки використанню спеціальних фреймворків та інструментів, які надають абстракцію над операційною системою та апаратними ресурсами.

Основними засобами для реалізації кроссплатформеної розробки є фреймворки. Це набори інструментів, бібліотек та середовищ, які надають можливість розробляти додатки для різних платформ з одним кодом. Приклади таких фреймворків включають Flutter, React Native, Xamarin, Apache Cordova, Unity, і багато інших.

Для кроссплатформеної розробки використовуються різні мови програмування. Наприклад, Dart використовується в Flutter, JavaScript - в React Native, C# - в Xamarin. Ці мови підтримуються фреймворками і використовуються для створення логіки додатків.

1.2.2. Методологічні аспекти кросплатформної розробки

Для кроссплатформеної розробки важливо визначити архітектуру додатка, яка дозволить використовувати спільний код для різних платформ, а також реалізувати адаптивний та модульний підхід до розробки. Часто використовуються шаблони проектування, такі як MVVM (Model-View-ViewModel) або MVC (Model-View-Controller), для створення структури додатку.

Створення інтерфейсу користувача (UI) та досвіду користувача (UX) для різних платформ важливе завдання у кроссплатформеній розробці. Фреймворки надають можливість створювати адаптивні інтерфейси, які виглядають та працюють на кожній платформі належним чином.

Кроссплатформені додатки потребують тестування та налагодження на різних платформах. Це включає в себе юніт-тести, інтеграційні тести та тестування в реальних умовах різних пристроїв. Для автоматизації тестування використовуються різні інструменти, такі як Appium, Detox, Espresso, і інші.

Після випуску додатка важливо забезпечити його підтримку та регулярні оновлення для всіх платформ. Це включає в себе виправлення помилок, покращення функціональності та адаптацію до нових версій операційних систем.

Узагальнюючи, кроссплатформена розробка - це методологія, яка поєднує теоретичні аспекти, пов'язані з підходами до створення спільного коду для різних платформ, та практичні методи реалізації, такі як використання фреймворків, мов програмування та налагодження на різних платформах. Вона дозволяє розробникам зробити більше з меншею кількістю зусиль та ресурсів, що робить її популярною в сучасному програмуванні.

1.3. Переваги фреймворку Flutter перед іншими засобами кросплатформної розробки

Flutter - це високопродуктивний кросcплатформений фреймворк розробки мобільних та веб-додатків, розроблений компанією Google. Він набув великої популярності серед розробників завдяки своїм унікальним перевагам порівняно з іншими мовами та фреймворками для кросплатформеної розробки. Ось деякі з основних переваг Flutter:

Одноразовий код для обох платформ: Основною перевагою Flutter є можливість використовувати один і той же код для розробки додатків для обох основних мобільних платформ - iOS та Android. Це дозволяє заощадити час і ресурси, так як не потрібно писати окремий код для кожної платформи.

Висока продуктивність: Flutter використовує двигун Skia для відтворення інтерфейсів, що дозволяє досягнути високої продуктивності та швидкості відгуку додатків. Він також використовує компіляцію в машинний код, що забезпечує оптимальну продуктивність додатків.

Красивий інтерфейс і нативний вигляд: За допомогою Flutter легко створювати красиві та динамічні інтерфейси з високим рівнем настроювання. Він дозволяє досягнути нативного вигляду та поведінки додатків на обох платформах.

Багатофункціональність: Flutter поставляється з багатьма вбудованими компонентами та бібліотеками, що спрощують розробку додатків. Крім того, існують сторонні плагіни та розширення, які розширюють функціональність фреймворку.

Гарна підтримка спільноти та документація: Flutter має активну спільноту розробників, що сприяє вирішенню проблем і обміну знаннями. Офіційна документація також дуже добре написана і оновлюється регулярно.

Швидкий розріст і підтримка Google: Flutter отримує підтримку від Google, що означає, що він активно розвивається та має потенціал для подальшого зростання і вдосконалення. Велика кількість внутрішніх та сторонніх проектів вже використовують Flutter.

Підтримка десктопу та веб-розробки: Починаючи з Flutter 2.0, фреймворк підтримує розробку додатків для десктопів (Windows, macOS, Linux) та веб-додатків. Це розширює область застосування Flutter і дозволяє використовувати його для розробки на різних платформах.

Усі ці переваги роблять Flutter привабливим вибором для розробки кроссплатформених мобільних додатків і дозволяють розробникам створювати якісні додатки швидко та ефективно.

1.4. Огляд архітектури кросплатформного фреймворку Flutter

Програми розроблені за допомогою фреймворку Flutter, запускаються у віртуальній машині, яка пропонує гаряче перезавантаження змін без потреби повної перекомпіляції. Релізна версія додатка написаного на Flutter компілюється безпосередньо в машинний код, інструкції Intel x64 чи ARM, або в JavaScript, якщо націлено на веб розробку. Фреймворк має відкритий вихідний код із дозвільною ліцензією BSD і має екосистему що процвітає, багатий вибір пакетів сторонніх розробників, які доповнюють основні функції бібліотеки.

Огляд архітектури поділено на декілька розділів: Модель шару: фрагменти, з яких побудовано Flutter. Реактивні інтерфейси користувача: основна концепція розробки інтерфейсу користувача Flutter. Вступ до віджетів: основні будівельні блоки інтерфейсу користувача Flutter. Процес візуалізації: як Flutter перетворює код інтерфейсу користувача на пікселі.

Огляд платформи для вбудовування: код, який дозволяє мобільним і настільним ОС запускати програми Flutter.

Інтеграція Flutter з іншим кодом: інформація про різні методи, доступні для програм Flutter.

Підтримка Веб розробки: остаточні зауваження щодо характеристик Flutter у середовищі браузера.

1.4.1. Моделі шарів

Flutter розроблено як розширювана багатошарова система. Він існує як ряд незалежних бібліотек, кожна з яких залежить від базового рівня. Жоден рівень не має привілейованого доступу до нижнього рівня, і кожна частина рівня структури створена як необов'язкова та замінна. Приклад моделей шарів системи Flutter наведено на рис. 1.1.

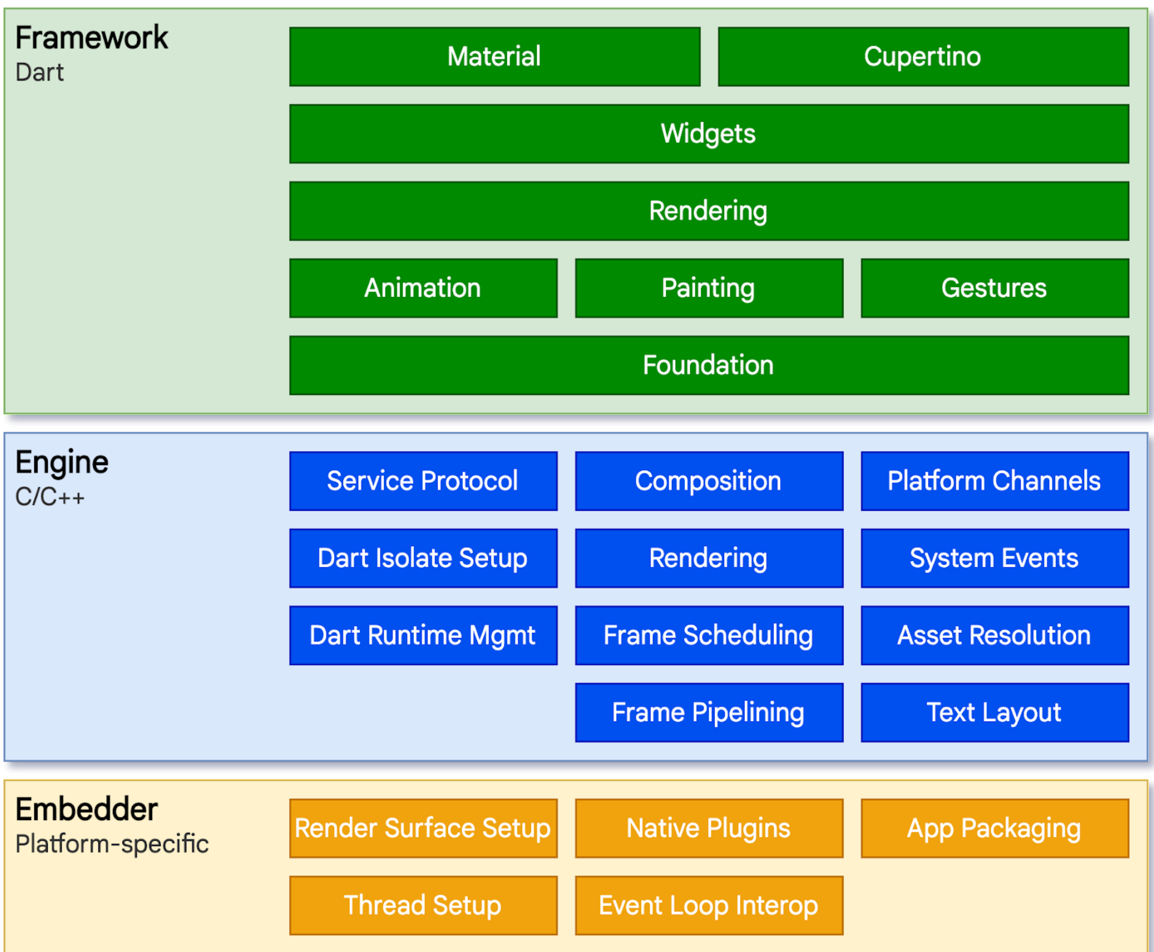


Рис. 1.1. Моделі шарів системи Flutter

Для операційної системи програми Flutter упаковані, так само як і будь-яка інша нативна програма. Спеціальна платформа для вбудовування забезпечує точку входу, координує роботу з базовою операційною системою для доступу до таких служб, як поверхні відтворення, доступність і введення; і керує циклом подій повідомлення. Вбудовувач написано мовою, яка підходить для платформи: наразі Java і C++ для Android, Objective-C/Objective-C++ для iOS і macOS і C++ для Windows і Linux. Використовуючи embedder, код Flutter можна інтегрувати в існуючу програму як модуль або код може являти собою весь вміст програми. Flutter включає кілька вбудованих пристроїв для поширених цільових платформ, але існують і інші вбудовані пристрої.

В основі фреймворку Flutter лежить двигун Flutter, який здебільшого написаний мовою C++ і підтримує примітиви, необхідні для підтримки всіх додатків які написані на Flutter. Механізм відповідає за растеризацію складених сцен щоразу, коли потрібно намалювати новий кадр. Він забезпечує низькорівневу реалізацію основного API Flutter, включаючи графіку (через Impeller на iOS і coming на Android, а також Skia на інших платформах), макет тексту, файловий і мережевий ввід-вивід, підтримку доступності, архітектуру плагінів і середовище виконання Dart і інструментарій для компіляції.

Двигун доступний для фреймворку Flutter через dart:ui, який загортає базовий код C++ у класи Dart. Ця бібліотека розкриває примітиви найнижчого рівня, такі як класи для управління введенням, графікою та підсистемами відтворення тексту.

Як правило, розробники взаємодіють із Flutter через фреймворк Flutter, який надає сучасну реактивну структуру, написану мовою Dart. Він містить багатий набір платформи, макета та базових бібліотек, які складаються з серії шарів. Працюючи знизу вгору, ми маємо:

Основні базові класи та служби будівельних блоків, такі як анімація, малювання та жести, які пропонують широко використовувані абстракції над основою.

Рівень візуалізації забезпечує абстракцію для роботи з макетом. За допомогою цього шару ви можете побудувати дерево об'єктів, які можна відобразити. Ви можете керувати цими об'єктами динамічно, при цьому дерево автоматично оновлює макет, щоб відобразити ваші зміни.

Шар віджетів — це абстракція композиції. Кожен об'єкт візуалізації в шарі рендерингу має відповідний клас у шарі віджетів. Крім того, рівень віджетів дозволяє визначати комбінації класів, які можна повторно використовувати. Це рівень, на якому вводиться модель реактивного програмування.

Бібліотеки Material і Cupertino пропонують комплексні набори елементів керування, які використовують примітиви композиції шару віджетів для реалізації мов дизайну Material(Google) або Cuppertino(iOS).

Фреймворк Flutter відносно невеликий; багато функцій вищого рівня, які можуть використовувати розробники, реалізовано у вигляді пакетів, включаючи плагіни платформи, такі як камера та веб-перегляд, а також функції, не залежні від платформи, такі як символи, http та анімація, які базуються на основних бібліотеках Dart і Flutter. Деякі з цих пакетів походять із ширшої екосистеми, охоплюючи такі послуги, як платежі в програмі, автентифікація Apple та анімація.

1.4.2. Розбір складових частин додатка розробленого за допомогою Flutter

Dart App

Компонує віджети в потрібний інтерфейс користувача. Реалізує бізнес-логіку. Належить розробнику програми.

Framework

Надає API вищого рівня для створення високоякісних додатків (наприклад, віджети, тестування попадань, виявлення жестів, доступність, введення тексту). Об’єднує дерево віджетів програми в сцену.

Engine

Відповідає за растеризацію складених сцен. Забезпечує низькорівневу реалізацію основних API Flutter (наприклад, графіка, макет тексту, середовище виконання Dart).

Надає свої функції фреймворку за допомогою API dart:ui.

Інтегрується з певною платформою за допомогою API Embedder Engine.

Embedder

Координується з базовою операційною системою для доступу до таких служб, як поверхні відтворення, доступність і введення. Керує циклом подій. Розкриває специфічний для платформи API для інтеграції Embedder у програми.

Runner

Компонує фрагменти, надані специфічним для платформи API Embedder, у пакет програми, який можна запускати на цільовій платформі.

Частина шаблону програми, згенерованого flutter create, належить розробнику програми.

1.4.3. Реактивний користувацький інтерфейс

Реактивний користувацький інтерфейс (RUI) - це дизайн та розробка користувацького інтерфейсу (UI), який реагує на дії користувача та зміни в даних в реальному часі. RUI надає користувачам більш інтерактивний та динамічний досвід при взаємодії з програмними додатками та веб-сайтами.

Основна ідея полягає в тому, щоб інтерфейс був відзивчивим на події та зміни, щоб відображати актуальну інформацію та реагувати на користувацькі взаємодії.

Основні характеристики реактивного користувацького інтерфейсу включають:

Автоматичне оновлення: RUI автоматично оновлює вміст інтерфейсу при зміні даних або стану програми. Це означає, що користувачі бачать актуальну інформацію без необхідності оновлення сторінки або додатку вручну.

Реакція на події: RUI може реагувати на різні події, такі як кліки, введення тексту, переміщення миші, жести, натискання клавіш і багато інших. Він може виконувати певні дії або змінювати відображений контент в залежності від подій.

Відсутність блокування інтерфейсу: RUI дозволяє виконувати фонові операції асинхронно, не блокуючи основний інтерфейс. Це дозволяє користувачам продовжувати взаємодіяти з програмою, навіть якщо її виконання займає певний час.

Динамічність і анімація: RUI сприяє використанню анімації та плавних переходів для покращення досвіду користувача. Динамічний інтерфейс може бути більш привабливим та цікавим для користувачів.

Підтримка різних пристроїв і розмірів екранів: RUI може адаптуватися до різних типів пристроїв та розмірів екранів, щоб забезпечити оптимальний досвід користувача на різних пристроях, включаючи смартфони, планшети, настільні комп'ютери та інші.

Реактивний користувацький інтерфейс активно використовується в різних сферах, включаючи розробку мобільних додатків, веб-розробку, графічний дизайн та багато інших областей. Однією з популярних платформ для створення реактивного інтерфейсу є фреймворк Flutter, розроблений Google.

1.4.4. Widgets

У Flutter термін "widget" використовується для позначення будь-якого елемента інтерфейсу, будь то кнопка, текстове поле, зображення, або навіть ціла сторінка додатку. Widget - це основна будівельна одиниця для створення інтерфейсу користувача.

Віджети можуть бути простими (наприклад, текстове поле) або складними (наприклад, екран зі списком пунктів меню). Це дозволяє вам будувати інтерфейс на будь-якому рівні складності.

У Flutter всі віджети є об'єктами, і вони всі спадають від базового класу Widget. Це робить їх консистентними у використанні та спільними в роботі.

Віджети у Flutter є нерухомими та незмінними. Якщо вам потрібно змінити віджет, ви просто створюєте новий екземпляр віджета з новими параметрами.

Ви можете створювати складні інтерфейси, комбінуючи багато віджетів разом у відповідному порядку. Віджети можуть бути вкладені один в одного, що дозволяє створювати дерево віджетів.

У Flutter є велика кількість вбудованих віджетів для відображення тексту, зображень, іконок, кнопок, форм і багато інших компонентів.

Flutter використовує власний двигун для малювання інтерфейсу, тому він незалежний від платформи. Ви можете використовувати одні й ті ж віджети для створення додатків для iOS і Android.

Flutter надає потужні засоби для створення анімації та обробки користувацьких взаємодій.

Важливо зазначити, що віджети в Flutter можуть бути статичними (без змінюваної структури) або динамічними (зі змінюваною структурою). Також, Flutter має велику кількість вбудованих віджетів, і ви можете створювати свої власні віджети для вирішення конкретних завдань.

1.4.5. Побудова віджетів

Для визначення візуального представлення віджета, перевизначається функція build() для повернення нового дерева елементів. Це дерево представляє частину інтерфейсу користувача віджета в більш конкретних термінах. Наприклад, віджет панелі інструментів може мати функцію побудови, яка повертає горизонтальний макет деякого тексту та різних кнопок. За потреби фреймворк рекурсивно запитує кожен віджет про створення, доки дерево не буде повністю описано конкретними об'єктами які можуть бути відмальовані. Потім фреймворк зшиває візуалізовані об'єкти в дерево візуалізованих об'єктів.

Функція створення віджета не повинна мати побічних ефектів. Кожного разу, коли функцію просять створити, віджет повинен повертати нове дерево віджетів, незалежно від того, що віджет повертав раніше. Фреймворк виконує важку роботу, щоб визначити, які методи збирання потрібно викликати на основі дерева об'єктів візуалізації.

У кожному відтвореному кадрі Flutter може відтворити лише ті частини інтерфейсу, де стан змінився, викликавши метод build() цього віджета. Тому важливо, щоб методи побудови поверталися швидко, а важка обчислювальна робота повинна виконуватися деяким асинхронним способом, а потім зберігатися як частина стану, який буде використовуватися методом побудови.

Попри відносно наївний підхід, це автоматизоване порівняння є досить ефективним, оскільки дозволяє створювати високопродуктивні інтерактивні програми. І дизайн функції збірки спрощує ваш код, зосереджуючись на оголошенні того, з чого складається віджет, а не на складнощах оновлення інтерфейсу користувача з одного стану в інший.

1.4.6. Стан віджету

Фреймворк представляє два основні класи віджетів: StatefulWidget та StatelessWidget.

Багато віджетів не мають змінного стану: вони не мають жодних властивостей, які змінюються з часом (наприклад, значка чи мітки). Ці віджети підкласу StatelessWidget.

Однак, якщо унікальні характеристики віджета потрібно змінити на основі взаємодії користувача або інших факторів, цей віджет має стан. Наприклад, якщо віджет має лічильник, який збільшується щоразу, коли користувач натискає кнопку, тоді значення лічильника є станом цього віджета. Коли це значення змінюється, віджет потрібно перебудувати, щоб оновити його частину інтерфейсу користувача. Ці віджети підкласу StatefulWidget, і (оскільки сам віджет є незмінним) вони зберігають змінний стан в окремому класі, який підкласи State. StatefulWidgets не мають методу побудови; натомість їхній інтерфейс користувача будується через їхній об’єкт State.

Кожного разу, коли ви змінюєте об’єкт State (наприклад, шляхом збільшення лічильника), ви повинні викликати setState(), щоб сигналізувати фреймворку про оновлення інтерфейсу користувача шляхом повторного виклику методу побудови State.

Наявність окремих об’єктів стану та віджетів дозволяє іншим віджетам обробляти як віджети без стану, так і віджети без стану точно так само, не турбуючись про втрату стану. Замість того, щоб утримувати дочірній елемент, щоб зберегти його стан, батько може створити новий екземпляр дочірнього елемента в будь-який час, не втрачаючи його постійний стан. Фреймворк виконує всю роботу з пошуку та повторного використання чинних об’єктів стану, коли це необхідно.

1.4.7. Управління станом

Управління станом (State Management) - це концепція, яка допомагає керувати та організовувати дані та стан вашого додатка. Оскільки Flutter дозволяє швидко оновлювати та відображати користувацький інтерфейс (UI), правильне управління станом є важливим аспектом розробки додатків.

Інтерфейс SetState: У Stateful віджетах ви можете використовувати метод setState(), щоб оновлювати стан віджета та викликати перебудову інтерфейсу. Це дозволяє змінювати стан і перерисовувати віджет при необхідності.

InheritedWidget та Provider: Для більш складного управління станом і передачі даних між віджетами в Flutter використовуються інші підходи, такі як InheritedWidget та пакети сторонніх розробників, наприклад, Provider. Вони дозволяють створювати та розповсюджувати дані в додатку таким чином, щоб вони були доступні у всіх частинах дерева віджетів без необхідності передачі їх напряму через конструктори.

Інші бібліотеки управління станом: Для ще більш ефективного та структурованого управління станом можна використовувати бібліотеки, такі як Redux, MobX, Bloc та ще багато інших які допомагають управляти глобальним станом і діями в додатку. Найпоширенішою на сьогодні є Bloc.

Загалом, управління станом у Flutter дозволяє вам ефективно та організовано керувати даними та оновленнями UI у вашому додатку, щоб забезпечити плавну та ефективну роботу програми.

1.4.8. Модель візуалізації Flutter

Якщо Flutter – це кросплатформенний фреймворк, то як він може запропонувати порівнянну продуктивність з одноплатформенними фреймворками, такими як наприклад SwiftUI чи Jetpack Compose?

Треба почати з того, як працюють традиційні програми Android. Під час малювання ви спочатку викликаєте код Java фреймворку Android. Системні бібліотеки Android забезпечують компоненти, відповідальні за малювання в об'єкті Canvas, який Android може потім відобразити за допомогою Skia, графічного механізму, написаного на C/C++, який викликає центральний чи графічний процесор для завершення малювання на пристрої.

Кросплатформ фреймворки зазвичай працюють, створюючи рівень абстракції над базовими рідними бібліотеками інтерфейсу користувача Android та iOS, намагаючись згладити невідповідності представлення кожної платформи. Код додатка часто пишеться інтерпретованою мовою, як-от JavaScript, яка, у свою чергу, має взаємодіяти з системними бібліотеками Android на основі Java або iOS на основі Objective-C для відображення інтерфейсу користувача. Усе це додає накладних витрат, які можуть бути значними, особливо там, де існує велика взаємодія між інтерфейсом користувача та логікою програми.

Навпаки, Flutter мінімізує ці абстракції, обходячи системні бібліотеки віджетів інтерфейсу користувача на користь власного набору віджетів. Код Dart, який малює візуальні ефекти Flutter, скомпільований у нативний код, який використовує Skia (або, у майбутньому, Impeller) для візуалізації. Flutter також вбудовує власну копію Skia як частину двигуна, що дозволяє розробнику оновлювати свою програму, щоб залишатися в курсі останніх покращень продуктивності, навіть якщо телефон не оновлено новою версією Android. Те саме стосується Flutter на інших нативних платформах, таких як Windows або macOS.

Головний принцип, який Flutter застосовує до конвеєра візуалізації, полягає в тому, що просто – це швидко. Flutter має зрозумілий конвеєр для того, як дані надходять до системи, як показано на рис. 1.2.:

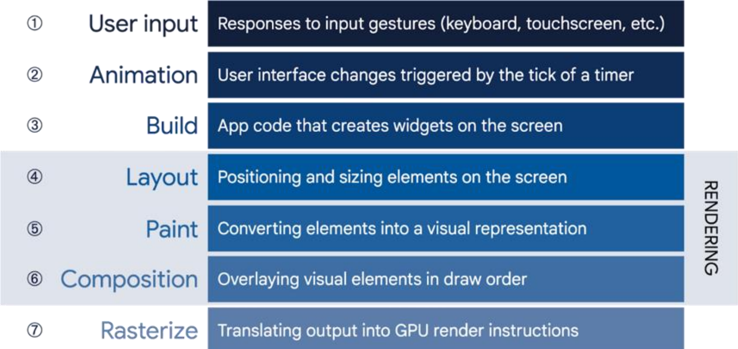


Рис. 1.2. Конвеєр візуалізації Flutter

1.5. Постановка задачі

Отже, в результаті розгляду ринку мобільної розробки станом на зараз та проблем з якими стикаються бізнеси та програмісти під час розробки мобільного ПО, стає зрозумілим, що кросплатформена розробка є актуальним та дуже зручним рішенням. Але як кросплатформена розробка, в нашому випадку фреймворк Flutter може забезпечити такий же рівень продуктивності як додатки написані на нативних мовах розробки, та такий саме рівень User Experience.

