МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ БІЗНЕС-КОЛЕДЖ

кафедра комп’ютерної інженерії та інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему
Порівняння декларативних UI-фреймворків

Виконала: студентка групи 1П-20
Спеціальності
121 – «Інженерія програмного забезпечення»

Софія ЄРЕМЕНКО

Керівник:
Станіслав МАРЧЕНКО

Черкаси 2024

ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ БІЗНЕС-КОЛЕДЖ

Кафедра комп'ютерної інженерії та інформаційних технологій
(повна назва випускової кафедри)

Спеціальність 121 "Інженерія програмного забезпечення"
(шифр і назва спеціальності)

Освітня програма Інженерія програмного забезпечення
(назва освітньої програми)

ЗАТВЕРДЖУЮ

Завідувач кафедри
комп'ютерної інженерії та
інформаційних технологій
(назва кафедри)
_____ Хотунов В.І.
(підпис) (ПІБ)
«_____»_____2023 р.

З А В Д А Н Н Я

НА ВИПУСКНУ РОБОТУ СТУДЕНТУ

Єременко Софії В'ячеславівни
(прізвище, ім'я, по батькові студента)

1. Тема випускної роботи Порівняння декларативних UI-фреймворків
Керівник роботи Марченко Станіслав Віталійович, спеціаліст І категорії
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом закладу вищої освіти від " 13 " жовтня 2023 року № 65У.
2. Строк подання студентом випускної роботи 03.06.2024
3. Вихідні дані до випускної роботи мови програмування Kotlin, Swift, фреймворки Flutter, React Native, SwiftUI, Jetpack Compose, Combine, інструменти розробки Android Studio, Xcode, інтеграція з інструментами Firebase, GitHub, Jira, Jenkins, Docker.
4. Зміст випускної роботи (перелік питань, які потрібно розробити) огляд декларативних UI-фреймворків (опис основних принципів та функціональних можливостей фреймворків у мобільній розробці, переваги та недоліки існуючих фреймворків, розповсюдженість декларативних UI-фреймворків серед мобільних розробників), технічний аналіз декларативних UI-фреймворків (детальний огляд синтаксису та основних конструкцій фреймворків, аналіз прикладів коду та його використання у реальних проектах, підтримка функціональності та можливостей фреймворків), типові сценарії використання декларативних UI-фреймворків (практичні приклади використання фреймворків у мобільній розробці, особливості та ефективність в конкретних сценаріях розробки, адаптація декларативних UI-фреймворків до вимог конкретного проекту).
5. Дата видачі завдання 15.09.2023р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Терміни виконання етапів	Примітка про виконання з підписами керівника і студента
1	Вступ	20.10.2023	
2	Розділ 1. Огляд декларативних UI-фреймворків	22.12.2023	
3	Розділ 2. Технічний аналіз декларативних UI-фреймворків	15.03.2024	
4	Розділ 3. Типові сценарії використання декларативних UI-фреймворків	15.05.2024	
5	Висновки	17.05.2024	
6	Оформлення випускної роботи (чистовий варіант)	27.05.2024	
7	Здача випускної роботи на кафедру для рецензування (за 14 днів до захисту)	31.05.2024	
8	Перевірка випускної роботи на наявність ознак плагіату (за 10 днів до захисту)	03.06.2024	
9	Подання випускної роботи на затвердження завідувачу кафедри (за 7 днів до захисту)	06.06.2024	

Студентка _____ Єременко С.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Марченко С.В.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Ця дипломна робота присвячена порівняльному аналізу декларативних UI-фреймворків у мобільній розробці. З огляду на стрімкий розвиток мобільних технологій та значну важливість користувацького інтерфейсу для успіху мобільних додатків, дослідження різних декларативних UI-фреймворків має велике практичне значення. Метою роботи є детальне вивчення та порівняння різних декларативних UI-фреймворків з метою з’ясування їх переваг, недоліків та областей застосування. Об’єктом дослідження виступають відомі декларативні UI-фреймворки, які використовуються для створення мобільних додатків.У роботі детально аналізуються та порівнюються різні інструменти, які дозволяють розробникам створювати користувацький інтерфейс мобільних додатків.Робота включає у себе детальний аналіз функціональних можливостей, ефективності та прикладів використання кожного з розглянутих декларативних UI-фреймворків. У роботі було розглянуто декларативні UI фреймворки, які використовуються для розробки мобільних програм. Вони були порівняні між собою технічно на прикладі коду під різні завдання та ситуації, а також були виявлені їхні сильні та слабкі сторони.

Робота допомагає визначитися з фреймворком, позначає, під яку ситуацію кожен з них є доречним, а так само були знайдені найкращі фрейворки для початківців із найширшими бібліотеками, кращою швидкістю вивчення та простотою розуміння.