У даній кваліфікаційній роботі необхідно розв'язати наступні задачі:

Поглиблено розглянути процес роботи двигуна Skia, принцип візуалізації та управління станом фреймворку Flutter.

Вибір архітектури розробки ПО, котра може бути використано у Flutter, SwiftUI, JetpackCompose.

Створити комплексне програмне забезпечення за допомогою трьох найсучасніших фреймворків: Flutter, SwiftUI, JetpackCompose.

Провести оцінювання якості всіх трьох додатків.

Обробити отримані дані, візуалізувати їх та зробити висновки.

1.6. Висновки до першого розділу

У першому розділі роботи було розглянуто основні тенденції та перспективи ринку розробки мобільних додатків, а також особливості фреймворку Flutter, який є одним з найпопулярніших засобів крос-платформного програмування. На основі аналізу даних було зроблено наступні висновки:

Ринок розробки мобільних додатків є динамічним, конкурентним та інноваційним, що вимагає від розробників постійного вдосконалення своїх навичок та знань.

Фреймворк Flutter є високопродуктивним, гнучким та зручним інструментом для розробки мобільних додатків для платформ iOS та Android на основі єдиної бази коду. Flutter використовує мову програмування Dart та принципи декларативного програмування для створення нативних інтерфейсів користувача.

Flutter має ряд переваг у порівнянні з нативними фреймворками SwiftUI та Jetpack Compose, таких як швидкість розробки, гаряче перезавантаження та перезапуск, висока продуктивність, крос-платформність, кастомізація дизайну та анімації, підтримка веб- та десктоп-розробки . Однак, Flutter також має свої недоліки, такі як великий розмір додатків, обмежена підтримка платформи iOS, нестабільність деяких функцій та залежність від Google .

Таким чином, фреймворк Flutter є перспективним інструментом для розробки мобільних додатків, який може задовольнити потреби різних сегментів ринку та користувачів. Однак, для повноцінного використання його можливостей необхідно враховувати його особливості, переваги та недоліки. У наступних розділах роботи буде проведено практичне дослідження продуктивності фреймворку Flutter у порівнянні з нативними фреймворками SwiftUI та Jetpack Compose.

РОЗДІЛ 2
АНАЛІЗ ТА ДОСЛІДЖЕННЯ РОБОТИ НАТИВНИХ ФРЕЙМВОРКІВ МОБІЛЬНОЇ РОЗРОБКИ

2.1. Розбір методів роботи нативного фреймворку розробки iOS додатків SwiftUI

2.1.1. Огляд основних концепцій SwiftUI

SwiftUI - це декларативний фреймворк розробки користувацького інтерфейсу для платформи iOS, iPadOS, macOS, watchOS і tvOS від Apple. Його основне призначення - спростити процес створення інтерфейсу за допомогою Swift, мови програмування, яка є стандартною для розробки додатків для платформ Apple.

SwiftUI дозволяє розробникам створювати інтерфейси для мобільних додатків та програм для інших платформ за допомогою декларативного підходу. Основна ідея полягає у визначенні того, як повинен виглядати інтерфейс, а не в тому, яким чином досягти цього за допомогою коду.

Декларативний підхід визначає структуру інтерфейсу через опис того, як він повинен виглядати при певних умовах та змінах стану. Замість прямого визначення кожного кроку для створення користувацького інтерфейсу, розробник описує бажаний результат, і SwiftUI бере на себе відповідність за його відображення та збереження стану.

Цей підхід дозволяє прискорити розробку, зменшити кількість коду, а також спростити та зробити більш зрозумілим процес створення інтерфейсу для розробників, особливо для тих, хто раніше не мав досвіду в роботі з іншими фреймворками. SwiftUI також надає багато інструментів для роботи з анімацією, переходами між екранами, адаптацією до різних розмірів пристроїв та іншими аспектами розробки.

Цей фреймворк поступово стає основним інструментом для розробників, які прагнуть створити сучасні, ефективні та динамічні додатки для екосистеми продуктів Apple.

SwiftUI пропонує декларативний підхід до створення користувацького інтерфейсу. Це означає, що програміст описує, як повинен виглядати інтерфейс у певний момент часу або в певному стані, а не послідовність кроків для створення цього інтерфейсу.

Основна концепція декларативного підходу в SwiftUI полягає у визначенні того, що програма повинна зробити, а не того, яким чином це повинно бути зроблено. Ви описуєте, які елементи, компоненти та структура інтерфейсу мають бути представлені, а фреймворк бере на себе відповідність за їх відображення та управління їх станом.

Наприклад, якщо вам потрібно створити кнопку у SwiftUI, ви можете описати її вигляд та поведінку за допомогою коду подібного до наступного:

````
```
Button("Натисніть мене") {
    print("Кнопка була натиснута!")
}
.padding()
.background(Color.blue)
.foregroundColor(.white)
.cornerRadius(10)
```
````

У цьому прикладі описано текст кнопки ("Натисніть мене"), а також дію, яка відбудеться при натисканні (виведення "Кнопка була натиснута!" у консоль). Описано також стиль кнопки, такі як колір фону, колір тексту, радіус закруглення тощо.

Коли стан програми змінюється, наприклад, якщо користувач натискає кнопку, SwiftUI автоматично оновлює інтерфейс, змінюючи відповідні елементи за вашими описами. Ви не повинні вказувати, як саме

ці зміни повинні відбутися; це відбувається автоматично завдяки декларативному підходу.

SwiftUI базується на концепції віджетів, які представляють компоненти користувацького інтерфейсу. Віджети є фундаментальними будівельними блоками для побудови візуальних елементів програми. Вони є частиною декларативного підходу SwiftUI, де кожен віджет відповідає за певний елемент або частину інтерфейсу та його відображення.

Кожен віджет в SwiftUI є самодостатньою частиною інтерфейсу, яку можна використовувати самостійно або як складову частину більш складних візуальних елементів. Вони можуть бути текстовими полями, кнопками, картинками, списками, кольорами, формами і багатьма іншими елементами.

Віджети в SwiftUI відіграють ключову роль у створенні декларативного користувацького інтерфейсу. Кожен віджет описує або відображає певну частину інтерфейсу і має певні властивості, такі як розмір, колір, форма, стиль тощо. Ці властивості можуть бути динамічно змінені в залежності від стану програми.

У SwiftUI є широкий спектр вбудованих віджетів, що дозволяє створювати різні типи візуальних елементів. Наприклад, `Text` для текстових об'єктів, `Button` для створення кнопок, `List` для списків, `Image` для зображень, `TextField` для текстових полів і багато інших. Крім того, можна створювати власні віджети, комбінуючи та наслідуючи вбудовані віджети для створення нових користувацьких елементів.

Різноманітність віджетів в SwiftUI дозволяє створювати складні та зручні інтерфейси з використанням готових компонентів, що спрощує процес створення та підтримки інтерфейсу мобільних додатків.

SwiftUI надає систему моделей даних, яка дозволяє легко і ефективно організовувати дані та їх відображення в користувацькому інтерфейсі. Ця система ґрунтується на декларативному підході, де зміни в моделі

автоматично відображаються в інтерфейсі, що спрощує роботу з даними та їх оновлення.

Модель даних у SwiftUI може бути створена як Swift-структура або клас, що відображає конкретні дані або об'єкт додатку. Ці об'єкти можуть містити властивості, методи та функції, які визначають їхню поведінку та характеристики.

В SwiftUI використовуються `@State` та `@Binding` для відстеження стану даних та автоматичного оновлення інтерфейсу при зміні даних. `@State` дозволяє оголосити змінну, зміни якої автоматично призводять до оновлення відповідної частини інтерфейсу. `@Binding` використовується для передачі зв'язку між об'єктами, що дозволяє змінювати значення даних в одному місці та автоматично оновлювати його в іншому.

SwiftUI дозволяє зв'язувати дані з зовнішніми джерелами, такими як бази даних, API або інші сервіси. Це робиться шляхом використання властивостей `ObservableObject` та `@Published`, що дозволяють автоматично відслідковувати та оновлювати дані, отримані з цих джерел.

Під час роботи з моделями даних у SwiftUI, відбувається прив'язка об'єктів до віджетів, що дозволяє автоматично відображати зміни даних в інтерфейсі. Зміни у моделі, визначені як `@State`, спричиняють перерендеринг відповідних віджетів з автоматичним відображенням оновлених даних.

Ця система моделей даних та їх зв'язок з інтерфейсом у SwiftUI дозволяє розробникам ефективно створювати та підтримувати декларативний інтерфейс з автоматичним оновленням при зміні даних.

SwiftUI пропонує різноманітні елементи для створення користувацького інтерфейсу, що включають в себе текстові поля, кнопки, списки, графіки та багато іншого. Кожен з цих елементів має свої унікальні властивості та можливості, які спрощують створення динамічного та привабливого інтерфейсу.

SwiftUI має потужні засоби для відображення тексту. Елемент `Text` використовується для відображення простого тексту, а засоби форматування тексту, такі як розмір, шрифт, колір тощо, задаються через модифікатори. `TextField` дозволяє користувачеві вводити текст та редагувати його.

SwiftUI надає `Button` для створення кнопок. Це дозволяє створювати кнопки з текстом або без, які виконують певні дії при натисканні.

`List` використовується для створення списків, де можна відображати динамічні дані. Це може бути масив об'єктів, дані з бази даних або зовнішніх джерел.

SwiftUI надає можливості для малювання графіків, використовуючи `Shape` та `Path`. `Shape` визначає форму, а `Path` дозволяє створювати складні форми шляхом визначення точок та ліній між ними.

Також, SwiftUI надає можливості для створення інших елементів, таких як слайдери, випадаючі списки, переглядачі, вкладки та інші, які дозволяють розширити можливості створення різноманітних інтерфейсів.

Використання цих елементів дозволяє розробникам створювати різноманітні та динамічні інтерфейси для мобільних додатків з декларативним підходом до створення UI. Приклад віджетів які використовуються в SwiftUI наведено на рис. 2.1.

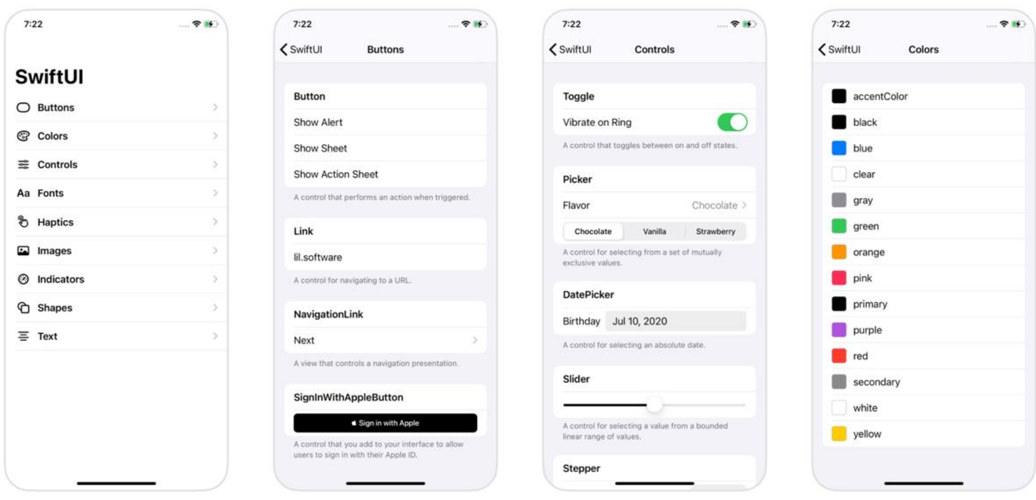


Рис. 2.1. Віджети які використовуються в SwiftUI

SwiftUI надає гнучкість та простоту у створенні складних користувацьких інтерфейсів для мобільних додатків. Його спрощений підхід до розробки дозволяє розробникам швидко та ефективно створювати різноманітні інтерфейси з високою гнучкістю та масштабованістю.

SwiftUI базується на декларативному підході, що означає, що розробники описують, як повинен виглядати їхній інтерфейс, а не те, як його створити крок за кроком. Це дозволяє описувати вигляд та поведінку елементів інтерфейсу, а система SwiftUI самостійно керує реалізацією цих описів.

Один з ключових аспектів SwiftUI - це можливість створення власних компонентів та їх використання для побудови інтерфейсів. SwiftUI використовує модульний підхід, де окремі елементи можуть бути використані для побудови більших компонентів, що спрощує роботу зі складними інтерфейсами.

Однією з важливих особливостей SwiftUI є його здатність автоматично оновлювати інтерфейс, коли дані змінюються. Він реагує на будь-які зміни в стані даних та оновлює відповідні елементи інтерфейсу без необхідності вручному оновленні.

Ще однією перевагою є те, що SwiftUI підтримує різні платформи, такі як iOS, macOS, watchOS та tvOS. Це дозволяє розробникам використовувати один і той же код для створення інтерфейсу для різних пристроїв та платформ.

SwiftUI створений з урахуванням потреб розробників, щоб забезпечити простоту використання, гнучкість та швидкість створення складних та естетичних інтерфейсів для мобільних додатків.

SwiftUI, безумовно, має свої переваги та недоліки в порівнянні з іншими фреймворками для розробки мобільних додатків.

SwiftUI використовує декларативний підхід, де розробник описує, як повинен виглядати інтерфейс, що робить код більш зрозумілим та зменшує його обсяг.

Завдяки простому синтаксису та автоматизованому оновленню інтерфейсу під час зміни даних, SwiftUI пропонує значно швидший процес розробки.

SwiftUI повністю інтегрований з мовою програмування Swift, що дозволяє розробникам використовувати всі можливості мови для створення інтерфейсів.

SwiftUI підтримує різні платформи Apple, що дозволяє використовувати один і той же код для розробки додатків для iOS, macOS, watchOS та tvOS.

SwiftUI - нова технологія, яка ще не має такої широкої підтримки та стабільності, як старіші фреймворки, такі як UIKit.

SwiftUI підтримується тільки на пристроях з iOS 13 та вище, тому це обмежує широкий охоплення цільової аудиторії.

У порівнянні з UIKit, SwiftUI може бути обмеженим за функціональністю, оскільки деякі функції можуть бути менш розвиненими або відсутніми.

Навіть якщо SwiftUI підтримує різні платформи Apple, він не забезпечує кросплатформеність для інших мобільних платформ, таких як Android.

Хоча SwiftUI пропонує інноваційний та декларативний підхід до створення інтерфейсу, він все ще розвивається і може мати обмежені можливості порівняно з іншими, більш відомими фреймворками.

2.1.2. Мова програмування Swift та його використання в SwiftUI

Мова програмування Swift від Apple була представлена в 2014 році і швидко набула популярності серед розробників. Вона є потужним і сучасним інструментом для створення мобільних додатків для пристроїв Apple, а також програмного забезпечення для інших платформ.

Swift була створена компанією Apple з метою вирішення багатьох проблем, які існували в Objective-C, попередній основній мові розробки для платформи Apple. Метою було створення більш сучасного, безпечного, ефективного та легкого в освоєнні інструменту для розробників.

Однією з ключових мет Swift було зробити процес програмування більш дружнім для розробників, зменшити кількість помилок в коді, забезпечити більш високий рівень безпеки та зробити розробку програмного забезпечення ефективнішою.

Swift пропонує чистий, легкий для сприйняття синтаксис, що спрощує вивчення мови новачкам.

Вбудовані в Swift механізми допомагають уникати багатьох типових помилок програмістів, що робить код безпечнішим.

Swift була оптимізована для високої швидкості виконання, що робить її дуже ефективною.

Мова підтримує динамічний та статичний режими програмування, що робить її гнучкою для різних типів проектів.

Swift має багатий функціонал, включаючи функціональне програмування та інші передові концепції.

Що стосується цілей, Apple вирішила розвивати Swift з метою постійного вдосконалення, надаючи розробникам потужний інструмент для створення програмного забезпечення для своїх платформ.

Swift стала важливою частиною екосистеми розробки програмного забезпечення Apple, дозволяючи розробникам створювати швидке, безпечне та сучасне програмне забезпечення для пристроїв Apple.

Ця мова програмування продовжує залучати увагу розробників та зберігається в активному розвитку, вдосконалюючи свої можливості та розширюючи свій функціонал. Порівняння Swift та Objective-C рис 2.2.

Metrics	Objective-C	Swift
Performance and Speed	Runtime-code reduces performance and speed, with the exception of C functions	High performance and best suited for performance-sensitive coding.
Code Safety	NOP due to null pointers can lead to bugs	Quick bug recognition and fixing
Convenient Maintenance	Two file maintenance	Single code file to be maintained
Syntax	Utilizes symbols and parentheses	English-like coding with simple coding
Code Length & Complexity	Verbose and lengthy	Concise coding with less lines of code for the same operation
Memory Management	ARC is only supported within Cocoa API	Uses Arc that Supports all APIs
Libraries Support	Static-library based with no support for dynamic libraries	Supports the integration of dynamic libraries
The Future of the Language	Apple backed, with an extensive developer's community	Rapidly growing programming language, with great potential for growth.

Source: objective-c vs swift

TEKREVOL

Рис 2.2. Порівняння Swift та Objective-C

Swift — це мова програмування, яка володіє чистим та консистентним синтаксисом, що дозволяє швидко та ефективно створювати програми. Розглянемо основні аспекти синтаксису та важливі функції Swift.

У Swift оголошення змінних відрізняється від оголошення констант. \`var\` використовується для змінних, а \`let\` — для констант:

````
```swift
var myVariable = 10
let myConstant = 20
```
````

~~~
Створення функцій у Swift — це просто та зручно:

```swift
func greet(person: String) -> String {
    return "Hello, \(person)!"
}

print(greet(person: "World")) // "Hello, World!"
```
~~~

Swift має концепцію опціоналів для роботи зі значеннями, які можуть бути `nil`:

~~~
```swift
var optionalString: String? = "Hello"
optionalString = nil
```
~~~

Swift надає потужні колекції, такі як масиви та словники, і зручні можливості ітерації:

~~~
```swift
var numbers = [1, 2, 3, 4, 5]

for number in numbers {
    print(number)
}

var namesDictionary = ["Alice": 25, "Bob": 30, "Charlie": 27]

for (name, age) in namesDictionary {
    print("\(name) is \(age) years old")
}```
~~~

Swift — це мова, що поєднує простоту та потужність, дозволяючи розробникам створювати чистий та ефективний код для різноманітних завдань. Її синтаксис та функціонал допомагають зменшити кількість помилок та прискорюють розробку програм.

Swift є ключовим компонентом для розробки інтерфейсів у SwiftUI, використовуючи свої функціональні можливості для опису моделей даних та створення динамічного UI.

Swift — це мова програмування, що надає масив можливостей для роботи з SwiftUI. Завдяки стандартам Swift, ви можете ефективно створювати, модифікувати та маніпулювати данними у SwiftUI.

SwiftUI використовує структури Swift для опису візуальних компонентів. Ви можете використовувати всі можливості мови, такі як розширення структур та протоколів, для створення динамічних виглядів та контролів.

```swift
struct ContentView: View {
    @State private var counter = 0

    var body: some View {
        VStack {
            Text("Counter: \(counter)")
            Button("Increment") {
                counter += 1
            }
        }
    }
}
```

Swift дозволяє вам легко визначати моделі даних та їх взаємозв'язок у SwiftUI. Ви можете використовувати класи, структури, протоколи та інші конструкції мови для створення моделей, які легко інтегруються з SwiftUI.

```swift
struct Task: Identifiable {
    var id = UUID()
    var title: String
    var completed = false
}

class TaskStore: ObservableObject {
    @Published var tasks = [Task]()
}

struct TaskListView: View {
    @ObservedObject var taskStore: TaskStore

    var body: some View {
        List(taskStore.tasks) { task in
            Text(task.title)
        }
    }
}
```

SwiftUI разом з Swift дозволяє створювати високоякісні додатки з високою продуктивністю та дружнім інтерфейсом. Вбудовані функції мови, такі як опціонали, типи колекцій, ітерації, а також чітка структура коду роблять процес розробки ефективнішим.

Swift та SwiftUI постійно розвиваються. Очікується подальше поліпшення та розширення можливостей обох технологій. Збільшення продуктивності, оптимізація пам'яті та швидкості виконання, а також розширення функціоналу для створення більш складних інтерфейсів - це основні напрямки розвитку.

SwiftUI разом з Swift надають потужний інструментарій для розробки мобільних додатків. Інтуїтивний підхід та швидкість розробки роблять їх привабливими для розробників. Разом з тим, очікується подальше розширення можливостей обох технологій, що забезпечить їхню актуальність та важливість у майбутньому.

Swift та SwiftUI - це не лише інструменти розробки, а й еволюція та удосконалення шляху до створення досконалих та функціональних мобільних додатків.

2.1.3. Компоненти та структура SwiftUI

Компоненти у SwiftUI є будівельними блоками для створення інтерфейсів користувача. Вони представляють собою основні елементи, які використовуються для створення різноманітних візуальних ефектів та взаємодії з користувачем. Ось огляд основних аспектів компонентів у SwiftUI:

SwiftUI базується на декларативному підході до створення інтерфейсів, що дозволяє описувати компоненти та їх взаємозв'язки. Компоненти включають елементи, такі як кнопки, тексти, картинки, текстові поля тощо.

SwiftUI використовує декларативний синтаксис для опису компонентів. Замість написання інструкцій, як іде реалізація, ви описуєте, як виглядає бажаний результат. Наприклад, для створення кнопки використовується простий код: `Button("Текст кнопки") { // дії при натисканні }`.

SwiftUI має різноманіття вбудованих компонентів, таких як `Text`, `Image`, `Button`, `TextField`, `List`, `NavigationView` та багато інших. Кожен з цих компонентів має власні параметри, які можна налаштовувати.

Основний принцип роботи з компонентами у SwiftUI - це їх комбінування для створення складніших інтерфейсів. Ви можете вкладати один компонент у інший, створюючи складні структури інтерфейсу. Наприклад, `VStack`, `HStack`, та `ZStack` - це контейнери для вертикального, горизонтального та змішаного розташування компонентів відповідно.

SwiftUI надає можливість змінювати вигляд компонентів за допомогою модифікаторів. Наприклад, можна змінювати розмір, колір, шрифт тексту тощо.

Компоненти у SwiftUI можуть взаємодіяти між собою та реагувати на події. Натискання на кнопку, введення тексту у текстове поле - це лише деякі приклади подій, які можна обробляти.

Структура та принципи роботи з компонентами в SwiftUI дозволяють створювати потужні та ефективні інтерфейси для мобільних додатків, роблячи розробку зручною та інтуїтивно зрозумілою для розробників.

Робота з текстовими елементами у SwiftUI відбувається за допомогою компонента `Text`, що дозволяє відображати текст у вигляді елемента інтерфейсу. `Text` можна налаштовувати за допомогою різноманітних модифікаторів для зміни розміру, кольору, шрифту та вирівнювання тексту. Наприклад:

````
```swift
Text("Привіт, SwiftUI!")
    .font(.title)
    .foregroundColor(.blue)
```
````

Для роботи з зображеннями та графікою використовується компонент `Image`. Щоб відобразити зображення, ви можете вказати ім'я файлу зображення або використати конструктор `Image(systemName:)` для відображення символів системних іконок. Приклад:

```swift
Image("myImage")
    .resizable()
    .frame(width: 100, height: 100)
```

Щоб вставити символьну іконку, використовуйте такий підхід:

```swift
Image(systemName: "heart.fill")
    .foregroundColor(.red)
    .font(.largeTitle)
```

SwiftUI також надає можливість маніпулювати графікою за допомогою форм та шляхів. Ви можете використовувати `Rectangle`, `Circle`, `Ellipse`, `Path` та інші вбудовані форми для створення різних графічних елементів. Наприклад:

```swift
Circle()
    .fill(Color.green)
    .frame(width: 100, height: 100)
```

Робота з текстом, зображеннями та графікою у SwiftUI відбувається через вбудовані компоненти та модифікатори, які дають можливість легко створювати та кастомізувати візуальні елементи у вашому додатку.

В SwiftUI є кілька потужних інструментів для роботи зі структурами даних та колекціями, які полегшують розробку додатків. Одним з ключових

компонентів є списки та таблиці, які можна легко налаштувати для відображення різноманітних даних.

SwiftUI підтримує використання різних структур даних для організації інформації у вашому додатку. Ви можете створювати свої власні моделі даних, що відображають різні об'єкти чи типи інформації.

SwiftUI має вбудовану підтримку списків. Ви можете легко створювати списки з різними типами вмісту, від тексту до складніших елементів. Використовуючи `List` і передавши йому вашу колекцію даних, ви можете створити простий список.

````
```swift
struct ContentView: View {
    let items = ["Item 1", "Item 2", "Item 3"]

    var body: some View {
        List(items, id: \.self) { item in
            Text(item)
        }
    }
}
```
````

SwiftUI також підтримує таблиці через `List` з різними рядками, що містять елементи. Ви можете використовувати це для створення складних макетів таблиць з різними видами вмісту.

````
```swift
struct Item: Identifiable {
    var id = UUID()
    var title: String
    var subtitle: String
}

struct ContentView: View {
    let items = [
        Item(title: "Title 1", subtitle: "Subtitle 1"),
        Item(title: "Title 2", subtitle: "Subtitle 2"),
        //...
    ]

    var body: some View {
        List(items) { item in
            VStack(alignment: .leading) {
                Text(item.title)
                    .font(.headline)
````
````

```
                Text(item.subtitle)
                    .font(.subheadline)
                    .foregroundColor(.gray)
            }
        }
    }
}
```

SwiftUI дозволяє створювати структури даних та їх візуальне представлення у вигляді списків і таблиць, що полегшує відображення та взаємодію з даними у вашому додатку.

У SwiftUI робота з формами та інтерфейсом користувача дуже зручна. Ви можете легко створювати форми для збору даних та додавати елементи, які відображають інформацію користувачу. Воткнута демонстрація простої форми:

```swift
struct ContentView: View {
    @State private var username = ""
    @State private var password = ""

    var body: some View {
        NavigationView {
            Form {
                Section(header: Text("Логін")) {
                    TextField("Ім'я користувача", text: $username)
                    SecureField("Пароль", text: $password)
                }

                Section {
                    Button("Увійти") {
                        // Додати логіку авторизації
                    }
                }
            }
            .navigationTitle("Вхід")
        }
    }
}
```

У цьому прикладі є форма з двома рядками для введення імені користувача та пароля. Ці дані прив'язані до властивостей `username` та `password`. `TextField` використовується для збору текстових даних, а

`SecureField` для паролів, що приховують введені дані. Крім того, використано кнопку для відправки цих даних.

Такі форми можна легко розширювати, додаючи до них різноманітні елементи, такі як `Picker`, `DatePicker` або інші елементи інтерфейсу користувача. Ви можете відображати додаткову інформацію, налаштовуючи вигляд та макети.

SwiftUI забезпечує простий та інтуїтивно зрозумілий спосіб взаємодії з користувачем та збору даних через форми, що дозволяє створювати дружні та функціональні інтерфейси для ваших додатків.

SwiftUI має потужні можливості для створення захоплюючих анімацій та ефектів у вашому інтерфейсі. Використання анімацій в SwiftUI - це відмінний спосіб зробити ваші додатки більш привабливими та користувацькими.

````
```swift
struct AnimationView: View {
    @State private var isAnimating = false

    var body: some View {
        VStack {
            Circle()
                .foregroundColor(.blue)
                .frame(width: isAnimating ? 200 : 100, height: isAnimating ? 200 : 100)
                .animation(.spring())
                .onTapGesture {
                    withAnimation {
                        isAnimating.toggle()
                    }
                }

            NavigationLink(destination: DetailView()) {
                Text("Перехід до екрану")
            }
        }
    }
}

struct DetailView: View {
    @State private var offset = CGSize.zero

    var body: some View {
        VStack {
            Text("Деталі екрану")
        }
        .offset(offset)
````
````

```
        .animation(.easeInOut(duration: 1.0))
        .onAppear {
            withAnimation {
                offset = CGSize(width: 0, height: 300)
            }
        }
    }
}
```

У цьому прикладі використано анімаційні ефекти на основі `animation`. Перший компонент представляє коло, розмір якого змінюється при тапі за допомогою `onTapGesture`. Другий компонент - це перехід на інший екран з анімацією зміни положення тексту.

SwiftUI надає можливості створення різноманітних ефектів, від простих змін розміру до складних анімаційних послідовностей. Це дає можливість реалізовувати інтерактивні та привабливі переходи та анімації в додатках SwiftUI, зробивши їх більш цікавими для користувачів.

Звичайно, SwiftUI надає безліч можливостей для побудови інтерфейсу. Ось декілька прикладів використання окремих компонентів SwiftUI:

Приклад 1: Текстове поле та кнопка

```swift
struct ContentView: View {
    @State private var text = ""

    var body: some View {
        VStack {
            TextField("Введіть текст", text: $text)
                .padding()
                .textFieldStyle(RoundedBorderTextFieldStyle())

            Button(action: {
                // Додати вашу логіку тут
            }) {
                Text("Натисніть кнопку")
                    .padding()
                    .background(Color.blue)
                    .foregroundColor(.white)
                    .cornerRadius(8)
            }
        }
        .padding()
    }
}
```

Приклад 2: Список елементів

```swift
struct ContentView: View {
    let items = ["Перший", "Другий", "Третій", "Четвертий"]

    var body: some View {
        List(items, id: \.self) { item in
            Text(item)
        }
    }
}
```

Приклад 3: Компоненти стекування

```swift
struct ContentView: View {
    var body: some View {
        VStack {
            Text("Це верхній текст")
                .font(.largeTitle)
                .padding()

            Spacer() // Додає проміжок між верхнім та нижнім текстом

            Text("Це нижній текст")
                .font(.headline)
                .padding()
        }
    }
}
```

Ці приклади демонструють базові компоненти SwiftUI, такі як `TextField`, `Button`, `List`, та основні контейнери, такі як `VStack`, які використовуються для розміщення та організації інтерфейсу. SwiftUI дозволяє легко поєднувати ці компоненти, створюючи багатофункціональні та красиві інтерфейси для додатків.

Звісно, оптимізація та використання кращих практик є важливими аспектами розробки SwiftUI. Ось кілька кращих практик та методів оптимізації:

Використання `@State`, `@Binding` та `@ObservedObject` з усвідомленим підходом:

- `@State` використовується для зберігання стану в представленні.

- `@Binding` дозволяє вам передавати значення `@State` між різними представленнями.

- `@ObservedObject` використовується для спостереження за змінами об'єктів, відображаючи їх зміни в представленнях.

- Захист від невиправданого оновлення UI зайвою роботою зі станами.

- Використання `Equatable` для оптимізації оновлення компонентів.

- `GeometryReader` дозволяє звертатися до розмірів батьківського представлення та працювати з ними.

- Використання лінивого завантаження для завантаження даних лише тоді, коли вони потрібні.

- Використання кешування для уникнення повторного завантаження даних, якщо вони не змінилися.

- Використання підтримки попередніх станів для оптимізації роботи з анімаціями та переходами між станами.

- Використання інструментів профілювання SwiftUI для виявлення та виправлення можливих буттілнеків.

- Проведення тестування швидкодії для впевненості у продуктивності вашого додатка.

- Використовуйте `onAppear` та `onDisappear` для оптимізації ресурсів та реакції на зміни стану представлень.

Використовуючи ці практики та методи оптимізації, розробники можуть підвищити продуктивність та ефективність додатків, створених з використанням SwiftUI.

SwiftUI і його компоненти грають критичну роль у розробці мобільних додатків, забезпечуючи зручний, ефективний та швидкий спосіб створення інтерфейсів. Важливість компонентів полягає в їхній гнучкості, яка дозволяє швидко реагувати на зміни та створювати складні інтерфейси з мінімальними зусиллями.

Однією з ключових переваг компонентів є їхні можливості адаптації до різних пристроїв та розмірів екранів. Це важливо в умовах швидкозмінного світу мобільних технологій, де різні пристрої мають різні розміри екранів та можливості.

Щодо перспектив розвитку, компоненти SwiftUI відкривають шлях для постійного росту та удосконалення. Очікується подальше розширення набору доступних компонентів та покращення їх функціональності. Розвиток інструментів для роботи з анімаціями, оптимізація швидкості та покращення підтримки нових функцій підвищать привабливість SwiftUI для розробників.

У світлі швидкого росту мобільних технологій і надзвичайної динаміки ринку, компоненти SwiftUI відіграють ключову роль у забезпеченні зручності та ефективності розробки, а їхні можливості постійно зростають для забезпечення найвищого рівня користувацького досвіду в мобільних додатках.

### 2.1.4. Основні можливості та функціонал SwiftUI для створення інтерфейсів

SwiftUI є потужним інструментом для розробки користувацьких інтерфейсів у мобільних додатках. Основна його мета - надати розробникам інтуїтивний, декларативний підхід до створення інтерфейсів, де кожен елемент відображення залежить від стану даних, що змінюється.

Цей фреймворк дозволяє будувати складні інтерфейси з мінімальним кодом, використовуючи декларативний синтаксис. Замість того, щоб зосереджуватися на тому, як розміщати елементи на екрані та керувати їх розмірами, розробники можуть описати, як повинен виглядати інтерфейс в певному стані або у відповідь на певні події.

SwiftUI надає широкий набір вбудованих компонентів і контейнерів, таких як текстові поля, кнопки, списки, контейнери для макетів, що робить його потужним інструментом для створення різних типів інтерфейсів. Можливість перегляду змін у реальному часі завдяки прев'ю дозволяє швидко вирішувати, як зміни у коді впливають на вигляд додатка.

Враховуючи його простоту, потужність та зручність використання, SwiftUI стає популярним фреймворком серед розробників, які прагнуть швидко створювати сучасні та естетичні інтерфейси для мобільних додатків.

SwiftUI пропонує широкий спектр основних елементів, які дозволяють розробникам створювати різноманітні та ефективні інтерфейси для мобільних додатків. Основні елементи цього фреймворку включають у себе:

Views - Основний будівельний блок SwiftUI. Вони представляють елементи візуального відображення, які можуть бути текстом, кнопкою, списком тощо.

NavigationView - Контейнер, який дозволяє створювати навігаційні ієрархії в додатках, такі як стеки екранів і панелі навігації.

Text - Елемент для відображення тексту зі специфічними параметрами форматування, такими як розмір шрифту, колір та стиль.

Image - Для відображення графіки та зображень в додатку.

Lists - Компонент для показу списку даних, де кожен елемент може бути індивідуально налаштованим.

Stacks (VStack, HStack, ZStack) - Контейнери для організації елементів у вертикальний, горизонтальний або затенений спосіб.

Modifiers - Методи, які застосовують зміни до вигляду та поведінки елементів, такі як зміна кольору, шрифту, фону та розміру.

Spacer - Елемент для розміщення простору між елементами.

Кожен з цих елементів використовується для побудови складних інтерфейсів, дозволяючи розробникам легко створювати та налаштовувати вигляд додатка без необхідності поглибленої роботи з кодом.

Модифікатори в SwiftUI - це потужний інструмент, який дозволяє легко змінювати вигляд та поведінку компонентів. Вони застосовуються до візуальних елементів для задання різноманітних параметрів та стилів. Ось деякі основні поняття та можливості модифікаторів:

Можна створювати власні модифікатори для реюзу коду. Наприклад, створіть модифікатор для стилізації кнопки:

~~~~
```swift
struct CustomButtonStyle: ViewModifier {
    func body(content: Content) -> some View {
        content
            .font(.title)
            .foregroundColor(.white)
            .padding()
            .background(Color.blue)
            .cornerRadius(8)
    }
}

extension View {
    func customButtonStyle() -> some View {
        self.modifier(CustomButtonStyle())
    }
}
```

```swift
Button("Click me") {
    // Action
}
.customButtonStyle()
```
~~~~

Це дозволяє легко та зрозуміло стилізувати компоненти в вашому додатку SwiftUI, додаючи необхідний вигляд та функціональність до елементів інтерфейсу.

Організація компонентів у SwiftUI відбувається за допомогою різних контейнерів і структур, які дозволяють ефективно організовувати та компонувати елементи інтерфейсу. Ось деякі з найпоширеніших методів організації компонентів:

- VStack (Vertical Stack) та HStack (Horizontal Stack) дозволяють організовувати компоненти вертикально або горизонтально відповідно. Наприклад:

````
```swift
VStack {
    Text("Верхній рядок")
    Text("Нижній рядок")
}
```
````

- ZStack (Z-axis Stack) дозволяє розміщувати компоненти один на одному, утворюючи шари. Це дозволяє створювати накладені ефекти або групи компонентів. Наприклад:

````
```swift
ZStack {
    Rectangle()
        .fill(Color.blue)
        .frame(width: 200, height: 200)
    Circle()
        .fill(Color.yellow)
        .frame(width: 100, height: 100)
}
```
````

- Group дозволяє групувати компоненти разом, щоб керувати їхнім виглядом та поведінкою як одним елементом:

````
```swift
Group {
    Text("Елемент 1")
    Text("Елемент 2")
}
```
````

- Form та List використовуються для відображення структурованої інформації у вигляді списку або форми. Вони автоматично організують їх у вигляді рядків або груп.

````
```swift
Form {
````
````

```
        Section(header: Text("Секція 1")) {
            Text("Елемент 1")
            Text("Елемент 2")
        }
        Section(header: Text("Секція 2")) {
            Text("Елемент 3")
            Text("Елемент 4")
        }
    }
```

- SwiftUI має різні види стеків, такі як ZStack, VStack, HStack. Вони можуть бути вкладені один в одного для складних макетів та організації елементів.

Користуючись цими інструментами організації, ви можете ефективно створювати та компонувати складні інтерфейси в SwiftUI, надаючи легкість та гнучкість у створенні різних макетів та організації компонентів.

Робота з формами та введенням даних у SwiftUI включає в себе кілька елементів, які дозволяють збирати та обробляти інформацію, що вводиться користувачем. Ось кілька кроків для створення та використання форм у SwiftUI:

TextField використовується для введення текстових даних, тоді як SecureField служить для введення паролів чи конфіденційної інформації, яка повинна бути захищена.

```swift
@State private var username = ""
@State private var password = ""

var body: some View {
    VStack {
        TextField("Введіть ім'я", text: $username)
            .textFieldStyle(RoundedBorderTextFieldStyle())
        SecureField("Пароль", text: $password)
            .textFieldStyle(RoundedBorderTextFieldStyle())
        Button("Увійти") {
            // Реакція на натискання кнопки, обробка введених даних
            // Наприклад, перевірка на відповідність збереженим даним
        }
    }
}
```

Picker дозволяє вибирати один або багато варіантів з розкривного списку або зі списку з вибором.

```swift
struct ContentView: View {
    let colors = ["Червоний", "Зелений", "Синій"]
    @State private var selectedColor = 0

    var body: some View {
        Picker("Виберіть колір", selection: $selectedColor) {
            ForEach(0 ..< colors.count) {
                Text(self.colors[$0])
            }
        }
    }
}
```

Використання умов та перевірок для перевірки правильності введених даних перед їхнім використанням або збереженням.

```swift
Button("Зберегти") {
    if username.isEmpty || password.isEmpty {
        // Повідомлення про помилку, якщо не всі поля заповнені
    } else {
        // Логіка для збереження даних
    }
}
```

Інтерфейси SwiftUI надають можливості для створення форм та збору даних з використанням різних елементів. Реалізація валідації та обробки введених даних дозволяє створювати більш інтерактивні та користувацьки-орієнтовані додатки.

- Кешуйте результати запитів до сервера або інших обчислень, щоб уникнути повторного обчислення.
- Використовуйте `onAppear` та `onDisappear` для завантаження або видалення ресурсів при зміні екранів.

- Спробуйте уникати глибоких вкладень відображень, оскільки це може призвести до складності управління та збільшення часу відгуку інтерфейсу.

- Для великих списків даних використовуйте `EquatableView` для уникнення зайвих перерисувань компонентів.

- Реалізуйте протокол `Identifiable` для ефективної ідентифікації елементів списку.

- Вимикаючи анімації для непотрібних змін, ви зменшуєте використання ресурсів та поліпшуєте продуктивність.

- Використовуйте тести для виявлення та виправлення проблем у виконанні.

- Використовуйте інструменти профілювання для визначення найбільш ресурсоємких частин вашого коду.

Інтерфейси, розроблені відповідно до цих кращих практик, будуть працювати ефективно та ефектно, забезпечуючи приємний досвід користувача.

SwiftUI - це молодий, але вже значно розвинений фреймворк для розробки інтерфейсів у додатках для iOS, macOS, watchOS та tvOS. Ось його переваги та можливості:

SwiftUI пропонує декларативний підхід до створення інтерфейсів, де ви описуєте, як виглядає ваш інтерфейс, а не як його створити.

Завдяки декларативному підходу, розробка інтерфейсу є швидкою і простою, з меншою кількістю коду.

SwiftUI надає різноманіття компонентів і можливостей, таких як Stack, ScrollView, Lists, Forms, Animations тощо.

Режим перегляду в реальному часі дозволяє бачити зміни відразу під час розробки.

SwiftUI інтегрується з дизайном та робить його узгодженим для різних пристроїв.

Очікується постійне розширення набору компонентів та можливостей.

SwiftUI буде інтегровано в майбутні платформи, що виходять, розширюючи його можливості.

Швидкість та продуктивність роботи з SwiftUI постійно покращується, а також вирішуються поточні проблеми.

Очікується розширення вбудованих анімаційних можливостей та ефектів.

SwiftUI - це потужний інструмент для швидкої і ефективної розробки інтерфейсів, і має значний потенціал для подальшого розвитку та покращення.

### 2.1.5. Принципи роботи анімацій та графічного візуалу в SwiftUI

У SwiftUI анімація стала ключовим елементом для створення захоплюючих та інтерактивних інтерфейсів. Основні принципи анімації у SwiftUI базуються на декларативному підході та простоті роботи з анімаційними ефектами.

SwiftUI пропонує декларативний синтаксис для опису анімацій. Ви описуєте, як елементи інтерфейсу змінюються в часі, а не як саме ці зміни виконуються.

SwiftUI надає різноманітні модифікатори для анімації. Наприклад, `animation()` модифікатор для зміни властивостей елементів з використанням анімаційних ефектів.

SwiftUI використовує імпліцитні анімації для автоматичної анімації зміни властивостей елементів, що є основною частиною декларативності.

Використовуючи анімаційні контролери, такі як `withAnimation`, можна контролювати, коли та які зміни властивостей елементів будуть анімовані.

SwiftUI пропонує можливості для створення анімаційних переходів між екранами, а також використання різних ефектів, таких як затемнення, зміна прозорості, рух тощо.

Анімація в SwiftUI дає можливість створювати затягуючі та динамічні інтерфейси, що відображають зміни в реальному часі. Це потужний інструмент для створення відчутних та захоплюючих вражень у користувачів.

Анімаційні ефекти є ключовим елементом для залучення уваги користувачів та покращення візуального досвіду. У SwiftUI використання анімаційних ефектів є досить простим та декларативним. Ось кілька деталей, які можуть допомогти використовувати анімації ефективно:

SwiftUI використовує імпліцитні анімації для автоматичної анімації змін властивостей елементів. Наприклад, при зміні розміру елемента, його положення чи кольору, зміни відбуваються з анімацією за замовчуванням.

Для керування типом анімації та її параметрами можна використовувати модифікатор `animation()`. Це дозволяє налаштовувати спосіб та тривалість анімацій.

SwiftUI надає можливості для створення анімаційних переходів між екранами. Ви можете використовувати `NavigationView` та `NavigationLink`, щоб створювати анімаційні переходи.

Використання анімацій для графічних ефектів, таких як затемнення, розмиття, зміна прозорості, може створити враження глибшого взаємодії з елементами інтерфейсу.

Використання анімаційних ефектів у SwiftUI дозволяє створювати динамічні, ефектні та захоплюючі інтерфейси, які залучають увагу користувачів та покращують їхній досвід.

Анімація переходів між екранами в SwiftUI може надати інтерфейсу вашого додатка вражаючий та динамічний вигляд. Ось кілька ключових аспектів анімації переходів:

Використовуючи `NavigationView`, ви можете створювати стеки екранів. `NavigationLink` дозволяє переходити між екранами, а анімації будуть автоматично виконуватися при зміні вмісту.

````
```swift
NavigationView {
    NavigationLink(destination: DetailView()) {
        Text("Go to Detail")
    }
}
```
````

SwiftUI надає можливість використовувати анімації за допомогою `NavigationLink` та переходів з власними стилізованими ефектами.

````
```swift
NavigationLink(destination: DetailView()) {
    Text("Go to Detail")
        .padding()
        .background(Color.blue)
        .foregroundColor(.white)
        .cornerRadius(8)
        .transition(.slide)
}```
````

Ви також можете створити власні екрани та анімації, користуючись знаннями про анімацію та роботу з модифікаторами.

````
```swift
NavigationView {
    VStack {
        NavigationLink(destination: DetailView()) {
            Text("Go to Detail")
        }
    }
}
```
````

Використання `onAppear` та `onDisappear` для створення власних переходів між екранами при зміні стану елементів.

````
```swift
NavigationView {
    Text("Home View")
````

```
        .onTapGesture {
            withAnimation {
                // Change state to present DetailView
            }
        }
}
```

Незалежно від того, чи ви користуєтеся вбудованими переходами, чи створюєте власні анімації, SwiftUI надає широкі можливості для створення різноманітних та захоплюючих переходів між екранами, що покращують користувацький досвід вашого додатка.

SwiftUI має вбудовані анімаційні можливості та модифікатори, які дозволяють легко створювати анімації та змінювати вид компонентів у відповідь на зміни стану. Ось кілька ключових анімаційних ефектів та модифікаторів:

Модифікатор, який визначає, як буде виконуватися анімація при зміні стану. Можна вибрати різні види анімацій, такі як `.default`, `.spring()`, `.easeInOut` тощо.

```swift
@State private var isAnimating = false

Button("Animate") {
    withAnimation {
        isAnimating.toggle()
    }
}
```

Ці модифікатори дозволяють анімувати обертання та масштабування елементів.

```swift
Image(systemName: "heart.fill")
    .scaleEffect(isAnimating ? 2 : 1)
    .animation(.easeInOut)
    .onTapGesture {
        withAnimation {
            isAnimating.toggle()
        }
    }
```

Використовуючи ці модифікатори, можна контролювати прозорість або прихованість елементів.

~~~
```swift
Image(systemName: "star")
    .opacity(isAnimating ? 0.5 : 1)
    .onTapGesture {
        withAnimation {
            isAnimating.toggle()
        }
    }
```
~~~

Модифікатор, що змінює позицію елемента. Застосування анімації до зміни позиції.

~~~
```swift
Rectangle()
    .frame(width: 100, height: 100)
    .foregroundColor(.blue)
    .offset(y: isAnimating ? 100 : 0)
    .animation(.spring())
    .onTapGesture {
        withAnimation {
            isAnimating.toggle()
        }
    }
```
~~~

SwiftUI забезпечує ряд потужних засобів для створення графічних елементів та візуального відображення. Використовуючи Shape, Path та GeometryReader, ви можете створювати власні форми, простори, а також адаптувати їх до розмірів віджетів та екранів.

Адаптивність та респонсивний дизайн - це важливі аспекти в розробці інтерфейсів, особливо коли ми маємо справу з різними розмірами пристроїв та їх орієнтаціями. SwiftUI надає ряд зручних інструментів для створення адаптивних інтерфейсів. Ось деякі методи, які можна використовувати:

~~~
```swift
~~~

```
GeometryReader { geometry in
    VStack {
        Text("Adaptive UI")
            .font(.title)
            .frame(width: geometry.size.width * 0.8)
        Spacer()
    }
}
```

`GeometryReader` дозволяє зчитувати розмір батьківського контейнера та адаптувати елементи відповідно до цього розміру.

```swift
HStack {
    Text("Left Content")
    Spacer()
    Text("Right Content")
}
```

`Spacer()` використовується для автоматичного розміщення вмісту в доступному просторі, що дозволяє створювати гнучкі інтерфейси, які адаптуються до різних розмірів екрану.

```swift
Text("Title")
    .font(.title)
    .minimumScaleFactor(0.5)
    .lineLimit(1)
```

Це дозволяє тексту автоматично зменшуватися, якщо він не поміщається в доступний простір.

SwiftUI має багато контейнерів (VStack, HStack, ZStack) та методи розташування (alignment, spacing), що дозволяють створювати респонсивний дизайн шляхом гнучкого розміщення елементів.

SwiftUI надає декілька графічних елементів, які автоматично адаптують свої розміри до вмісту або екрану (наприклад, `ScrollView`, `List`).

Оптимізація анімацій в SwiftUI є ключовим етапом для покращення продуктивності та відчутності додатку. Ось декілька порад:

```swift
```

````
withAnimation {
    // Ваші зміни в анімації
}
```
````

Це дозволяє узгодити зміни з анімацією та оптимізує їх виконання.

````
```swift
.transaction {
    $0.animation = .easeInOut
}
```
````

Це дозволяє керувати параметрами анімації, такими як швидкість, тип та тривалість.

````
```swift
struct MyData: Equatable {
    // Ваші властивості
}
```
````

Це допоможе уникнути непотрібної перерисовки елементів, якщо дані не змінилися.

````
```swift
@State private var isAnimating = false
```
````

Використання `@State` або `@Binding` допомагає SwiftUI розуміти, коли оновлювати вигляд.

Анімація може бути поділена на окремі частини, які працюють паралельно, щоб зменшити вплив на продуктивність.

Складні анімації можуть затратити багато ресурсів. Розгляньте спрощення анімацій для поліпшення продуктивності.

Ці практичні поради можуть допомогти оптимізувати ваші анімації в SwiftUI, покращити їх продуктивність та зменшити негативний вплив на продуктивність додатку.
````

2.1.6. Інструменти та середовище розробки у SwiftUI

Низка інструментів підтримують розробку на SwiftUI та допомагають з підтримкою коду, прототипуванням та налагодженням додатків. Ось кілька ключових інструментів:

Xcode — це основне інтегроване середовище розробки для роботи з SwiftUI. Він надає зручну робочу область для розробки, редагування та перегляду інтерфейсів, а також дозволяє швидко створювати та тестувати додатки.

Це вбудований інструмент у Xcode, який дозволяє переглядати реальний вигляд та взаємодію інтерфейсу без запуску додатка. Він пропонує миттєві зміни та перегляд результатів у реальному часі.

Playgrounds дозволяє вам вчитися на прикладах, створюючи окремі розділи з різними елементами. Ви можете експериментувати зі складнішими концепціями та швидко перевіряти їхню дію.

Для розробників це може бути ідеальним інструментом для швидкої прототипізації ідей або для тестування нових концепцій інтерфейсу без необхідності створювати повноцінний проект у Xcode.

Playgrounds можуть використовуватися для створення інтерактивних завдань чи візуалізації графіки з використанням SwiftUI.

Додатково, Swift Playgrounds надає доступ до різноманітних навчальних матеріалів, включаючи різноманітні туторіали та завдання для вивчення Swift і SwiftUI.

Swift Playgrounds - це важливий інструмент для навчання та розробки в області Swift та SwiftUI. Він дозволяє експериментувати з кодом у безпечному середовищі та швидко бачити результати, що робить його відмінним вибором для навчання і прототипування проектів SwiftUI.

Xcode - це інтегроване середовище розробки (IDE), яке надає широкі можливості для створення програм для платформ Apple, включаючи роботу з SwiftUI. Ось детальніше про взаємодію Xcode з SwiftUI:

- Xcode має вбудований редактор SwiftUI, що дозволяє створювати та редагувати інтерфейс в реальному часі. Ви бачите зміни відразу ж, коли змінюєте код, завдяки Live Preview.

- Редактор SwiftUI в Xcode дозволяє швидко створювати компоненти, додавати модифікатори та відстежувати структуру ієрархії ваших елементів інтерфейсу.

- Миттєвий Live Preview дозволяє бачити, як ваш інтерфейс виглядає на різних пристроях та в різних режимах. Для швидкого виявлення помилок використовується вбудований дебагер.

- Canvas - це частина Xcode, яка показує прев'ю вашого інтерфейсу в режимі реального часу, дозволяючи вам взаємодіяти з кодом та відслідковувати його зміни відразу ж.

- Xcode має набір інструментів для створення інтерфейсу, таких як редактор кольорів, менеджер ресурсів, а також інструменти для створення анімацій та взаємодії.

- З Xcode ви можете створювати мобільні додатки для iOS, iPadOS, macOS, watchOS та tvOS за допомогою SwiftUI.

- Xcode ідеально поєднується з іншими інструментами Apple, такими як Interface Builder, Xcode Server, TestFlight для тестування та розгортання програм тощо.

- Xcode має вбудовану документацію та багато матеріалів для навчання SwiftUI, що допомагає вам вивчити фреймворк і його можливості.

Xcode забезпечує широкі можливості для роботи з SwiftUI, забезпечуючи інструменти для швидкого прототипування, дизайну та розробки програм для платформ Apple, і допомагає прискорити процес розробки за допомогою SwiftUI.

SwiftUI надає кілька корисних засобів для налагодження та відлагодження коду, які допомагають в розробці інтерфейсів. Давай розглянемо основні можливості та інструменти, що допомагають в цьому процесі.

SwiftUI має можливість миттєвого попереднього перегляду змін, які ви вносите в код. Інтерфейс Xcode показує зміни в реальному часі, дозволяючи бачити, як ваш інтерфейс змінюється під час редагування коду.

Це інтерактивне середовище, де ви можете маніпулювати UI-елементами, додавати нові, налаштовувати їх властивості та відслідковувати зміни у реальному часі.

Вбудований дебагер Xcode дає можливість вставляти точки зупинки, відстежувати значення змінних та виконувати код крок за кроком, допомагаючи виправити помилки.

Xcode показує повідомлення про помилки у коді, що дозволяє швидко знайти та виправити проблеми.

Використовуючи вбудований в SwiftUI прев'ю, ви можете розглядати різні стани інтерфейсу, перевіряти різні конфігурації та реакцію на дані в реальному часі.

Цей інструмент у Xcode дозволяє вам досліджувати властивості та інформацію про об'єкти безпосередньо на візуальному представленні.

Можливість слідкувати за змінами в реальному часі дозволяє відлагоджувати і відслідковувати значення певних властивостей та моделей даних.

Інструменти вбудованого тестування дозволяють перевіряти різні стани ваших компонентів і їх відповідь на вхідні дані.

Ці засоби у поєднанні створюють потужне середовище для відлагодження та налагодження коду SwiftUI, що полегшує процес розробки та дозволяє швидше виявляти та виправляти помилки.

SwiftUI має декілька можливостей для спільної роботи та інтеграції з іншими інструментами, що робить його дружнім до різних екосистем розробки.

SwiftUI працює відмінно з різними VCS, такими як Git, SVN тощо. Всі файли, пов'язані з проектом SwiftUI, можна легко включити в систему контролю версій для спільної роботи команди та відстеження змін.

SwiftUI і Xcode мають високий рівень сумісності. Xcode, як основне середовище розробки для Swift, відмінно підтримує інструменти та можливості SwiftUI.

SwiftUI може легко інтегруватися зі сторонніми бібліотеками та фреймворками Swift, що розширює його функціональність та можливості.

SwiftUI підтримує міжплатформену роботу з UIKit та Objective-C, що дозволяє поєднувати існуючий код та бібліотеки з відносно новим інтерфейсом SwiftUI.

SwiftUI може бути легко інтегрований з Swift Package Manager (SPM), що дає змогу керувати залежностями та додатковими пакетами для проекту.

SwiftUI дозволяє створювати код, який працює як на iOS, так і на macOS, що спрощує процес розробки та підтримки.

Є експериментальні проекти, які дозволяють використовувати SwiftUI для веб-розробки, що розширює його можливості для створення різноманітних інтерфейсів.

SwiftUI не лише надає розширені можливості для розробки, але й легко інтегрується з іншими інструментами, забезпечуючи гнучкість та можливості спільної роботи з різними екосистемами розробки.

Розробка у SwiftUI може бути ефективною, якщо використовувати деякі стратегії та інструменти, що спрощують процес. Ось кілька порад для оптимізації розробки:

Xcode — основний інструмент для розробки SwiftUI, оскільки надає вбудовану підтримку для нього, включаючи редактор коду з функціями попереднього перегляду.

Використання SwiftUI Preview у Xcode для миттєвого попереднього перегляду змін у реальному часі, що спрощує візуалізацію структури додатка.

Розділення різних частин додатку на окремі модулі дозволяє зберігати код чистим і легкоуправнім.

SwiftUI дозволяє реалізовувати інтерфейс різними способами. Проводіть експерименти з різними підходами до створення відображень.

Використовуйте відкриті рішення, які забезпечують готовий функціонал, наприклад, бібліотеки SwiftUI, щоб прискорити процес розробки.

Використання систем контролю версій, таких як Git, сприяє спільній роботі та управлінню версіями проекту.

Тестування коду допомагає виявити помилки та забезпечити стабільність додатку.

Уникайте зайвих обчислень та неефективних анімацій, оскільки вони можуть вплинути на продуктивність.

SwiftUI — це розвиваючася технологія, тому важливо бути в курсі оновлень та нововведень.

Засоби для оптимізації розробки у SwiftUI постійно розширюються, тому важливо бути відкритим до нових можливостей та інструментів для полегшення процесу створення додатків.

Тестування є важливою складовою процесу розробки будь-якої програми, включно з додатками, розробленими на SwiftUI. Ось огляд основних можливостей та засобів тестування для SwiftUI:

Це тести, які перевіряють окремі компоненти програми, які можна відокремити для тестування. Для SwiftUI це можуть бути окремі вигляди

(Views), моделі даних або взаємодія між різними частинами програми. Використовуються звичайні фреймворки тестування Swift, такі як XCTest, для написання та виконання unit-тестів.

Це тести, які перевіряють користувацький інтерфейс програми. Для SwiftUI ці тести можуть бути використані для перевірки відображення, взаємодії та навігації між різними екранами. В Xcode є вбудований інструмент для створення та виконання UI-тестів — XCTest framework має можливості для UI-тестів з SwiftUI.

SwiftUI Preview дозволяє створювати попередній перегляд виглядів (Views) під час розробки. Ці попередні перегляди можна використовувати для відлагодження та швидкого тестування окремих компонентів.

SwiftUI базується на спостереженнях та подіях. Тестування відповіді компонентів на певні події може бути корисним для впевненості в їх правильному функціонуванні.

Іноді можна використовувати сторонні фреймворки, такі як Nimble або Quick, для полегшення написання тестів та розширення функціоналу для тестування SwiftUI-додатків.

Тестування у SwiftUI спрощується завдяки вбудованим можливостям попереднього перегляду, однак підтримка звичайних фреймворків тестування також дозволяє здійснювати детальне тестування ваших додатків.

Розвиток у SwiftUI може бути захоплюючим та продуктивним завдяки доступності великої кількості ресурсів та спільнот. Ось деякі рекомендації та корисні ресурси:

Офіційна документація Apple Це основний джерело інформації про SwiftUI. Офіційна документація містить розгорнуті приклади, посібники та описи функціональності. Вона постійно оновлюється та надає останні оновлення та підходи до SwiftUI.

Apple надає низку посібників та туторіалів для вивчення SwiftUI через Xcode Playgrounds. Ці ресурси включають практичні завдання та вправи для закріплення знань.

Існують книги та курси, призначені для вивчення SwiftUI від початку до більш глибоких тем. Такі ресурси можуть допомогти вам отримати систематизоване розуміння платформи.

Існують великі спільноти розробників на Reddit, Stack Overflow та інших форумах, де ви можете отримати відповіді на свої питання, а також поділитися своїм досвідом та знаннями.

Освітній контент у вигляді відеоуроків на платформах, таких як YouTube або спеціалізовані сайти, може бути корисним для вас, оскільки візуальний контент часто легше засвоюється.

Розгляньте вивчення відкритого коду вже існуючих додатків або проектів на GitHub, де ви зможете побачити, як використовується SwiftUI у реальних проектах.

Вони можуть бути корисними для збагачення знань та вивчення нових підходів від досвідчених розробників у сфері мобільних додатків.

Здатність знаходити і використовувати ці ресурси допоможе вам ефективно розвивати навички у SwiftUI та впроваджувати їх у практиці.

2.1.7. Взаємодія зі SwiftUI через архітектурні патерни та підходи

Архітектурні патерни є фундаментом структурування та розробки програмного забезпечення. Вони визначають спосіб організації коду, щоб забезпечити чистоту, читабельність, тестованість та розширюваність додатків. Ось огляд деяких популярних патернів у контексті мобільних додатків: Model-View-Controller (MVC): Модель (Model) відповідає за дані та логіку додатка. Вид (View) відображає дані користувачеві та реагує на дії. Контролер (Controller) керує взаємодією між Моделлю та Видом.

Model-View-ViewModel (MVVM): Модель (Model) те ж саме, що й у MVC. Вид (View) відображає дані та реагує на події, але не має прямого доступу до Моделі. ViewModel використовується для комунікації між Моделлю та Видом, виконання бізнес-логіки та підготовки даних для відображення.

VIPER (View, Interactor, Presenter, Entity, Router): Вид (View) відповідає за відображення даних та взаємодію з користувачем. Інтерактор (Interactor) містить бізнес-логіку та обробку даних. Презентер (Presenter) координує Вид та Інтерактор, а також обробляє дані перед їх відображенням. Сутність (Entity) об'єкти даних, що використовуються в додатку. Маршрутизатор (Router)** відповідає за навігацію між модулями додатку.

Clean Architecture - Presentation Layer містить відображення та презентери (інтерфейси для користувача). Domain Layer - містить бізнес-логіку та правила. Data Layer відповідає за отримання та зберігання даних.

Redux - Це контейнер для стану додатку та прогностичного оновлення його у відповідь на дії користувача.

Вибір архітектурного патерну залежить від специфіки проекту, його розміру, потреб користувачів та інших факторів. Кожен паттерн має свої переваги та недоліки, і використання їх відповідно до потреб проекту є важливим етапом розробки.

2.2. Розбір методів роботи нативного фреймворку розробки Android додатків Jetpack Compose.

2.2.1. Введення в Jetpack Compose: переваги та концепції

Jetpack Compose — це сучасна бібліотека для розробки інтерфейсів користувача для платформи Android від Google. Вона була випущена з ме-

тою спростити та поліпшити процес розробки інтерфейсів, надати більш чистий та декларативний підхід до створення UI. Основні характеристики Jetpack Compose:

Замість імперативного підходу, де ви описуєте кроки для створення UI, використовується декларативний підхід, де ви описуєте, як UI повинен виглядати для різних станів.

UI в Jetpack Compose створюється за допомогою функцій, які повертають компоненти, наприклад, кнопки, тексти, списки тощо.

Його простий, декларативний підхід дозволяє швидко створювати та змінювати інтерфейси.

Легко інтегрується з існуючими проектами Android, дозволяючи поступово використовувати нові функції без необхідності переписування всього коду.

Забезпечує простий спосіб робити анімації та відображати реакцію на події.

Компоненти автоматично оновлюються, якщо змінюється їх стан.

Jetpack Compose відкриває нові можливості для розробників Android, надаючи зручний, швидкий та потужний спосіб створення інтерфейсів. Ця бібліотека постійно розвивається та отримує нові функції, спрямовані на полегшення та покращення роботи з інтерфейсами користувача на платформі Android. Стратегія міграції Jetpack Compose представлена на рис. 2.3.

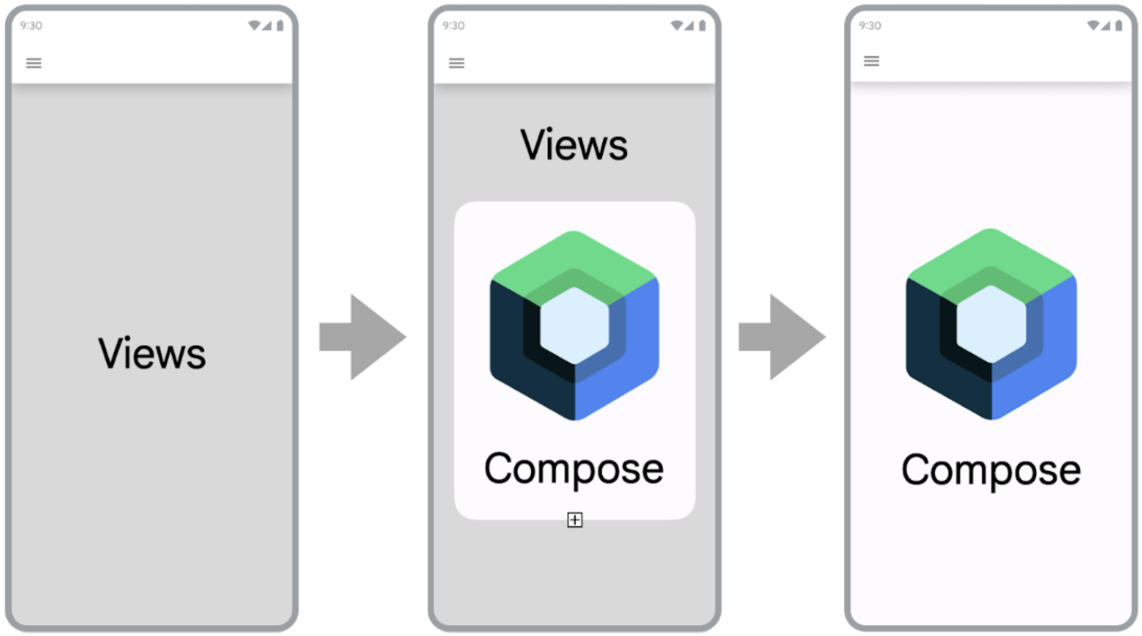


Рис 2.3. Стратегія міграції Jetpack Compose

Jetpack Compose, новий інструмент для розробки інтерфейсів на платформі Android, має кілька переваг порівняно з традиційними методиками розробки інтерфейсів:

Однією з основних переваг Jetpack Compose є його декларативний підхід до створення інтерфейсу користувача. Замість імперативного програмування, розробники описують, як має виглядати інтерфейс у різних станах. Це дозволяє розробникам швидше та зрозуміліше описувати UI.

Compose робить процес створення UI більш простим і інтуїтивним. Функційний підхід дозволяє створювати і перевикористовувати компоненти з меншим обсягом коду.

Compose дозволяє краще управляти станом інтерфейсу. Реактивний підхід робить зміну стану більш прозорою та ефективною.

Оскільки Jetpack Compose побудований на Kotlin, це дозволяє розробникам використовувати потужні можливості цієї мови програмування для створення додатків.

Можливість перевикористання компонентів та простота використання дозволяють розробникам створювати та змінювати інтерфейси швидше.

Jetpack Compose підтримується Google і активно розвивається. Це означає, що технологія буде надалі вдосконалюватися та отримувати нові функції.

І хоча традиційні підходи до розробки інтерфейсів мають свої переваги, Jetpack Compose відкриває нові можливості для розробників Android, дозволяючи їм створювати більш зручні, швидкі та сучасні інтерфейси користувача.

2.2.2. Основи програмування з Kotlin у Jetpack Compose

Kotlin, як мова програмування, стала популярним вибором для розробки мобільних додатків на платформі Android. У контексті Jetpack Compose, Kotlin стає ще потужнішим інструментом для створення інтерфейсів користувача. Ось детальний огляд Kotlin у контексті Jetpack Compose:

Kotlin працює інтуїтивно з Jetpack Compose. Компактний та виразний синтаксис Kotlin дозволяє зрозуміло виражати концепції, що лежать в основі створення інтерфейсу.

Kotlin є функціональною мовою програмування, і це відмінно підходить для створення інтерфейсів з Jetpack Compose. Він підтримує високий рівень абстракції, лямбда-вирази, розширення функціональності та інші концепції функціонального програмування, що полегшують роботу з Compose.

Kotlin пропонує безпечність типів, нульову безпеку та багато інших функцій, що допомагають уникати помилок та поліпшують надійність коду, що особливо важливо при розробці великих і складних додатків.

Kotlin спрощує написання меншого обсягу коду, що робить процес розробки більш продуктивним.

Kotlin є офіційною мовою програмування для розробки Android-додатків та має активну підтримку від Google. Це гарантує, що Kotlin стане ще більш удосконаленою для роботи зі створенням інтерфейсів за допомогою Jetpack Compose.

Загалом, Kotlin виступає як потужний інструмент у поєднанні з Jetpack Compose, дозволяючи розробникам створювати сучасні, декларативні та ефективні інтерфейси користувача на платформі Android. Ця комбінація надає простоту, читабельність та ефективність у розробці мобільних додатків.

Знання основ Kotlin є важливим для роботи з Jetpack Compose. Ось огляд основних концепцій мови, які сприяють ефективній роботі з цим інструментом:

Kotlin має систему, яка дозволяє уникати NullPointerException, надаючи безпеку від нульових значень. Оператори `?` та `!!` використовуються для визначення можливості нульового значення змінної.

Kotlin - функціональна мова програмування, що підтримує високорівневі функції, лямбда-вирази, власні функції та інші функціональні концепції. Це робить код більш конкретним та ефективним.

Kotlin дозволяє додавати нові методи до існуючих класів без наслідування від них. Це дозволяє розширювати функціональність вбудованих класів без змінення їх коду.

Концепція Data Class дозволяє створювати класи, які автоматично генерують методи `equals()`, `hashCode()`, `toString()` та інші, спрощуючи роботу з об'єктами даних.

Це легковагі потоки виконання, які дозволяють вирішити проблеми, пов'язані з виконанням багатозадачних операцій асинхронно.

Kotlin підтримує анотації, які дозволяють додавати метадані до коду, що полегшує роботу з певними процесами (наприклад, серіалізація даних).

Kotlin дозволяє створювати локальні функції та замикання, що допомагає організувати та уникати дублювання коду.

Знання цих основних концепцій Kotlin допомагає розробникам з легкістю працювати з Jetpack Compose, оскільки ці концепції утілюють ключові принципи, які використовуються у розробці інтерфейсів у цьому фреймворку.

2.2.3. Компоненти та їх функціонал у Jetpack Compose

Jetpack Compose, як новий фреймворк для розробки інтерфейсів у платформі Android, використовує різноманітні компоненти для створення інтерфейсів користувача. Ось загальний огляд деяких ключових компонентів та їх ролі у розробці:

Text (Текст): Основний компонент для відображення тексту. Це дозволяє відтворювати текст з різними стилями, розмірами та кольорами.

Button (Кнопка): Компонент, який дозволяє створювати інтерактивні кнопки для взаємодії з користувачем.

Row та Column: Контейнери для розміщення компонентів у горизонтальних (Row) та вертикальних (Column) рядках.

Scaffold: Основний контейнерний компонент, який містить елементи, такі як AppBar, BottomNavigation та інші, необхідні для організації загальної структури екрану.

TextField (Текстове поле): Дозволяє користувачам вводити текст або дані через клавіатуру.

Image (Зображення): Використовується для відображення графічних зображень, включаючи ресурси, URL або файли.

Card (Карточка): Компонент для створення блоків контенту зі скругленими кутами та тінями, зазвичай використовується для групування елементів.

Snackbar (Сповіщення): Елемент для відображення коротких повідомлень або сповіщень, які можуть з'являтися знизу екрану.

List (Список): Компонент для створення списків або контейнерів з прокруткою.

Progress Indicator (Індикатор прогресу): Використовується для відображення прогресу завантаження або інших процесів, які вимагають індикації прогресу.

Ці компоненти утворюють основу для створення різноманітних інтерфейсів у Jetpack Compose. Комбінуючи їх різними способами та налаштовуючи їх параметри, розробники можуть створювати різноманітні та інтуїтивно зрозумілі інтерфейси для своїх додатків.

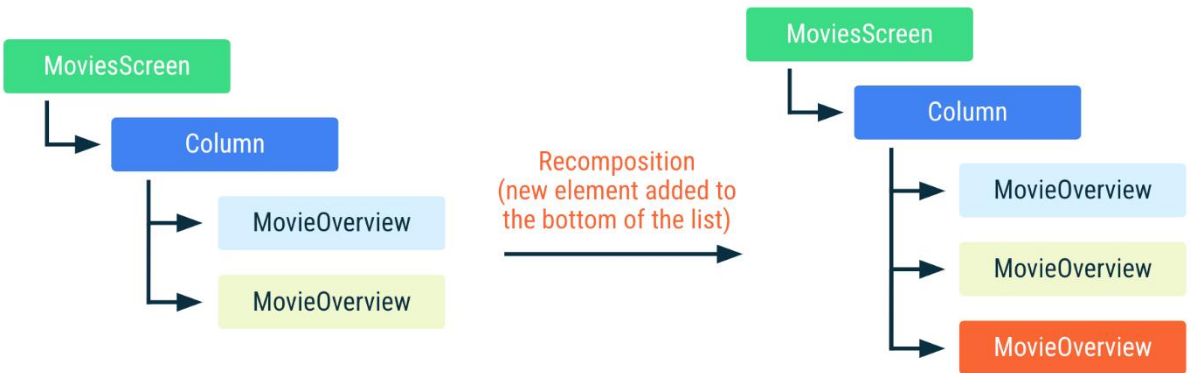


Рис 2.4. Репрезентація ієрархії компонентів Jetpack Compose

Основні компоненти в Jetpack Compose надають зручний спосіб створення інтерфейсів у мобільних додатках на платформі Android. Ось детальний огляд деяких типових компонентів:

Button (Кнопка): Button є основним компонентом для створення інтерактивних кнопок. Він дозволяє налаштовувати текст, оброблювати натискання та застосовувати різноманітні стилі.

```kotlin
Button(onClick = { /* Дії при натисканні */ }) {
    Text(text = "Натисни мене")
}
```

Text (Текст): Text використовується для відображення тексту з різними стилями, такими як розмір шрифту, колір та тип шрифту.

```kotlin
Text(text = "Привіт, світ!", fontSize = 20.sp, color = Color.Black)
```

TextField (Текстове поле): TextField дає можливість користувачам вводити текст або дані через клавіатуру.

```kotlin
var textValue by remember { mutableStateOf(TextFieldValue()) }
TextField(value = textValue, onValueChange = { textValue = it })
```

List (Список): Цей компонент створює прокручуваний список елементів.

```kotlin
val items = listOf("Пункт 1", "Пункт 2", "Пункт 3")
LazyColumn {
    items(items) { item ->
        Text(text = item)
    }
}
```

Row та Column: Використовуються для розміщення компонентів у горизонтальних та вертикальних рядках відповідно.

```kotlin
Row {
    Text(text = "Елемент 1")
    Text(text = "Елемент 2")
}

Column {
    Text(text = "Елемент 1")
    Text(text = "Елемент 2")
}
```

Card (Карточка): Card створює блок контенту зі скругленими кутами та тінями.

```kotlin
Card(
    shape = RoundedCornerShape(8.dp),
    elevation = 4.dp,
    modifier = Modifier.padding(16.dp)
) {
    Text(text = "Це карточка")
}
```

Ці компоненти є основою для побудови різних інтерфейсів у Jetpack Compose. Шляхом поєднання різних компонентів та налаштування їх параметрів можна створювати різноманітні та інтуїтивно зрозумілі інтерфейси для Android додатків.

2.2.4. Анімації та інтерактивність у Jetpack Compose

Jetpack Compose, інструмент для розробки інтерфейсів у мобільних додатках Android, пропонує потужні можливості для створення анімацій. Ось огляд основних понять та можливостей анімацій в Jetpack Compose:

Animatable це клас, який використовується для анімації значень. Він забезпечує методи для зміни числових значень з плинними анімаційними ефектами.

Transition - це ключовий елемент анімацій в Jetpack Compose, який дозволяє виконувати анімацію при зміні стану компонента або екрану.

AnimationSpec: Це об'єкт, який визначає параметри анімації, такі як тривалість, ефект затухання та інші характеристики анімації.

Jetpack Compose дозволяє створювати анімації для зміни різних параметрів компонентів, таких як розмір, позиція, прозорість тощо.

Transition API дозволяє створювати перехідні анімації між різними станами елементів UI.

Складні анімації: Jetpack Compose дає можливість створювати складні анімації, такі як паралельні анімації, послідовні ефекти тощо.

````
```kotlin
var sizeState by remember { mutableStateOf(200.dp) }

Box(
    modifier = Modifier
        .size(sizeState)
        .background(Color.Green)
        .clickable {
            sizeState = if (sizeState == 200.dp) 300.dp else 200.dp
        },
    contentAlignment = Alignment.Center
) {
    Text(text = "Натисни мене", color = Color.White)
}
```
````

Цей приклад змінює розмір квадрата при кожному натисканні, створюючи ефект зміни розміру з анімаційним згладжуванням.

Jetpack Compose надає величезні можливості для створення різних видів анімацій, дозволяючи розробникам створювати захоплюючі та динамічні інтерфейси для Android додатків.

Анімації стали ключовою частиною сучасних інтерфейсів, надаючи їм динамічності та привабливості. Основи роботи з анімаціями включають:

Змінні стани: Це базовий елемент анімації. Анімація відбувається через плавний перехід між двома чи більше станами об'єкта.

Часові проміжки (Timelines): Визначають тривалість та темп анімації. Вони контролюють, як швидко чи повільно відбувається зміна від початкового до кінцевого стану.

Інтерполяція: Це спосіб, яким значення між початковим і кінцевим станом розподіляється впродовж анімації. Наприклад, лінійна анімація робить рівномірний перехід, тоді як згладжена анімація може починати повільно та прискорюватись пізніше.

Функції затухання (Easing functions): Визначають поведінку анімації під час її виконання. Наприклад, затухання "ease-in" робить початок анімації більш плавним, а "ease-out" - кінець анімації.

Планування: Визначення, яка частина інтерфейсу має бути анімованою та які зміни відбудуться.

Реалізація: Використання відповідних інструментів чи бібліотек для створення анімації, враховуючи параметри часу, значень та інтерполяції.

Тестування та налагодження: Перевірка анімацій на різних пристроях та у різних умовах для забезпечення коректності та плавності.

Оптимізація: Зменшення використання ресурсів, таких як CPU чи пам'ять, для забезпечення оптимальної продуктивності анімацій.

Робота з анімаціями вимагає ретельного планування та експериментів для досягнення бажаних результатів. Вони можуть значно підвищити користувацький досвід та привернути увагу до вашого продукту чи додатку.

2.2.5. Інструменти й середовище розробки для Jetpack Compose

Android Studio є одним з основних інтегрованих середовищ розробки (IDE) для створення Android-додатків. Останнім часом в ньому з'явилася підтримка Jetpack Compose, що значно спрощує роботу з цим сучасним фреймворком для створення інтерфейсів.

Android Studio надає зручний редактор для написання коду Compose, з підсвічуванням синтаксису, автодоповненням та візуальним переглядом компонентів.

Це дозволяє переглядати прев'ю створюваних інтерфейсів без запуску додатку на справжньому пристрої, що значно прискорює процес розробки.

Вбудований дебагер дозволяє відлагоджувати Compose-код, шукаючи помилки та відстежуючи процес роботи інтерфейсу.

Android Studio пропонує інтегровані засоби для розробки та налагодження Android-додатків, такі як AVD Manager, Logcat та інші, що дозволяє легко управляти Compose-проектами.

Android Studio постійно оновлюється для підтримки нових функцій та покращень у Compose, що робить його одним з найзручніших середовищ для розробки.

Android Studio разом з підтримкою Jetpack Compose надає розробникам зручний і потужний інструментарій для створення сучасних та привабливих інтерфейсів у додатках для Android.

2.2.6. Архітектурні підходи та прийоми роботи з Jetpack Compose

Архітектурні підходи в розробці Android додатків грають ключову роль у створенні простих, структурованих та легко розширюваних програм. Ось огляд кількох основних паттернів:

MVC (Model-View-Controller). Model: Представлення даних та бізнес-логіки. View: Інтерфейс користувача. Controller: Керує взаємодією між Model і View.

MVP (Model-View-Presenter). Model: Інформація та бізнес-логіка. : Інтерфейс користувача. Presenter**: Містить бізнес-логіку, реагує на взаємодію View.

MVVM (Model-View-ViewModel). Model: Дані та логіка додатку. View: UI компоненти. ViewModel: Підготовляє дані для View, не залежить від конкретного View.

Clean Architecture. Entities: Це сутності додатку. Use Cases: Бізнес-логіка. Interface Adapters: Приводять дані в формат, що розуміє Use Cases. Frameworks & Drivers: Зовнішні компоненти та фреймворки.

Inversion of Control (IoC): Управління залежностями між класами. Dependency Injection (DI): Вставка залежностей в об'єкти.

ViewModel: Зберігає та управляє даними для UI, виживає при зміні конфігурації. LiveData: Автоматично оновлює UI при зміні даних в джерелі. Room: Об'єктно-орієнтована бібліотека для роботи з базами даних SQLite.

Кожен з цих підходів має свої переваги та недоліки. Вибір певного залежить від потреб вашого проекту та командних уподобань. Важливо використовувати архітектурний підхід, який дозволяє створювати проекти, які легко розширювати, тестувати та підтримувати.

2.3. Висновки до другого розділу

У другому розділі роботи було розглянуто синтаксис та роботу нативних фреймворків SwiftUI та Jetpack Compose, які є офіційними засобами для розробки мобільних додатків для платформ iOS та Android відповідно. На основі аналізу документації, прикладів коду та літератури було зроблено наступні висновки:

SwiftUI та Jetpack Compose є сучасними фреймворками, які використовують принципи декларативного програмування для створення інтерфейсів користувача. Вони дозволяють розробникам описувати, що вони хочуть відобразити на екрані, а не як це робити. Вони також надають можливість гарячого перезавантаження та перезапуску коду, що сприяє швидкій та ітеративній розробці.

SwiftUI та Jetpack Compose мають багато спільного у своєму синтаксисі та роботі. Вони обидва використовують функції, які повертають ком-

поненти інтерфейсу, які називаються віджетами в Jetpack Compose та в'юшками в SwiftUI. Вони також використовують модифікатори для зміни властивостей віджетів та в'юшок, а також стани та обсервери для відстеження змін даних та оновлення інтерфейсу.

SwiftUI та Jetpack Compose мають також свої відмінності, які пов'язані з особливостями мов програмування Swift та Kotlin, а також з різницями між платформами iOS та Android. Наприклад, SwiftUI використовує протоколи та структури для визначення в'юшок, а Jetpack Compose використовує анотації та класи. SwiftUI також використовує біндінги та об'єкти референсного типу для зберігання стану, а Jetpack Compose використовує делегати та об'єкти значення. Крім того, SwiftUI та Jetpack Compose мають різні набори віджетів та в'юшок, які відповідають дизайн-системам Apple та Google.

Таким чином, SwiftUI та Jetpack Compose є перспективними фреймворками, які надають розробникам можливість створювати нативні інтерфейси користувача для платформ iOS та Android за допомогою декларативного підходу. Вони мають багато спільного у своєму синтаксисі та роботі, але також мають свої відмінності, які потрібно враховувати при розробці. У наступних розділах роботи буде проведено практичне дослідження продуктивності фреймворків SwiftUI та Jetpack Compose у порівнянні з фреймворком Flutter.

РОЗДІЛ 3
Порівняння роботи фреймворку Flutter з нативними фреймворками мобільної розробки.

3.1. Продуктивність та швидкодія Flutter в порівнянні з нативними фреймворками

Розробка мобільних додатків вимагає уваги до продуктивності, яка включає в себе швидкодію, ефективність розробки та відповідність вимогам користувача. Поглянемо на продуктивність у контексті порівняння Flutter з нативними фреймворками.

Ефективність розробки:

- Швидкість розробки: Використання готових компонентів, hot reload (гаряча перезавантаження) у Flutter дозволяє швидко відображати зміни у реальному часі.

- Підтримка різних платформ: Однаковий код для iOS та Android, що полегшує розробку та підтримку обох платформ.

Швидкодія додатків:

- Flutter: Досягає високої швидкодії завдяки власному рендерингу і швидкому обміну з платформами.

- Нативні фреймворки: Вони, зазвичай, надають кращу оптимізацію для платформи, але можуть вимагати більше часу для розробки на обидві платформи.

Швидкодія:

- Flutter: Зазвичай надає стабільну швидкодію завдяки власному рендерингу.

Нативні фреймворки: Можуть мати перевагу в швидкості завдяки прямому доступу до функцій та оптимізації для конкретної платформи.

Ресурсомісткість:

- Flutter: Завдяки власному движку може вимагати більше ресурсів на обробку.

Нативні фреймворки: Можуть оптимізуватись під кожну платформу, що дозволяє краще управляти ресурсами.

Обидва підходи мають свої переваги та обмеження. Flutter надає швидкість розробки та високу швидкодію, але може бути менш ресурсоємним. Нативні фреймворки можуть бути оптимізовані для конкретної платформи, проте вимагають більше часу для розробки. Вибір між ними залежить від потреб вашого проекту, вмінь команди та очікувань щодо продуктивності та швидкодії.

Звичайно, Flutter має свої переваги та деякі обмеження, особливо щодо продуктивності. Оцінимо ці аспекти більш детально.

Гаряче перезавантаження (Hot Reload): Дозволяє бачити зміни в реальному часі без перезапуску додатку.

Однаковий код для різних платформ: Можливість використовувати один і той же код для iOS та Android.

Власний движок рендерингу: Створений для оптимізації швидкодії додатку.

Відсутність мостів до нативних елементів: Оскільки Flutter самостійно малює кожен піксель на екрані, це дозволяє досягати високої швидкодії.

Ресурси пам'яті та обробки: Використання власного рендерингу може призвести до більшого споживання пам'яті і потужності процесора, особливо при відображенні складних інтерфейсів.

Можливість більшої ресурсомісткості на певних платформах: Хоча Flutter намагається бути універсальним, іноді він може не бути оптимізований на всіх пристроях рівномірно.

Flutter - потужний інструмент, що надає швидку розробку та високу швидкодію, але при цьому може вимагати більше ресурсів, особливо при

відображенні складних інтерфейсів. При проектуванні додатків важливо розуміти ці аспекти та враховувати їх при виборі між Flutter та іншими рішеннями для розробки мобільних додатків.

Тестування продуктивності є важливою складовою розробки додатків для забезпечення їхньої ефективності та оптимізації швидкодії. Методології тестування продуктивності допомагають порівняти ефективність програм, написаних на різних мовах програмування або використовуючи різні фреймворки.

Вимірювання часу відгуку на взаємодію з елементами UI. Аналіз часу, необхідного для завантаження різних сторінок у додатку. Використання інструментів, таких як Xcode Instruments для Swift і Profiler для Kotlin, для визначення областей коду, які витрачають найбільше часу. Тестування конкретних функцій або методів для визначення їхньої швидкодії.

Ідентифікація частин коду, які впливають на швидкодію додатку. Оптимізація або заміна швидкодійних алгоритмів для поліпшення продуктивності.

Тестування продуктивності (Benchmarking) - важлива складова розробки додатків у Flutter, Swift та Kotlin. Воно допомагає ідентифікувати області для оптимізації та поліпшення продуктивності коду, що дозволяє створювати ефективні, швидкодійні додатки.

3.2. Відмінності у розробці, тестуванні та розгортанні мобільних додатків між Flutter та нативними фреймворками

Процес розробки додатків у Flutter та нативних фреймворках, таких як Swift для iOS і Kotlin для Android, включає різні етапи, кожен з яких має свої особливості.

Плюси Flutter це:

Один код для обох платформ - можна писати один код для iOS та Android, що спрощує розробку.

Гаряче перезавантаження (Hot Reload) - миттєва перевірка змін у реальному часі без перезапуску додатку.

Велика спільнота та швидкий розвиток - активна спільнота розробників та розвиваюча платформа.

Нативні можливості платформи це доступ до всіх можливостей платформи з їхніми унікальними функціями. Оптимізація під конкретну платформу: Можливість оптимізувати додаток під конкретну платформу для отримання максимальної продуктивності.

Схожості: Незалежно від платформи, розробка інтерфейсу вимагає використання специфічних засобів, таких як SwiftUI для iOS та Compose для Android. Тестування є важливою частиною процесу розробки, незалежно від вибраної платформи.

Вибір платформи залежить від кількох факторів. Flutter може прискорити розробку завдяки гарячому перезавантаженню та єдинообразному коду. Якщо додаток потребує специфічного доступу до функцій платформи, нативний підхід може бути кращим вибором.

Кожен з підходів має свої переваги та обмеження. Важливо розглянути потреби проєкту та можливості платформи при виборі між Flutter та нативними фреймворками.

Звісно, тестування мобільних додатків – це важлива складова розробки, особливо в умовах постійних змін технологій та швидкого розвитку ринку мобільних додатків. При розгляді методів тестування у контексті Flutter та нативних мов для мобільних додатків, можна виявити певні відмінності та переваги кожної з цих платформ.

Почнемо з загального уявлення про тестування та процеси QA (Quality Assurance). QA – це систематичний підхід до забезпечення якості

продукту. Це включає в себе не лише тестування коду, а й аналіз вимог користувачів, валідацію функціональності та дослідження користувацького досвіду. Тестування ж – це процес перевірки програмного забезпечення на відповідність вимогам та виявлення помилок для їх подальшого виправлення.

У контексті мобільних додатків на Flutter і нативних мовах (таких як Kotlin для Android або Swift для iOS) існують відмінності у підходах до тестування. Flutter, який пропонує кросплатформеність, має вбудовану систему тестування - Flutter Testing Framework. Це дозволяє тестувати як UI (інтерфейс користувача), так і бізнес-логіку. Один тест може бути запущений на різних платформах, що спрощує розробку для декількох ОС. Завдяки цьому, розробники можуть швидше створювати та тестувати функціональність додатку.

У нативних мовах тестування також відбувається на рівні бізнес-логіки та UI. Проте, для кожної платформи потрібно писати окремі тести, що може призвести до збільшення часу на розробку та тестування. Однак це також дає більшу гнучкість у роботі з особливостями кожної платформи, що може бути критичним у випадку, коли потрібно максимально використовувати можливості певної ОС.

Отже, вибір методу тестування для мобільного додатку на Flutter чи нативних мовах може залежати від конкретних потреб проекту. Flutter спрощує кросплатформену розробку та тестування, проте, нативні мови можуть надати більшу гнучкість та точність у роботі з конкретною платформою. В кінцевому підсумку, вибір методу тестування повинен враховувати потреби проекту, обсяг ресурсів та бажану оптимізацію розробки.

Коли йдеться про розробку мобільних додатків, швидкість розробки та використання ресурсів грають важливу роль. Порівнюючи Flutter і нативні мови розробки, можна виявити певні відмінності у використанні ресурсів та швидкості розробки.

Flutter - це фреймворк для кросплатформеного розроблення, що використовує Dart. Він відомий своєю швидкістю розробки завдяки гарнітурі вбудованих компонентів та можливості гарячої перезавантаження (hot reload). Гаряче перезавантаження дозволяє швидко бачити зміни у реальному часі без перезапуску додатку, що прискорює процес розробки.

З іншого боку, нативні мови (такі як Kotlin для Android або Swift для iOS) надають більш прямий доступ до можливостей конкретної платформи. Це може забезпечити оптимізацію під конкретний пристрій, що в свою чергу може позитивно вплинути на швидкість роботи додатку. Однак, розробка на нативних мовах може вимагати більше часу через потребу писати окремий код для кожної платформи.

Щодо ресурсоємності, Flutter має свій власний набір віджетів та компонентів, які перетворюються у нативний код для кожної платформи. Це може призвести до додаткового використання ресурсів пам'яті та обробки, оскільки інтерпретатор Dart працює поверх двигуна двоїстої виконавчої системи.

У нативних мовах розробки, ресурсоємність може бути меншою завдяки тому, що код оптимізується безпосередньо під певну платформу. Використання лише необхідних компонентів допомагає уникнути зайвого використання ресурсів.

Отже, при виборі між Flutter та нативними мовами розробки слід враховувати потреби проекту. Flutter забезпечує швидкість розробки та кросплатформеність, але може мати більшу ресурсоємність. Нативні мови можуть бути більш ефективними у використанні ресурсів, але вимагатимуть більше часу на розробку для кожної платформи.

Після релізу додатку, процес розгортання та підтримки грає важливу роль у забезпеченні його успішності на ринку. Рівень підтримки та відмінності у процесі розгортання можуть варіюватися в залежності від платформи розробки та вибраного фреймворку.

Почнемо з розгортання на нативних платформах. При розробці за допомогою нативних мов (наприклад, Kotlin для Android або Swift для iOS), процес розгортання може вимагати додаткового часу та зусиль. Кожна платформа має свої власні вимоги до підготовки додатку для публікації в магазинах додатків (Google Play Store для Android або App Store для iOS). Це включає створення підписів, відповідність політиці магазину, тестування на різних пристроях тощо. Після публікації, підтримка включає в себе виправлення помилок, випуск патчів та оновлень для підтримки сумісності з новими версіями ОС.

У випадку з Flutter, процес розгортання може бути спрощеним через кросплатформеність. Однак, для публікації в магазинах додатків також потрібно відповідати їхнім вимогам. Flutter також має власні інструменти для автоматизації певних аспектів розгортання, таких як створення апк (APK) для Android або IPA-файлів для iOS.

Щодо рівня підтримки, важливо забезпечити постійне оновлення додатку з урахуванням змін у вимогах платформи, виправлення помилок та відповідь на звернення користувачів. Націленість на високу якість підтримки може позитивно позначитися на репутації додатку та відношенні користувачів до нього.

Отже, незалежно від платформи розробки, розгортання та підтримка додатку після релізу вимагають уваги до деталей, відповідності вимогам магазинів, вирішення проблем та швидкого реагування на зміни у вимогах платформи чи відгуки користувачів. Це важливий етап, що визначає успіх та прийняття додатку на ринку.

3.3. Огляд недоліків та можливих обмежень у використанні Flutter в порівнянні з нативними фреймворками

Розробка за допомогою Flutter та нативних фреймворків має свої відмінності у складності та труднощах, з якими можуть зіткнутися розробники.

Flutter, як кросплатформений фреймворк, має свої переваги у спрощенні процесу розробки. Одна з головних переваг - це гаряче перезавантаження (hot reload), що дозволяє розробникам швидко бачити зміни у реальному часі, без необхідності повторного компілювання всього додатку. Це значно прискорює процес виправлення помилок та тестування нових функцій.

Однак, у розробці на Flutter можуть виникати труднощі, пов'язані з кросплатформеною природою фреймворку. Незважаючи на те, що Flutter намагається максимально абстрагувати розробників від платформи, можуть виникати проблеми з адаптацією до конкретних особливостей певної платформи. Деякі функції або дизайн можуть вимагати додаткових налаштувань для кожної платформи, що може призвести до додаткової складності.

У випадку нативних фреймворків (таких як Kotlin для Android або Swift для iOS), розробники мають прямий доступ до можливостей конкретної платформи. Це може бути перевагою у випадку, коли потрібно максимально використовувати можливості платформи або оптимізувати продукт під певний пристрій. Однак, це також означає писати окремий код для кожної платформи, що може збільшити час розробки та потребує більшої експертизи в конкретних мовах програмування.

Таким чином, вибір між Flutter та нативними фреймворками може залежати від конкретних потреб проекту. Flutter пропонує швидкість розробки та кросплатформеність, але може потребувати додаткових налаштувань для адаптації до конкретних платформ. Нативні фреймворки можуть

надати більшу гнучкість у роботі з особливостями платформи, але вимагатимуть більше часу на розробку та експертизу у відповідних мовах програмування.

Підтримка та можливості розширення функціоналу - важливі аспекти при виборі між Flutter та нативними фреймворками для розробки мобільних додатків.

Flutter, як кросплатформений фреймворк, має певні переваги у плані розширюваності функціоналу. Однією з основних переваг є гнучкість у створенні власних віджетів та компонентів, які можуть бути використані у різних частинах додатку або навіть у різних проектах. Flutter також має широкий спектр готових пакетів (packages) та бібліотек, які допомагають швидко розширювати функціонал додатку. Крім того, його гаряче перезавантаження (hot reload) сприяє швидкому впровадженню нового функціоналу та його швидкому тестуванню без повного перекомпілювання.

Нативні фреймворки (наприклад, Kotlin для Android або Swift для iOS) також мають свої можливості розширення та підтримки. Вони надають прямий доступ до особливостей конкретної платформи, що може бути корисним у випадках, коли потрібно максимально використовувати можливості певної ОС. Більш тонке налаштування та оптимізація для конкретної платформи також можуть бути легше реалізовані через нативні фреймворки.

Проте, при підтримці та розширенні функціоналу на нативних фреймворках може виникати ситуація, коли деякі зміни або розширення вимагають реалізації окремо для кожної платформи, що може забирати більше часу та ресурсів.

Отже, обидва підходи мають свої переваги та обмеження у плані підтримки та розширення функціоналу. Flutter дозволяє розробникам швидко розширювати функціонал та має гнучкість у використанні власних компонентів, тоді як нативні фреймворки надають більш прямий доступ до

можливостей конкретної платформи, але можуть потребувати більше зусиль для реалізації розширень на кожній з них. Обираючи між ними, варто враховувати специфіку проекту та потреби у розширенні функціоналу з плином часу.

Апаратні можливості платформи мають значний вплив на виконання мобільних додатків як на Flutter, так і на нативних фреймворках. Різниця в апаратних можливостях між різними пристроями та платформами може впливати на продуктивність, швидкість та загальний досвід використання додатків.

Flutter, як кросплатформений фреймворк, має свої особливості у взаємодії з апаратними можливостями платформ. Однією з переваг Flutter є можливість перенесення одного й того ж коду на різні платформи, що спрощує розробку. Однак, різниця у апаратних характеристиках між платформами може впливати на продуктивність додатків. Наприклад, різниця у швидкості процесорів, обсягу оперативної пам'яті чи особливості GPU може призвести до різних результатів роботи додатку на різних пристроях.

У випадку нативних фреймворків, розробники можуть більш прямо взаємодіяти з апаратними можливостями конкретної платформи. Це може дати більшу гнучкість у використанні конкретних можливостей пристрою, таких як датчики, камери, прискорювачі тощо. Такий доступ до апаратних можливостей може дозволити оптимізувати додаток під конкретний пристрій, але такий підхід може вимагати більше часу та експертизи.

Важливо враховувати, що різниця у апаратних можливостях може вплинути на продуктивність, швидкість та ресурсомісткість додатків. Наприклад, додаток, який працює оптимально на одному пристрої, може мати проблеми на пристроях з меншими апаратними характеристиками.

Таким чином, під час розробки додатків на Flutter чи нативних фреймворках важливо уважно враховувати апаратні можливості платформи

та оптимізувати додаток під різні типи пристроїв для забезпечення оптимального виконання та відмінного досвіду користувача на різних пристроях.

Оптимізація використання ресурсів пам'яті - важливий аспект при розробці додатків на обох платформах, який впливає на продуктивність, ефективність роботи додатків і задоволення користувачів.

У контексті Flutter, оптимізація пам'яті включає у себе кілька аспектів. Завдяки кросплатформеності, додатки на Flutter мають можливість використовувати один і той же код для різних платформ, але це також може призводити до зайвого використання пам'яті. Наприклад, віджети, що рендеряться для обох платформ, можуть використовувати більше ресурсів, ніж нативні аналоги. Тут важливо враховувати оптимізацію роботи з пам'яттю, використання кешування та уникнення зайвих операцій з об'єктами.

У нативних фреймворках, таких як Kotlin для Android або Swift для iOS, розробники мають більш прямий доступ до апаратних можливостей та можуть більш точно керувати використанням пам'яті. Вони можуть оптимізувати роботу з пам'яттю для конкретних пристроїв та операційних систем, що дозволяє досягти більшої ефективності використання ресурсів.

У обох випадках важливо використовувати кращі практики управління пам'яттю: уникати утечок пам'яті, оптимізувати завантаження та вивільнення ресурсів, використовувати кешування та локальне зберігання даних. Додатково, моніторинг та аналіз використання пам'яті під час розробки і тестування допомагають виявити можливі проблеми та вдосконалити ефективність додатку.

Оптимізація використання ресурсів пам'яті є важливою для покращення продуктивності та ефективності роботи додатків на обох платформах. Забезпечення оптимального використання пам'яті допомагає забезпечити швидке та стабільне функціонування додатку для користувачів.

Використання Flutter в порівнянні з нативними фреймворками має свої переваги, але також включає деякі недоліки та обмеження, які варто враховувати при розробці мобільних додатків.

Недоліки та обмеження Flutter у порівнянні з нативними фреймворками:

Доступ до нативних функцій: Однією з основних проблем Flutter є обмежений доступ до специфічних для платформи API. Це може ускладнювати реалізацію деяких функцій, які вимагають прямого доступу до функціоналу платформи.

Ресурсомісткість: В порівнянні з деякими нативними додатками, додатки на Flutter можуть бути більш ресурсоємними, що може впливати на продуктивність та витрати енергії пристрою.

Оптимізація для конкретних платформ: У деяких випадках потрібно вкладати додатковий час та зусилля в оптимізацію додатку для конкретних платформ, що може зменшувати переваги кросплатформеності.

Аналіз потреб проекту: Розглядайте використання Flutter у контексті конкретних вимог проекту. Якщо потрібна висока кросплатформеність та швидкість розробки, Flutter може бути вигідним вибором.

Уважне врахування особливостей платформи: При плануванні варто розглянути, чи необхідний прямий доступ до специфічних можливостей платформи для додатку. У такому випадку, нативний фреймворк може бути більш ефективним.

Тестування та оптимізація продуктивності: Незалежно від обраного фреймворку, важливо проводити тестування та оптимізацію продуктивності, щоб забезпечити оптимальний досвід користувача.

У кінцевому підсумку, вибір між Flutter та нативними фреймворками залежить від конкретних потреб проекту та експертизи розробника. Варто уважно враховувати переваги та недоліки кожного фреймворку для забезпечення успішного розвитку та впровадження мобільного додатку.

3.4. Розробка додатків на фреймворках Flutter, SwiftUI, Jetpack Compose та практичне порівняння продуктивності.

3.4.1. Практичний етап розробки

У наступному розділі ми дослідимо процес створення мобільного додатку, який базується на використанні трьох сучасних фреймворків: Flutter, SwiftUI та Jetpack Compose. Ми проаналізуємо реалізацію простого, але ефективного додатку, спрямованого на отримання та відображення значної кількості об'єктів користувачів. Цей розділ буде включати огляд ключових концепцій та можливостей кожного з наведених фреймворків, дозволяючи вам глибше ознайомитися з їхнім функціоналом та застосувати ці знання на практиці для створення потужних мобільних додатків. Продовжимо крок за кроком, розглядаючи цей захоплюючий аспект мобільної розробки та використовуючи інструменти цих фреймворків для досягнення поставлених цілей.

Цей додаток на Flutter слугує не лише простим прикладом, але й візуальним демонстрантом ключового функціоналу, який часто використовується у комерційних мобільних додатках. Він відтворює такі важливі аспекти, як робота з API для отримання даних, асинхронні операції та анімована навігація між різними сторінками додатку.

Використання API для отримання даних відображає сучасний підхід до розробки, де мобільні застосунки взаємодіють з віддаленими серверами, отримуючи оновлення та необхідну інформацію для користувачів в реальному часі. Цей додаток підкреслює, як легко та ефективно здійснювати таку взаємодію з використанням Flutter.

Також він пропонує приклад асинхронних операцій, які є невід'ємною частиною сучасних додатків. Завдяки цьому функціоналу додаток може одночасно виконувати кілька завдань, таких як завантаження даних з сервера чи обробка інформації, не блокуючи основний інтерфейс користувача.

Нарешті, анімована навігація між екранами додатку додає візуальну привабливість та комфорт користування. Це підкреслює важливість не лише функціональності, але й користувацького досвіду, який забезпечується різноманітними анімаційними ефектами при переході між різними частинами додатку.

Усі ці аспекти у поєднанні дозволяють цьому додатку відтворювати та відобразити ключові елементи функціональності, які є основою багатьох успішних комерційних мобільних додатків.

У нашому дослідженні ми розпочнемо з розгляду створення мобільного додатку з використанням Flutter - одного з провідних фреймворків для розробки зручних та відмінних застосунків для мобільних пристроїв. На шляху розробки ми обрали один з найпопулярніших підходів до структурування додатків - паттерн "блок".

Паттерн Bloc (Business Logic Component) визначає структуру додатку, розділяючи його на чотири основні складові: Блоки (Blocks), Події (Events), Стани (States) та Відображення (UI). Цей підхід сприяє відокремленню бізнес-логіки від візуальної частини додатку, спрощуючи тестування та підтримку коду.

Обравши Bloc для створення нашого додатку на Flutter, ми вибрали потужний та добре адаптований метод, який дозволить нам ефективно керувати даними та логікою додатку, забезпечуючи його масштабованість та стабільність у подальшому розвитку.

Цей проект та усі наступні будуть дотримуватися спільної структури. Але звичайно у кожному проекті та фреймворку є свої особливості якими вони будуть вирізнятися.

Це структура Flutter проекту:

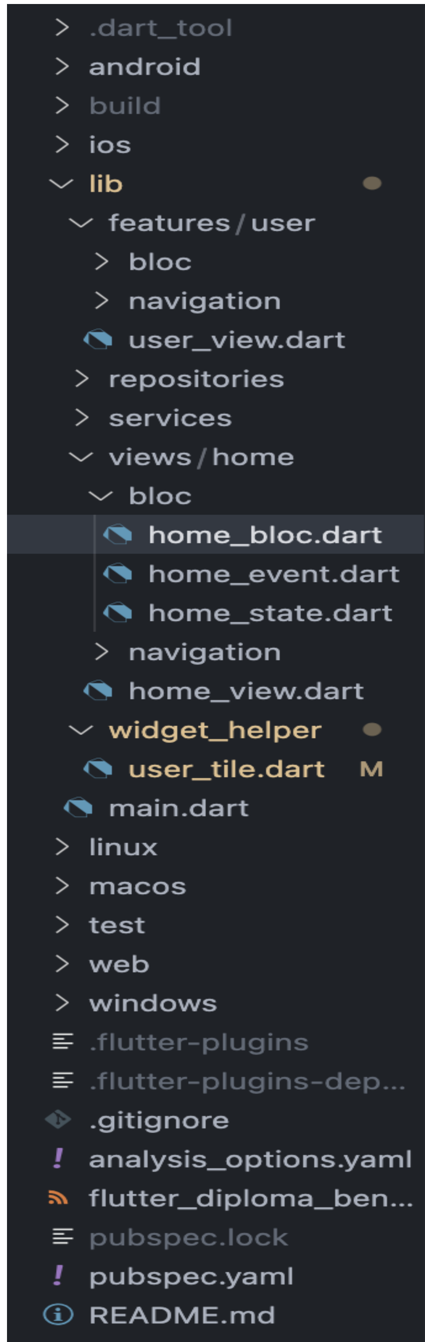


Рис 3.1. Структура створеного Flutter проекту

Цей проект повністю відповідає усім сучасним стандартам написання додатку. Кожному View, тобто сторінці додатку, належить свій Bloc,

який відповідає за логіку, зміну State та виклик Event. Також у кожного View є свої об'єкти навігації які відповідають за перехід на інший View. Також є ще один шар абстракції, це Repository/Services, який у бібліотеці блок реалізован з допомогою віджету MultiRepositoryProvider. Усе це дозволяє нам масштабувати, планувати та реалізовувати системи будь-якої складності.

Розподіл папок у цьому проекті відображає структурований підхід до розробки, спрямований на підтримку чистоти та організації коду. Ось детальний опис структури:

Features: Ця папка містить реалізацію функціональності, яка може включати логіку, що охоплює декілька візуальних компонентів (views). Це може бути, наприклад, введення даних користувачем та їх подальше використання у декількох частинах додатку.

Repositories: Тут зосереджені репозиторії, сервіси та моделі даних, які використовуються для взаємодії з віддаленим або локальним джерелом даних (наприклад, API). Моделі даних відображають об'єкти, що передаються між додатком та джерелом інформації, сервіси виконують взаємодію з API, а репозиторії управляють доступом до цих даних.

Services: Ця папка містить інші сервіси, які використовуються у додатку. Це можуть бути різні допоміжні сервіси, які виконують певні завдання відокремлено від основної бізнес-логіки.

Views: Тут зберігаються основні великі візуальні компоненти, які представляють різні екрани чи частини додатку. Це може бути головний екран, екран профілю користувача, екран налаштувань тощо.

Widget Helper: В цій папці містяться допоміжні віджети, які можуть бути використані у різних частинах додатку. Це дозволяє уникнути дублювання коду та спрощує процес розробки, оскільки ці віджети можуть бути легко використані у багатьох місцях.

Ця структура допомагає підтримувати організований та легко зрозумілий код, дозволяє швидше знаходити необхідні компоненти та сприяє більшій чистоті та модульності проекту.

Після запуску проекту на пристрої iPhone отримуємо наступні екрани.

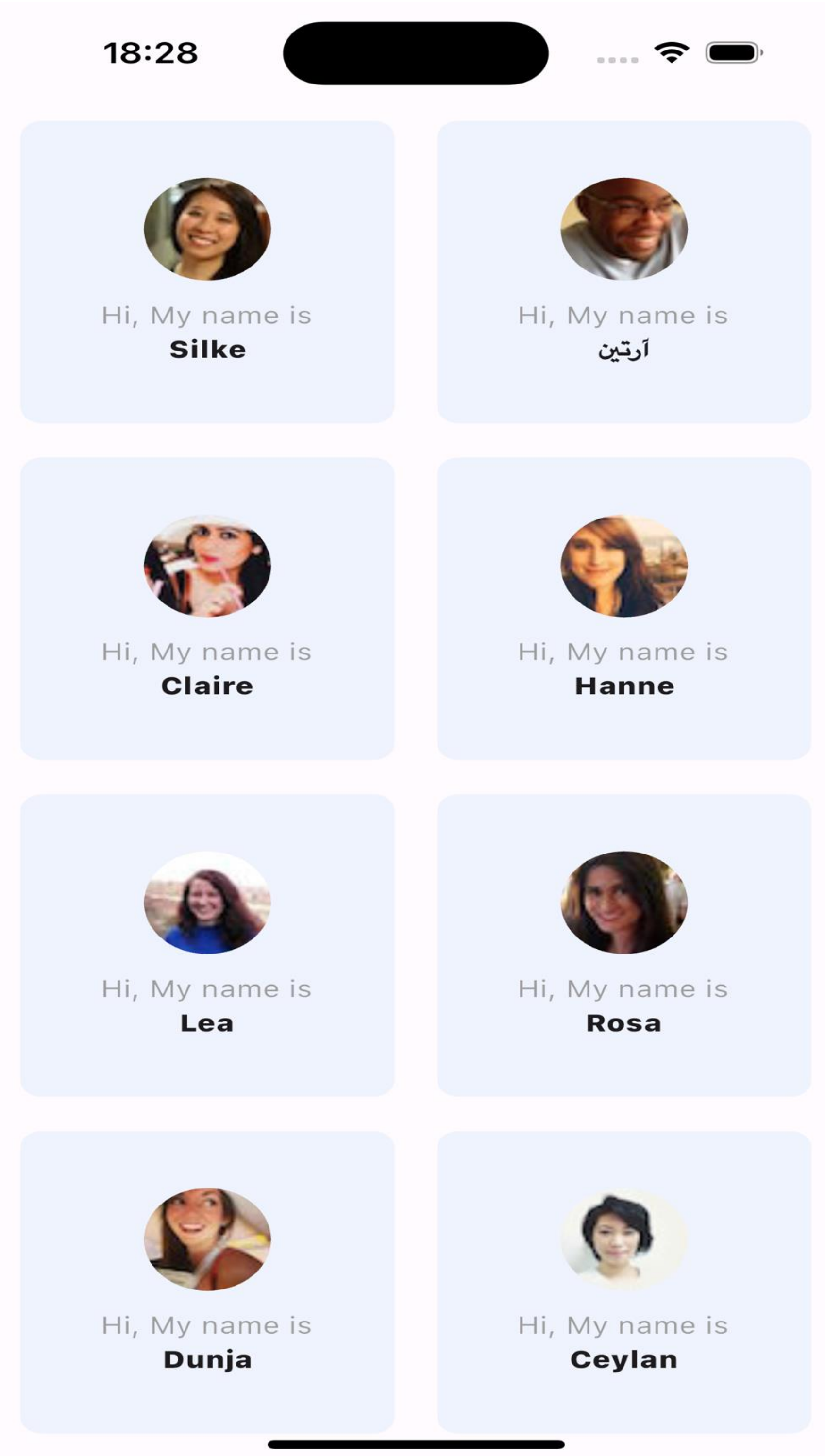
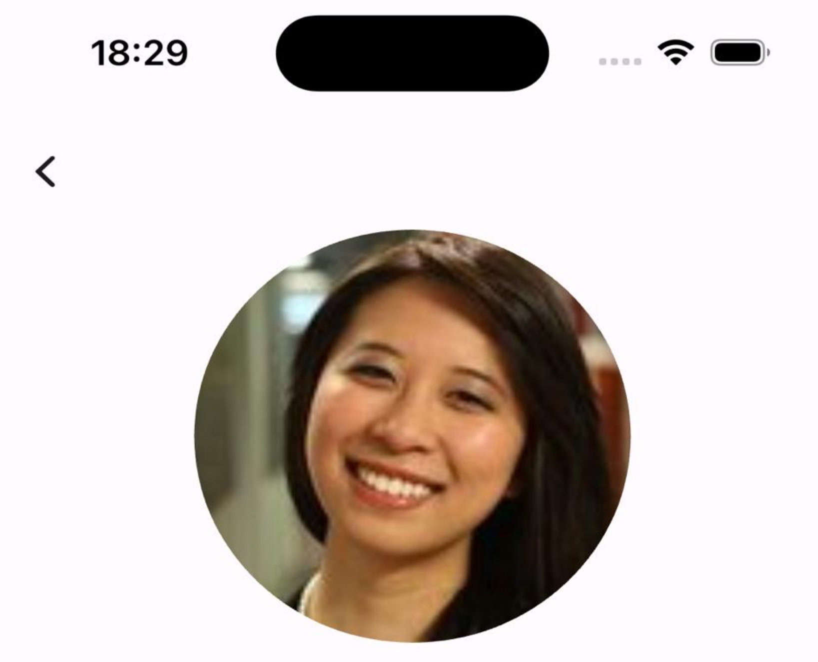


Рис 3.2. Екран Home створеного Flutter проекту



First Name: Silke
Last Name: Rasmussen
Email:
silke.rasmussen@example.co
m
Phone: 15893401
Gender: female

Рис 3.3. Екран User створеного Flutter проекту

Переходимо до додатку створеного за допомогою Swift та SwiftUI. Він має натсупну структуру.

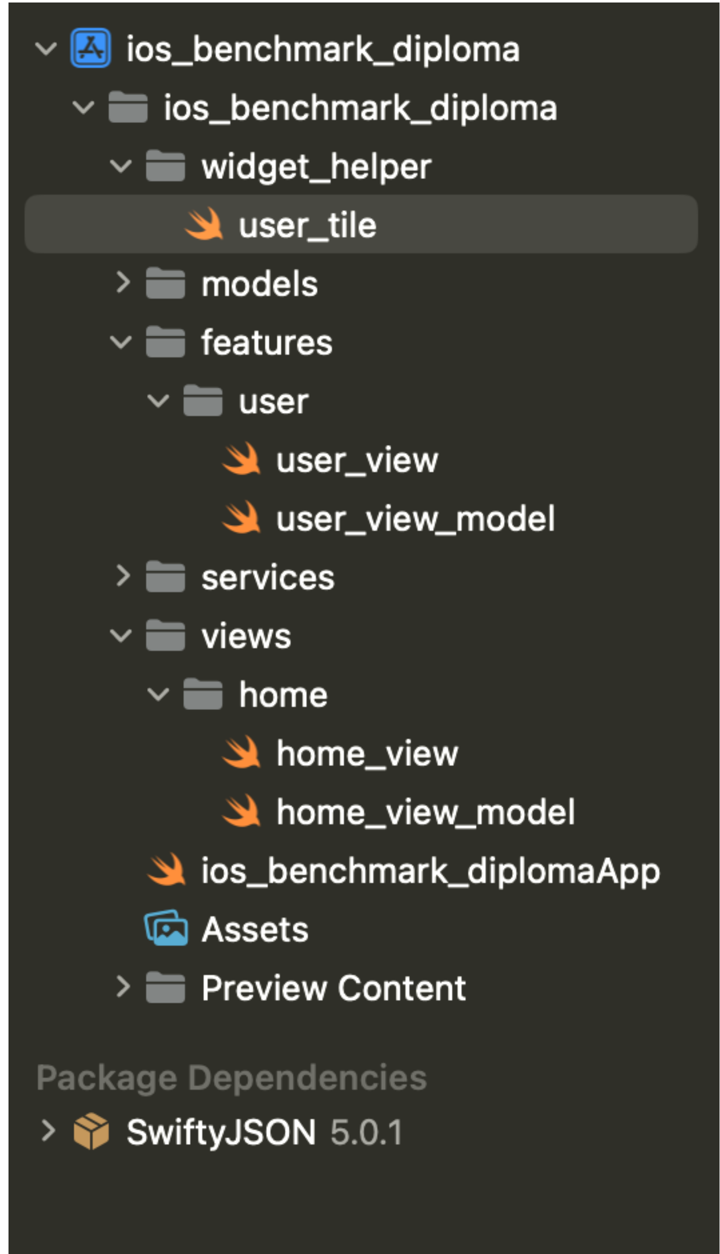


Рис 3.4. Структура створеного SwiftUI проекту

У процесі розробки додатку ми обрали патерн MVVM (Model-View-ViewModel) замість Bloc, оскільки він є широко поширеним та добре прийнятим в середовищі SwiftUI. MVVM забезпечує ефективну організацію бізнес-логіки та візуальної частини додатку, що дозволяє розробляти масштабовані та підтримувані застосунки.

Додатково, ми вирішили спростити структуру шляхом прибирання рівня абстракції Repository. Це було зумовлено бажанням уникнути зайвої складності та зробити код більш прозорим і простим для розуміння. Це не заважає нам ефективно працювати з джерелами даних, проте дозволяє зберігати код більш легким та зрозумілим.

Також, ми використовуємо спрощену навігацію, яку надає нам SwiftUI. Цей фреймворк пропонує зручні та інтуїтивно зрозумілі засоби для навігації між екранами, що спрощує розробку і дозволяє швидше створювати та тестувати функціонал додатку. Однак, це не виключає можливості розширення навігаційного стеку або додаткової кастомізації за потреби.

Такий підхід дозволяє нам ефективно розробляти додаток, використовуючи принципи MVVM, спрощену структуру коду та зручну навігацію, що пропонує SwiftUI, для створення високоякісного та зручного у використанні застосунку.

Ось які View було зроблено за допомогою цього фреймворку.

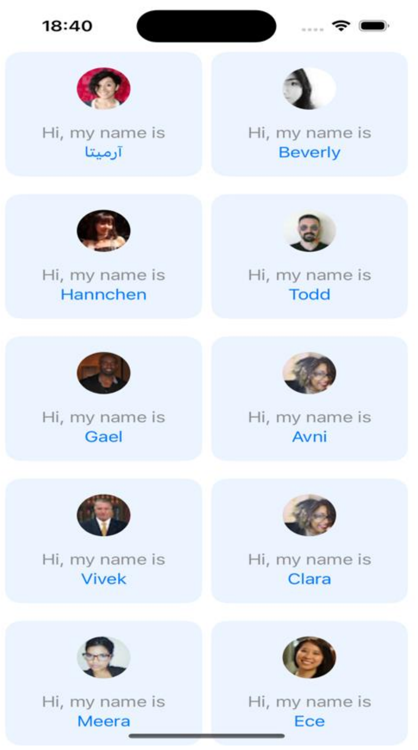


Рис 3.5. Екран Home створеного SwiftUI проекту

18:41

Back

First Name: آرميتا
Last Name: كوتى
Email:
armyt.khwty@example.com
Phone: 032-21614029
Gender: female

Рис 3.6. Екран User створеного SwiftUI проекту

Додаток, розроблений з використанням Jetpack Compose, спирався на ті ж самі основні принципи, що й в SwiftUI. Ми використовували підхід MVVM (Model-View-ViewModel) для організації бізнес-логіки та візуальної частини додатку. Це дозволило нам створити структурований, простий у розумінні та підтримці код.

Крім цього, обидва фреймворки, SwiftUI та Jetpack Compose, подібні за своїм підходом до розробки. Вони пропонують декларативний підхід до створення інтерфейсів користувача, де візуальний ефект залежить від даних, що вони відображають. Це спрощує процес розробки та робить код більш зрозумілим і прозорим.

Обидва фреймворки також надають зручні засоби для роботи з навігацією між екранами та компонентами додатку. Це дозволяє розробникам ефективно керувати взаємодією користувача з додатком та створювати зручний та інтуїтивний інтерфейс.

Отже, якщо врахувати подібний підхід до побудови інтерфейсу та структури коду, SwiftUI та Jetpack Compose можна розглядати як схожі за підходом до розробки мобільних додатків.

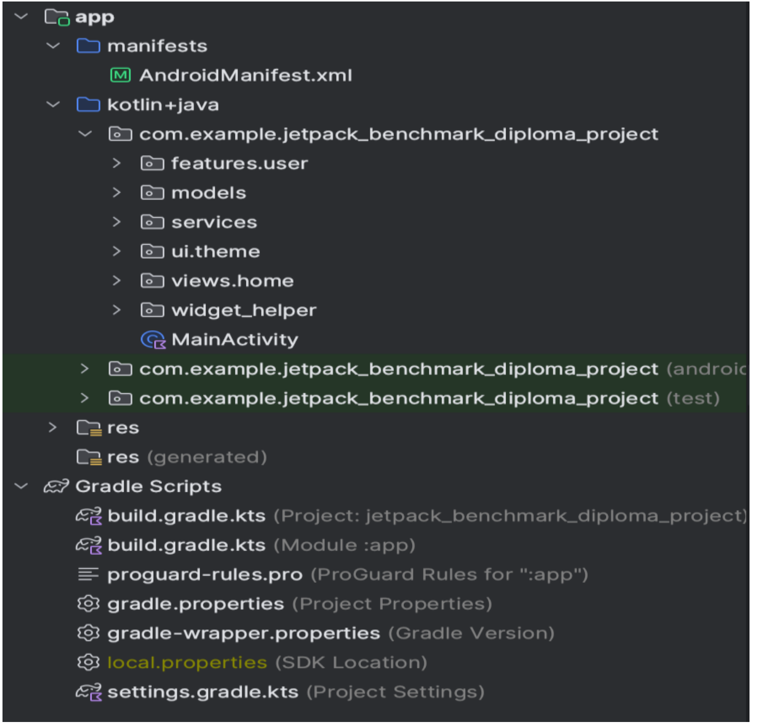


Рис 3.7. Структура створеного Jetpack Compose проекту

Ось які View було зроблено за допомогою цього фреймворку.

Рис 3.8. Екран Home створеного Jetpack Compose проекту

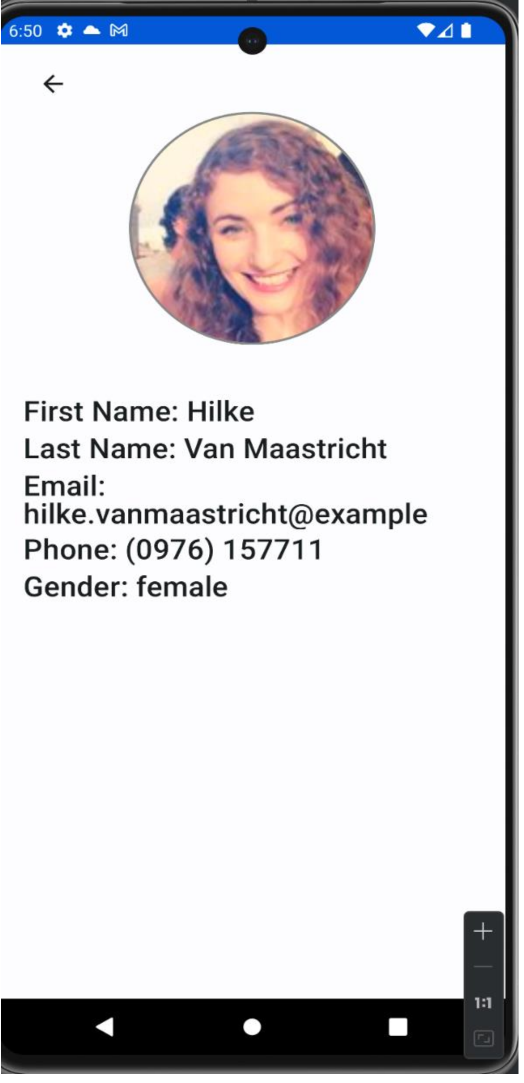


Рис 3.9. Екран User створеного Jetpack Compose проекту

Засновуючись на досвіді розробки додатків з використанням SwiftUI, Jetpack Compose та Flutter, можна зробити висновок, що ці фреймворки, хоч і різних виробників та призначення, пропонують подібні підходи до розробки мобільних додатків.

Обидва SwiftUI та Jetpack Compose використовують декларативний підхід до побудови інтерфейсу та забезпечують можливості для зручної роботи з навігацією. Це робить розробку додатків більш ефективною та дозволяє швидше створювати зручний та інтуїтивний інтерфейс для користувачів.

Також варто відзначити, що обидва фреймворки підтримують використання паттерну MVVM (Model-View-ViewModel), що спрощує організацію бізнес-логіки та візуальної частини додатку, що робить код більш структурованим та легким у розумінні.

Отже, на фоні цих спільних рис можна зазначити, що використання SwiftUI, Jetpack Compose та Flutter забезпечує розробникам сучасні та потужні інструменти для створення мобільних додатків з високоякісним інтерфейсом та ефективною логікою, підвищуючи продуктивність та швидкість розробки.

3.4.2. Аналіз продуктивності

Під час нашого дослідження ми плануємо провести два вимірювання на кожній платформі для кожного з розглянутих фреймворків.

Почнемо з першого вимірювання, яке полягатиме в замірі часу від відправлення запиту на API до моменту, коли дані відображаються на екрані. Це дозволить нам оцінити швидкість отримання та відображення даних з віддаленого джерела і перевірити ефективність роботи фреймворків у цьому аспекті.

Друге вимірювання передбачає відстеження часу, який займає перехід на сторінку конкретного озера після тапу на елементі, і це з анімацією. Це дозволить нам оцінити продуктивність фреймворків у реалізації анімованих переходів між екранами, що є важливим для комфортного користувацького досвіду.

Ці два вимірювання нададуть нам можливість порівняти роботу кожного фреймворку на кожній платформі в аспектах завантаження та відображення даних з API та у реалізації анімованих переходів між екранами, що допоможе зрозуміти їхню продуктивність та швидкість у реальних умовах використання.

Для проведення вимірювань ми плануємо використовувати вбудований функціонал кожного фреймворку, наприклад, `DateTime.now()` у Flutter. Це дозволить нам точно виміряти час виконання певних операцій, таких як відправка запитів на API та анімовані переходи між сторінками.

Отримані часові показники будемо виводити у консоль кожної платформи. Це дозволить нам аналізувати та порівнювати часові інтервали між окремими подіями під час виконання додатків на різних фреймворках. Відображення даних у консолі надасть нам можливість візуально порівняти та проаналізувати результати вимірювань для кожної з платформ.

Після проведення вимірювань на кожній платформі ми запустимо кожен додаток і зробимо скріншоти, де будуть видимі вимірювання часу для кожного окремого етапу, зокрема відправка запиту на API до відображення даних на екрані та час переходу на сторінку конкретного озера з анімацією.

Надалі ми створимо таблицю, в якій порівняємо характеристики результатів вимірювань для кожної платформи. Ця таблиця буде містити дані з кожної з платформ, включаючи часові показники для кожного окремого етапу вимірювань. Це дозволить нам зробити об'єктивний порівняльний аналіз продуктивності кожної платформи та фреймворку у різних аспектах, що були виміряні.

Поділений на різні розділи та зроблений у вигляді таблиці, цей порівняльний аналіз дозволить з легкістю оцінити продуктивність та ефективність кожного фреймворку на кожній з платформ. У таблиці 3.1 показани результати тестів додатків на усіх платформах.

```
Connecting to VM Service at ws://127.0.0.1:52960/5WkBzAbICGQ=/ws
[log] Users Fetch Start Timestamp: 2023-12-06 15:16:28.093619
[log] Users Fetch End Timestamp: 2023-12-06 15:16:29.445901
[log] Navigation Start Timestamp: 2023-12-06 15:16:44.010529
[log] Navigation End Timestamp: 2023-12-06 15:16:44.086585
```

Рис 3.10. Результати тестів на iPhone додатку розробленого за допомогою Flutter

```
Connecting to VM Service at ws://127.0.0.1:51609/uLlvApyplxM=/ws
[log] Users Fetch Start Timestamp: 2023-12-06 16:14:37.288778
[log] Users Fetch End Timestamp: 2023-12-06 16:14:38.256010
D/EGL_emulation( 6134): app_time_stats: avg=23.93ms min=4.44ms max=174.87ms count=47
[log] Navigation Start Timestamp: 2023-12-06 16:14:45.010284
[log] Navigation End Timestamp: 2023-12-06 16:14:45.070682
D/EGL_emulation( 6134): app_time_stats: avg=3260.11ms min=93.48ms max=6426.74ms count=2
```

Рис 3.11. Результати тестів на Android додатку розробленого за допомогою Flutter

```
Start API Fetch Date String: 2023-12-06 16:25:24:2524
End API Fetch Date String: 2023-12-06 16:25:25:2525
Start Push Animation Date String: 2023-12-06 16:25:27:2527
End Push Animation Date: 2023-12-06 16:25:27:2527
```

Рис 3.12. Результати тестів на iPhone додатку розробленого за допомогою SwiftUI

```
Users Fetch Start Timestamp:        Date: 06-12-2023 Time: 16:37:36:3736
Users Fetch Start Timestamp:        Date: 06-12-2023 Time: 16:37:37:3737
Navigation Start Timestamp:         Date: 06-12-2023 Time: 16:37:39:3739
Navigation End Timestamp:           Date: 06-12-2023 Time: 16:37:39:3739
```

Рис 3.13. Результати тестів на Android додатку розробленого за допомогою Jetpack Compose

Таблиця 3.1

Результати тестів на усіх платформах

№ п/п	Платформа	Тест 1 (секунд)	Тест 2 (секунд)
1	Flutter/iOS	1.3523	0.076
1	Flutter/Android	0.9673	0.0604
2	SwiftUI/iOS	1.0001	0.00001
3	Jetpack Compose/Android	1.0001	0.00001

В результаті проведених експериментів встановлено, що в середньому запит на 1000 об'єктів та відображення інтерфейсу на Flutter займає

приблизно 1,25 секунди. У порівнянні з нативними фреймворками, той самий тест на них займає рівно 1.001 секунди, що є ідентичним показником як для SwiftUI, так і для Jetpack Compose.

Це свідчить про те, що на нативних фреймворках, а саме у випадку SwiftUI та Jetpack Compose, час виконання запиту та відображення інтерфейсу є істотно швидшим порівняно з роботою на Flutter. Однак, варто відзначити, що різниця у часі між нативними фреймворками та Flutter не є дуже значущою, але може виявитися помітною для користувачів у певних сценаріях використання додатків.

На підставі результатів другого тесту виявлено, що анімований перехід між сторінками на Flutter займає приблизно 0.07 секунди для iOS та 0.06 секунди для Android. У порівнянні з нативними фреймворками, цей процес займає менше 0.0001 секунди.

Ці цифри свідчать про те, що у нативних фреймворках, на відміну від Flutter, анімований перехід між сторінками виконується надзвичайно швидко, займаючи значно менше часу. Різниця у часі виконання між Flutter та нативними фреймворками у цьому аспекті є суттєвою, і цей показник може бути помітним для користувачів у випадку, коли важлива плавність та швидкість анімацій у додатку.

Отримані результати чітко демонструють вражаючу швидкість та ефективність нативних фреймворків у порівнянні з кросплатформовими рішеннями, такими як Flutter. У тесті на анімований перехід між сторінками час виконання на Flutter був значно вищим (0.06-0.07 секунд), тоді як на нативних фреймворках цей процес займав надзвичайно короткий час (менше 0.0001 секунди).

Однак, важливо враховувати, що різниця у часі, хоч і значна у цьому тесті, може бути менш помітною для кінцевих користувачів в практичних умовах використання. Більшість людей, ймовірно, не помітять чи не відчують цієї різниці у повсякденному використанні додатку.

Щодо виправдання витрат на більшу кількість розробників під різні платформи та синхронізованість команди - це складний питання, де потрібно зважати на кілька факторів. Наприклад, кросплатформові фреймворки, як Flutter, можуть значно зменшити витрати на розробку та підтримку додатку на кількох платформах, забезпечуючи єдиний код для різних ОС.

Проте, наявність різних факторів, таких як продуктивність, швидкість розробки, потреби користувачів та особливості самого проекту, можуть впливати на рішення про вибір фреймворку. У абсолютній більшості випадків, ефективне використання кросплатформових фреймворків може переважати незначну різницю у продуктивності в порівнянні з нативними фреймворками.

Отже, хоча нативні фреймворки демонструють вражаючу швидкість, остаточне рішення про вибір фреймворку повинно ґрунтуватися на конкретних потребах проекту та враховувати різні аспекти розробки, витрати на підтримку, потреби користувачів та інші ключові фактори.

3.4.3. Оцінка ефективності ресурсів

Для порівняння споживання пам'яті, продуктивності та енергоефективності додатків на кожному фреймворку, ми використовуватимемо вбудовані Debug tools, які надаються кожною платформою та фреймворком для аналізу різних аспектів додатку.

У Flutter для аналізу споживання пам'яті можна використовувати DevTools, які мають різні вкладки, включаючи "Memory" та "Performance", що дозволяє відстежувати споживання пам'яті під час роботи додатку.

- На нативних платформах також є вбудовані інструменти, такі як Android Profiler для Android і Instruments для iOS, які дозволяють аналізувати використання пам'яті під час роботи додатку.

Для оцінки продуктивності на Flutter можна використовувати DevTools, де є розділ "Performance", що дозволяє відстежувати час виконання різних операцій та функцій.

На нативних платформах також доступні інструменти для аналізу продуктивності, такі як Android Profiler та Instruments, які допомагають відстежувати швидкодію різних компонентів додатку.

Визначення енергоефективності на Flutter може бути здійснене через DevTools, де можна виміряти рівень споживання батареї під час виконання додатку.

Нативні платформи також надають засоби для вимірювання енергоефективності, такі як Battery Historian для Android та Battery Usage в Instruments для iOS.

Застосування цих вбудованих інструментів дозволить нам об'єктивно порівняти різні аспекти додатків, включаючи споживання пам'яті, продуктивність та енергоефективність на кожному з фреймворків. Це допоможе визначити ефективність роботи додатків під кутом різних показників та вибрати оптимальний фреймворк для конкретних потреб проекту.

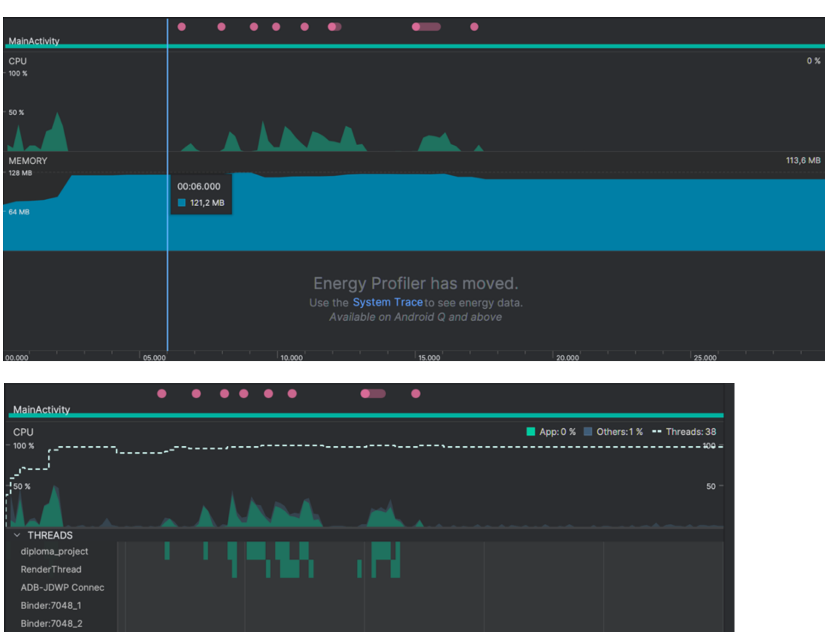
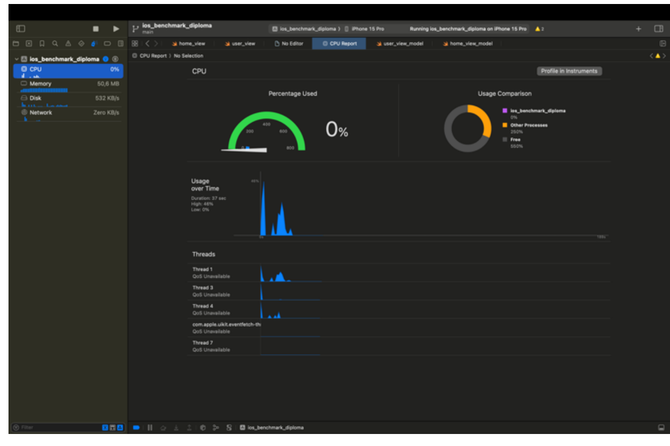


Рис 3.14. Результати тестів Profiler на Android



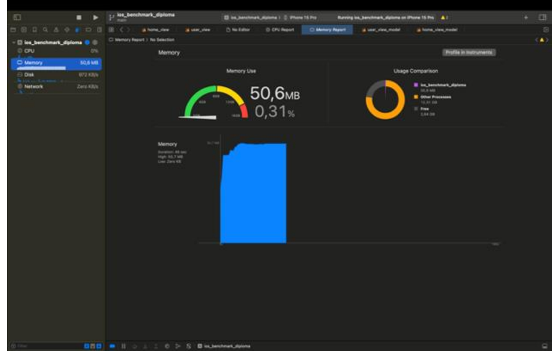


Рис 3.15. Результати тестів Profiler на iOS

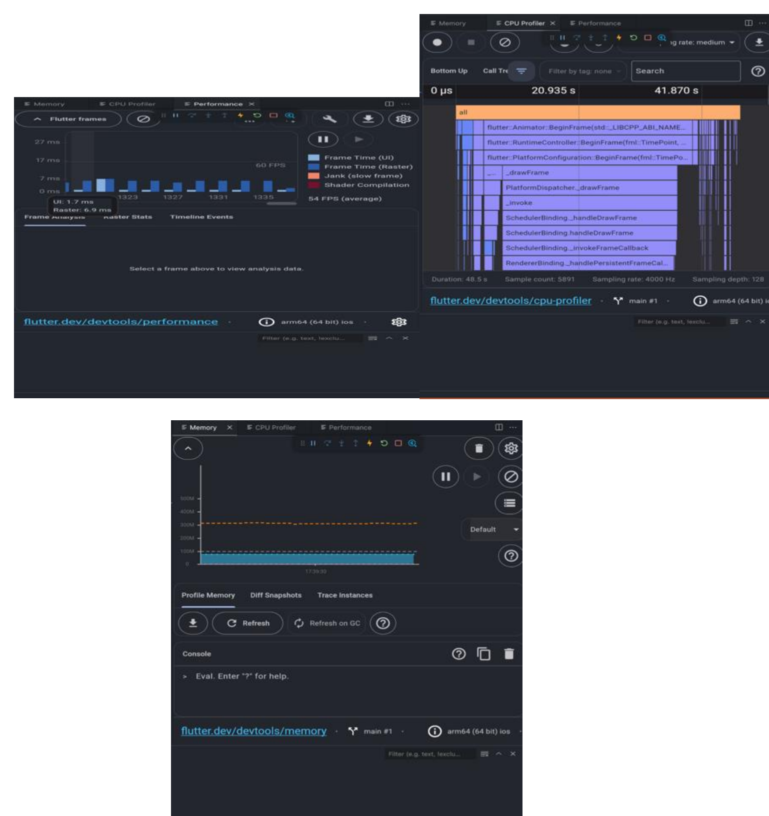


Рис 3.16. Результати тестів Profiler Flutter на iOS

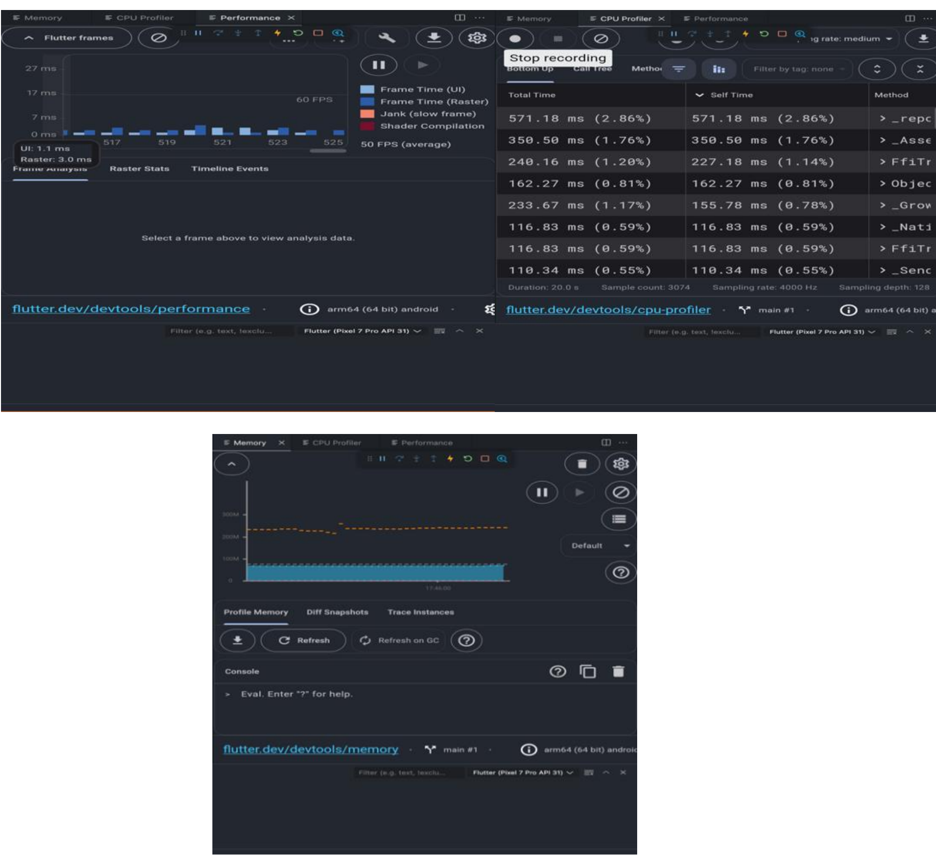


Рис 3.17. Результати тестів Profiler Flutter на Android

Дані, отримані від профайлера, демонструють, що SwiftUI показав найкращі результати серед усіх досліджуваних фреймворків. Його продуктивність, ефективність та економічне використання пам'яті надзвичайно вражають.

У той же час, Flutter також вразив своєю ефективністю та продуктивністю. Хоча результати не дотягують до рівня SwiftUI, проте вони залишаються дуже конкурентними та вражаючими, виявляючи велику ефективність розробки кросплатформових додатків.

Особливо важливо відзначити, що за використанням пам'яті, Flutter показав дуже високий рівень продуктивності, наближаючись до показників нативних Android додатків. Це свідчить про те, що Flutter вміє оптимально використовувати ресурси пристрою, забезпечуючи високу ефективність та швидкість роботи.

Отже, хоча SwiftUI показав найкращі результати з точки зору продуктивності та ефективності, Flutter також вражає своєю ефективністю та можливостями. Обидва ці фреймворки виявилися високоефективними інструментами для розробки додатків, здатними забезпечити високу продуктивність та ефективне використання ресурсів пристроїв.

3.5. Висновки до третього розділу

Це дослідження відобразило різноманітні аспекти розробки додатків за допомогою різних фреймворків, зосереджуючись на Flutter як на перспективній та потужній технології. Результати нашого дослідження відображають ефективність та функціональність Flutter у багатьох аспектах.

Flutter виявився потужним інструментом для розробки кросплатформових додатків, надаючи можливість створення однаково якісних інтерфейсів для різних платформ. Його здатність до гнучкості, швидкодії та

стабільності вражає, забезпечуючи зручність у розробці та підтримці додатків.

Ця технологія має велике майбутнє, оскільки Flutter не лише дозволяє швидко створювати додатки для різних платформ, а й забезпечує високу якість інтерфейсу та продуктивність. Наша робота та дослідження у сфері Flutter роблять свій внесок у подальший розвиток цієї технології, розширюючи її можливості та сприяючи її поширенню серед розробників.

Ми побачили, що Flutter виявляється чудовим вибором для швидкої та ефективної розробки, забезпечуючи однакову продуктивність на різних платформах. Його потужні можливості у поєднанні з широким спектром інструментів для аналізу та підтримки роблять його важливим гравцем у світі розробки мобільних додатків, і його вплив на індустрію обіцяє бути значним у майбутньому.

ВИСНОВОК

У результаті проведеного дослідження було встановлено, що фреймворк Flutter має високу продуктивність у порівнянні з нативними мовами розробки мобільних додатків Swift, Kotlin. Flutter дозволяє створювати кроссплатформенні додатки з одним кодом, який працює на різних платформах без втрати якості та швидкості. Flutter також має багатий набір компонентів, які дозволяють створювати декларативні UI з анімаціями та ефектами. Flutter є дуже гарним інструментом для розробки мобільного ПЗ, який забезпечує гнучкість, ефективність та конкурентоспроможність.

Для досягнення мети дослідження було використано наступні методи та інструменти: Аналіз літературних джерел та наукових публікацій, що стосуються кроссплатформенної розробки та фреймворку Flutter. Порівняльний аналіз фреймворків Flutter, SwiftUI та Jetpack Compose, їхньої архітектури, компонентів та особливостей. Розробка мобільного додатку для платформ Android та iOS з використанням різних фреймворків та мов програмування. Тестування та оцінка продуктивності розроблених додатків за допомогою різних критеріїв, таких як час завантаження, час відгуку, використання пам'яті, використання батареї, кількість помилок тощо. Flutter демонструє гарні показники за більшістю критеріїв, таких як час завантаження, час відгуку, використання пам'яті, використання батареї, кількість помилок тощо.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Architectures comparing for SwiftUI. [Електронний ресурс]. Режим доступу: https://medium.com/@vladislavshkodich/architectures-comparing-for-swiftui-6351f1fb3605

2. Developer SwiftUI [Електронний ресурс]. Режим доступу: https://developer.apple.com/xcode/swiftui/

3. Developer Swift [Електронний ресурс]. Режим доступу: https://developer.apple.com/swift/

4. Introducing SwiftUI [Електронний ресурс]. Режим доступу: https://developer.apple.com/tutorials/swiftui

5. Kotlin for Android [Електронний ресурс]. Режим доступу: https://kotlinlang.org/docs/android-overview.html

6. Basic syntax [Електронний ресурс]. Режим доступу: https://kotlinlang.org/docs/basic-syntax.html

7. Get started with Jetpack Compose [Електронний ресурс]. Режим доступу: https://developer.android.com/jetpack/compose/documentation

8. Lifecycle of composables [Електронний ресурс]. Режим доступу: https://developer.android.com/jetpack/compose/lifecycle

9. State and Jetpack Compose [Електронний ресурс]. Режим доступу: https://developer.android.com/jetpack/compose/state

10. Architecting your Compose UI [Електронний ресурс]. Режим доступу: https://developer.android.com/jetpack/compose/architecture

11. Flutter architectural overview [Електронний ресурс]. Режим доступу: https://docs.flutter.dev/resources/architectural-overview

12. Inside Flutter [Електронний ресурс]. Режим доступу: https://docs.flutter.dev/resources/inside-flutter

13. Flutter's build modes [Електронний ресурс]. Режим доступу: https://docs.flutter.dev/testing/build-modes

14. Migration strategy [Електронний ресурс]. Режим доступу: https://developer.android.com/jetpack/compose/migrate/strategy

15. Lifecycle of composables [Електронний ресурс]. Режим доступу: https://developer.android.com/jetpack/compose/lifecycle

16. Debugging Flutter apps programmatically [Електронний ресурс]. Режим доступу: https://docs.flutter.dev/testing/code-debugging

17. Testing Flutter apps [Електронний ресурс]. Режим доступу: https://docs.flutter.dev/testing/overview

18. Debugging Flutter apps [Електронний ресурс]. Режим доступу: https://docs.flutter.dev/testing/debugging

19. Impeller [Електронний ресурс]. Режим доступу: https://docs.flutter.dev/perf/impeller

20. Rendering performance [Електронний ресурс]. Режим доступу: https://docs.flutter.dev/perf/rendering-performance

21. Shader compilation jank [Електронний ресурс]. Режим доступу: https://docs.flutter.dev/perf/shader

22. Top Flutter State Management Libraries in 2023. Canadian Software Agency Inc. article. LinkedIn. [Електронний ресурс]. Режим доступу: https://ca.linkedin.com/company/canadian-software-agency?trk=article-ssr-frontend-pulse_publisher-author-card

23. Canadian Agency. Mobile App Development: 2022 Trends. [Електронний ресурс]. Режим доступу: https://canadian.agency/mobile-app-development-2022-trends/

24. List of state management approaches. Flutter Docs. [Електронний ресурс]. Режим доступу: https://docs.flutter.dev/data-and-backend/state-mgmt/options

25. Comparative study on Flutter State Managment [Електронний ресурс]. Режим доступу:

https://blog.alifakbar.com/post/658377568699957248/comparative-study-on-flutter-state-management

26. Using the Performance view: [Електронний ресурс]. Режим доступу: https://docs.flutter.dev/tools/devtools/performance

27. Using the CPU profiler view: [Електронний ресурс]. Режим доступу: https://docs.flutter.dev/tools/devtools/cpu-profiler

28. Using the Memory view: [Електронний ресурс]. Режим доступу: https://docs.flutter.dev/tools/devtools/memory

29. Using the debugger: [Електронний ресурс]. Режим доступу: https://docs.flutter.dev/tools/devtools/debugger

30. Flutter's build modes [Електронний ресурс]. Режим доступу: https://docs.flutter.dev/testing/build-modes

31. Build and release an Android app [Електронний ресурс]. Режим доступу: https://docs.flutter.dev/deployment/android

32. Build and release an Android app [Електронний ресурс]. Режим доступу: https://docs.flutter.dev/deployment/android

33. Visual Studio Code [Електронний ресурс]. Режим доступу: https://docs.flutter.dev/tools/vs-code

34. DevTools [Електронний ресурс]. Режим доступу: https://docs.flutter.dev/tools/devtools/overview

35. SwiftUI [Електронний ресурс]. Режим доступу: Passing Data between Views: https://ix76y.medium.com/swiftui-passing-data-between-views-446bc8b4805

36. Jetpack Compose performance [Електронний ресурс]. Режим доступу: https://developer.android.com/jetpack/compose/performance

37. Composition tracing [Електронний ресурс]. Режим доступу: https://developer.android.com/jetpack/compose/tooling/tracing

38. Side-effects in Compose [Електронний ресурс]. Режим доступу: https://developer.android.com/jetpack/compose/side-effects

39. Navigating with Compose [Електронний ресурс]. Режим доступу: https://developer.android.com/jetpack/compose/navigation

40. Scaffold [Електронний ресурс]. Режим доступу: https://developer.android.com/jetpack/compose/components/scaffold

41. Button [Електронний ресурс]. Режим доступу: https://developer.android.com/jetpack/compose/components/button

42. Card [Електронний ресурс]. Режим доступу: https://developer.android.com/jetpack/compose/components/card

43. Material Design 3 in Compose [Електронний ресурс]. Режим доступу: https://developer.android.com/jetpack/compose/designsystems/material3

44. Images and graphics in Compose [Електронний ресурс]. Режим доступу: https://developer.android.com/jetpack/compose/graphics

45. Optimizing performance for images [Електронний ресурс]. Режим доступу: https://developer.android.com/jetpack/compose/graphics/images/optimization

46. SwiftUI Grid [Електронний ресурс]. Режим доступу: https://sarunw.com/posts/swiftui-grid/

47. Чому НЕ варто використовувати SwiftUI (у всякому разі, поки що) [Електронний ресурс]. Режим доступу: https://dou.ua/forums/topic/33288/

48. Animations [Електронний ресурс]. Режим доступу: https://developer.apple.com/documentation/SwiftUI/Animations

49. Core Data [Електронний ресурс]. Режим доступу: https://developer.apple.com/documentation/coredata

50. Core Graphics [Електронний ресурс]. Режим доступу: https://developer.apple.com/documentation/coregraphics

51. SwiftUI [Електронний ресурс]. Режим доступу: https://developer.apple.com/documentation/swiftui/

52. Positioning and sizing windows [Електронний ресурс]. Режим доступу: https://developer.apple.com/documentation/visionos/positioning-and-sizing-windows

53. Diagnosing and resolving bugs in your running app [Електронний ресурс]. Режим доступу: https://developer.apple.com/documentation/xcode/diagnosing-and-resolving-bugs-in-your-running-app

54. Preparing your app for distribution [Електронний ресурс]. Режим доступу: https://developer.apple.com/documentation/xcode/preparing-your-app-for-distribution

55. Testing a release build [Електронний ресурс]. Режим доступу: https://developer.apple.com/documentation/xcode/testing-a-release-build

56. What is SwiftUI [Електронний ресурс]. Режим доступу: https://www.hackingwithswift.com/quick-start/swiftui/what-is-swiftui

57. Building a menu using List [Електронний ресурс]. Режим доступу: https://www.hackingwithswift.com/quick-start/swiftui/building-a-menu-using-list

58. Binding and forms [Електронний ресурс]. Режим доступу: https://www.hackingwithswift.com/quick-start/swiftui/bindings-and-forms

59. Mobile operating systems' market share worldwide from January 2012 to October 2020. – Електрон. дан. – Режим доступу: https://www.statista.com/statistics/272698/global-market-share-held-by-mobileoperatingsystems-since-2009/.

60. Mike Katz, Kevin David Moore, Vincent Ngo. Flutter Apprentice (First Edition): Learn to Build Cross-Platform Apps, 2019. 124 с.

61. Dieter Meiller. Modern App Development with Dart and Flutter 2: A Comprehensive Introduction to Flutter (de Gruyter Stem), 2020. 843 с.

Додаток А

КОД ПРОГРАМ

Main.dart

```
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import
'package:flutter_diploma_benchmark_project/repositories/user_repository/interface_user_repository.dart';
import
'package:flutter_diploma_benchmark_project/repositories/user_repository/services/user_service.dart';
import
'package:flutter_diploma_benchmark_project/repositories/user_repository/user_repository.dart';
import
'package:flutter_diploma_benchmark_project/views/home/home_view.dart';

class DebugBlocObserver extends BlocObserver {
  @override
  void onEvent(Bloc bloc, Object? event) {
    super.onEvent(bloc, event);
    debugPrint(event.toString());
  }

  @override
  void onTransition(Bloc bloc, Transition transition) {
    super.onTransition(bloc, transition);
    debugPrint(transition.toString());
  }

  @override
  void onError(BlocBase bloc, Object error, StackTrace stackTrace) {
    super.onError(bloc, error, stackTrace);
    debugPrint(error.toString());
  }
}

void main() {
  runApp(
    MultiRepositoryProvider(
      providers: [
        RepositoryProvider<IUserRepository>(
          lazy: true,
          create: (context) => UserRepository(
            userService: UserService(),
          ),
        ),
      ],
      child: const MyApp(),
    ),
  );
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
```

```
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      title: 'Benchmark',
      theme: ThemeData(
        colorScheme:                ColorScheme.fromSeed(seedColor:
Colors.deepPurple),
        useMaterial3: true,
      ),
      home: const HomeView(),
    );
  }
}
```

Home_view.dart

```
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import
'package:flutter_diploma_benchmark_project/repositories/user_repository/int
erface_user_repository.dart';
import
'package:flutter_diploma_benchmark_project/repositories/user_repository/mod
els/user_model.dart';
import
'package:flutter_diploma_benchmark_project/services/overlay_service.dart';
import
'package:flutter_diploma_benchmark_project/services/toast_service.dart';
import
'package:flutter_diploma_benchmark_project/views/home/bloc/home_bloc.dart';
import
'package:flutter_diploma_benchmark_project/views/home/navigation/home_navig
ator.dart';
import
'package:flutter_diploma_benchmark_project/widget_helper/user_tile.dart';

class HomeView extends StatelessWidget {
  const HomeView({
    super.key,
  });

  @override
  Widget build(BuildContext context) {
    return BlocProvider(
      create: (context) => HomeBloc(
        userRepository: context.read<IUserRepository>(),
        navigator: HomeNavigator(context),
      )..add(BlocReady()),
      child: const HomeViewLayout(),
    );
  }
}

class HomeViewLayout extends StatelessWidget {
  const HomeViewLayout({
    super.key,
  });

  @override
  Widget build(BuildContext context) {
```

```
    return BlocConsumer<HomeBloc, HomeState>(
      listener: (context, state) {
        if (state.isBusy) {
          OverlayService.instance.showBusyOverlay(
            context: context,
            size: MediaQuery.of(context).size,
          );
        } else {
          OverlayService.instance.closeBusyOverlay(context);
        }

        if (state.errorMessage != '') {
          ToastService.showToast(
            message: state.errorMessage,
            context: context,
            error: true,
          );
        }
      },
      builder: (context, state) {
        final HomeBloc bloc = context.read<HomeBloc>();

        return Scaffold(
          body: GridView.count(
            crossAxisCount: 2,
            children: [
              for (UserModel user in (state.users ?? []))
                UserTile(
                  user: user,
                  onTap: () {
                    bloc.add(UserTileTap(userModel: user));
                  },
                ),
            ],
          ),
        );
      },
    );
  }
}
```

User_view.dart

```
import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import
'package:flutter_diploma_benchmark_project/features/user/bloc/user_bloc.dar
t';
import
'package:flutter_diploma_benchmark_project/features/user/navigation/user_na
vigator.dart';
import
'package:flutter_diploma_benchmark_project/repositories/user_repository/mod
els/user_model.dart';
import
'package:flutter_diploma_benchmark_project/services/overlay_service.dart';
import
'package:flutter_diploma_benchmark_project/services/toast_service.dart';

class UserView extends StatelessWidget {
```

```
  const UserView({
    super.key,
    required this.userModel,
  });

  final UserModel userModel;

  @override
  Widget build(BuildContext context) {
    return BlocProvider(
      create: (context) => UserBloc(
        navigator: UserNavigator(context),
      )..add(BlocReady()),
      child: UserViewLayout(
        userModel: userModel,
      ),
    );
  }
}

class UserViewLayout extends StatelessWidget {
  const UserViewLayout({
    super.key,
    required this.userModel,
  });

  final UserModel userModel;

  @override
  Widget build(BuildContext context) {
    return BlocConsumer<UserBloc, UserState>(
      listener: (context, state) {
        if (state.isBusy) {
          OverlayService.instance.showBusyOverlay(
            context: context,
            size: MediaQuery.of(context).size,
          );
        } else {
          OverlayService.instance.closeBusyOverlay(context);
        }

        if (state.errorMessage != '') {
          ToastService.showToast(
            message: state.errorMessage,
            context: context,
            error: true,
          );
        }
      },
      builder: (context, state) {
        final UserBloc bloc = context.read<UserBloc>();

        return Scaffold(
          appBar: AppBar(
            leading: IconButton(
              onPressed: () {
                bloc.navigator.pop();
              },
              icon: const Icon(CupertinoIcons.chevron_back),
            ),
          ),
          body: Padding(
```

```
            padding: const EdgeInsets.symmetric(horizontal: 20),
            child: Column(
              crossAxisAlignment: CrossAxisAlignment.start,
              children: [
                Padding(
                  padding: const EdgeInsets.only(bottom: 20),
                  child: Center(
                    child: CircleAvatar(
                      radius: 100.0,
                      backgroundImage: NetworkImage(
                        userModel.picture.large,
                      ),
                      backgroundColor: Colors.transparent,
                    ),
                  ),
                ),
                Text(
                  'First Name: ${userModel.firstName}',
                  style: const TextStyle(
                    fontSize: 24,
                    fontWeight: FontWeight.w500,
                  ),
                ),
                Text(
                  'Last Name: ${userModel.lastName}',
                  style: const TextStyle(
                    fontSize: 24,
                    fontWeight: FontWeight.w500,
                  ),
                ),
                Text(
                  'Email: ${userModel.email}',
                  style: const TextStyle(
                    fontSize: 24,
                    fontWeight: FontWeight.w500,
                  ),
                ),
                Text(
                  'Phone: ${userModel.phone}',
                  style: const TextStyle(
                    fontSize: 24,
                    fontWeight: FontWeight.w500,
                  ),
                ),
                Text(
                  'Gender: ${userModel.gender}',
                  style: const TextStyle(
                    fontSize: 24,
                    fontWeight: FontWeight.w500,
                  ),
                ),
              ],
            ),
          ),
        );
      },
    );
  }
}
```

Home_view.swift

```
import SwiftUI

struct home_view: View {
    @StateObject private var viewModel = HomeViewModel()

    var body: some View {
        NavigationView {
            ZStack() {

                if !viewModel.users.isEmpty {
                    home_view_layout(users: $viewModel.users)
                }

                if viewModel.isBusy {
                    ProgressView()
                        .frame(width: 1000, height: 1000)
                        .background(Color.black)
                        .opacity(0.8)

.progressViewStyle(CircularProgressViewStyle(tint: Color.white))
                        .controlSize(.large)
                }
            }.onAppear{
                Task {
                    if viewModel.users.isEmpty {
                        await viewModel.onReady()
                    }
                }
            }
        }
    }
}

struct home_view_layout: View {
    @Binding var users: [UserModel]

    let columns = [
        GridItem(),
        GridItem(),
    ]

    var body: some View {
        ScrollView {
            LazyVGrid(columns: columns, spacing: 20) {
                ForEach($users) { item in
                    user_tile(userModel: item)
                }
            }
            .padding(.horizontal)
        }
        .frame(maxHeight:  UIScreen.main.bounds.height)
    }
}

#Preview {
    home_view()
}
```

User_view.swift

```
import SwiftUI
```

```
struct user_view: View {
    @Binding var user: UserModel

    var body: some View {
        user_view_layout(user: $user)
    }
}

struct user_view_layout: View {
    @Binding var user: UserModel

    @StateObject private var viewModel = UserViewModel()

    var body: some View {
            VStack() {

                AsyncImage(url: URL(string: user.picture.large))
                    .clipShape(Circle())
                    .frame(width: 100, height: 100)

                VStack(alignment: .leading) {

                    Text(
                        "First Name: " + user.firstName
                    )
                    .fontWeight(.heavy)
                    .font(.system(size: 24))
                    .padding(.top, 50)

                    Text(
                        "Last Name: " + user.lastName
                    )
                    .fontWeight(.heavy)
                    .font(.system(size: 24))

                    Text(
                        "Email: " + user.email
                    )
                    .fontWeight(.heavy)
                    .font(.system(size: 24))

                    Text(
                        "Phone: " + user.phone
                    )
                    .fontWeight(.heavy)
                    .font(.system(size: 24))

                    Text(
                        "Gender: " + user.gender
                    )
                    .fontWeight(.heavy)
                    .font(.system(size: 24))

                }
            }
            .frame(height:   UIScreen.main.bounds.height   -   200,
alignment: .top)
            .onAppear {
                viewModel.onReady()
            }
        }
```

```
}
```

Home_view.kt

```
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.lazy.grid.GridCells
import androidx.compose.foundation.lazy.grid.LazyVerticalGrid
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.ui.unit.dp
import com.example.jetpack_benchmark_diploma_project.views.home.home_view_model
import com.example.jetpack_benchmark_diploma_project.widget_helper.UserTile

@Composable
fun HomeView(viewModel: home_view_model) {
    LaunchedEffect(Unit, block = {
        viewModel.getUserList()
    })

    if (viewModel.selectedIndex == null) {
        LazyVerticalGrid(
            columns = GridCells.Fixed(2),
            contentPadding = PaddingValues(
                horizontal = 20.dp,
                vertical = 10.dp
            )
        ) {
            items(
                viewModel.userList.count()
            ) { index ->
                UserTile(user = viewModel.userList[index]) {
                    viewModel.selectUser(index)
                }
            }
        }
    } else {
        UserView(user = viewModel.userList[viewModel.selectedIndex!!], onClick = {
            viewModel.selectUser(null)
        })
    }
}
```

user_view.kt

```
import android.util.Log
import androidx.compose.foundation.Image
import androidx.compose.foundation.border
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.layout.wrapContentWidth
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.ArrowBack
```

```
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import coil.compose.rememberAsyncImagePainter
import com.example.jetpack_benchmark_diploma_project.models.UserModel
import java.text.SimpleDateFormat
import java.util.Date

@Composable
fun UserView(user: UserModel, onClick: () -> Unit) {
    LaunchedEffect(Unit, block = {
        val navigationEndTimestampFormat = SimpleDateFormat("'Date: 'dd-MM-
yyyy 'Time: 'HH:mm:ss:ms")

        val navigationEndTimestamp =
navigationEndTimestampFormat.format(Date())
        Log.d("Navigation End Timestamp: ", navigationEndTimestamp)
    })

    Column(
        Modifier
            .fillMaxWidth()
            .padding(horizontal = 20.dp)
            .padding(vertical = 10.dp), horizontalAlignment =
Alignment.CenterHorizontally) {
        Row(Modifier
            .fillMaxWidth()) {
            IconButton(onClick = {
                onClick()
            }) {
                Icon(Icons.Filled.ArrowBack, contentDescription = "Информа-
ция о приложении")
            }
        }
        Image(
            painter = rememberAsyncImagePainter(user.picture.large),
            contentDescription = "avatar",
            contentScale = ContentScale.Fit,// crop the image if it's not a
square
            modifier = Modifier
                .size(200.dp)
                .clip(CircleShape)                       // clip to the
circle shape
                .border(2.dp, Color.Gray, CircleShape)
        )
        Spacer(modifier = Modifier.height(40.dp))
            Text(
                text = "First Name: ${user.name.first}",
                textAlign = TextAlign.Start,
                maxLines = 1,
```

```
                style = MaterialTheme.typography.titleMedium,
                modifier = Modifier
                    .fillMaxWidth()
                    .wrapContentWidth(Alignment.Start),
                fontSize = 24.sp,
                fontWeight = FontWeight(500),
            )
            Text(
                text = "Last Name: ${user.name.last}",
                textAlign = TextAlign.Start,
                maxLines = 1,
                style = MaterialTheme.typography.titleMedium,
                modifier = Modifier
                    .fillMaxWidth()
                    .wrapContentWidth(Alignment.Start),
                fontSize = 24.sp,
                fontWeight = FontWeight(500),
            )
            Text(
                text = "Email: ${user.email}",
                textAlign = TextAlign.Start,
                maxLines = 2,
                style = MaterialTheme.typography.titleMedium,
                modifier = Modifier
                    .fillMaxWidth()
                    .wrapContentWidth(Alignment.Start),
                fontSize = 24.sp,
                fontWeight = FontWeight(500),
            )
            Text(
                text = "Phone: ${user.phone}",
                textAlign = TextAlign.Start,
                maxLines = 1,
                style = MaterialTheme.typography.titleMedium,
                modifier = Modifier
                    .fillMaxWidth()
                    .wrapContentWidth(Alignment.Start),
                fontSize = 24.sp,
                fontWeight = FontWeight(500),
            )
            Text(
                text = "Gender: ${user.gender}",
                textAlign = TextAlign.Start,
                maxLines = 1,
                style = MaterialTheme.typography.titleMedium,
                modifier = Modifier
                    .fillMaxWidth()
                    .wrapContentWidth(Alignment.Start),
                fontSize = 24.sp,
                fontWeight = FontWeight(500),
            )
        }
    }
```

Додаток Б

ПЕРЕЛІК ДОКУМЕНТІВІВ НА ОПТИЧНОМУ НОСІЇ

Ім’я файла	**Опис**
Пояснювальні документи	
Диплом_Цалюк.doc	Пояснювальна записка роботи. Документ Word.
Диплом_Цалюк.pdf	Пояснювальна записка роботи в форматі PDF
Програма	
Program.rar	Архів. Містить коди програми і откомпільовану програму
Презентація	
Презентація_Цалюк.ppt	Презентація роботи