ABSTRACT

This thesis is devoted to the comparative analysis of declarative UI frameworks in mobile development. Given the rapid development of mobile technologies and the significant importance of the user interface to the success of mobile applications, the study of various declarative UI frameworks is of great practical importance. The purpose of the work is a detailed study and comparison of various declarative UI frameworks in order to clarify their advantages, disadvantages and areas of application. Well-known declarative UI frameworks, which are used to create mobile applications, are the object of research. The paper analyzes and compares in detail various tools that allow developers to create user interfaces for mobile applications. The work includes a detailed analysis of the functionality, effectiveness and examples of use of each of the considered declarative UI frameworks. The work considered declarative UI frameworks used for the development of mobile applications. They were compared with each other technically on the example of code for different tasks and situations, and their strengths and weaknesses were also revealed.

The work helps to decide on a framework, indicates for which situation each of them is suitable or not, and also found the best frameworks for beginners with the widest libraries, the best speed of learning and ease of understanding.

ЗМІСТ

ВСТУП

Актуальність обраної теми. Конкуренція на ринку мобільних додатків змушує розробників шукати оптимальні технології для реалізації користувацького інтерфейсу, який би забезпечив високу продуктивність, ефективність та зручність в управлінні. Декларативні UI-фреймворки стають все більш популярними, оскільки дозволяють розробникам працювати на вищому рівні абстракції, спрощуючи процес розробки та підвищуючи швидкість впровадження нового функціоналу. Порівняльний аналіз функціональності, синтаксису та реалізації декларативних UI-фреймворків є важливим для визначення їх переваг та недоліків у контексті конкретних вимог проекту. Враховуючи швидкозростаючу кількість фреймворків та інструментів у цій області, такий аналіз дозволить вибрати найбільш оптимальний та підходящий для конкретного проєкту фреймворк, що, в свою чергу, позитивно вплине на якість та ефективність розробки мобільних додатків.

Об'єкт дослідження. Об'єктом дослідження виступають фреймворки, які забезпечують декларативну розробку користувацького інтерфейсу для мобільних додатків, зокрема, Jetpack Compose, Flutter, SwiftUI, React Native, тощо.

Предмет дослідження. Предметом дослідження є функціональні та технічні можливості декларативного опису інтерфейсу користувача, основні сценарії використання та порівняння їх реалізації засобами сучасних декларативних UI-фреймворків.

Мета дослідження. Метою дослідження є виконання систематичного огляду та аналізу функціональних можливостей і технічної реалізації сучасних декларативних UI-фреймворків для розробки мобільних додатків з формуванням цільових рекомендацій щодо їх застосування.

Завдання дослідження. Для досягнення мети необхідно виконати такі завдання:

1) оцінювання функціональних можливостей різних декларативних UI-

фреймворків у мобільній розробці;

2) аналіз синтаксису кожного фреймворку, порівняння його зручності та читабельності;

3) вивчення технічних особливостей реалізації декларативного підходу в мобільних додатках за допомогою різних фреймворків;

4) визначення переваг і недоліків кожного фреймворку з кута зору його застосовності та відповідності потребам розробників;

5) вироблення стратегії прийняття управлінських рішень щодо вибору найбільш доречного декларативного UI-фреймворку для конкретного проєкту мобільної розробки.

РОЗДІЛ 1
ОГЛЯД ДЕКЛАРАТИВНИХ UI-ФРЕЙМВОРКІВ

1.1. Основи UI-фреймворків

Важливою складовою успішного мобільного додатку є його користувацький інтерфейс, який повинен бути не лише зручним, а й ефективним у використанні. Одним із ключових аспектів створення сучасних мобільних інтерфейсів є вибір доречного UI-фреймворку. Декларативні UI-фреймворки здобувають все більшу популярність, оскільки вони дають змогу розробникам зосередитися на описі вигляду та поведінки елементів інтерфейсу, мінімізуючи складність роботи з DOM-структурою та маніпуляціями з низькорівневими API.

UI-фреймворки – це набір інструментів, бібліотек і компонентів, які допомагають розробникам будувати інтерфейси користувача (UI) для мобільних додатків. Основна мета цих фреймворків полягає в спрощенні процесу створення UI, забезпеченні консистентного вигляду та поведінки додатків на різних пристроях та платформах.

UI-фреймворки зазвичай надають широкий набір готових компонентів і віджетів, таких як кнопки, тексти, списки, картки тощо. Ці компоненти можна використовувати для швидкої побудови UI без необхідності писати кожен елемент з нуля. Фреймворки надають засоби для визначення стилів, кольорів, шрифтів та інших атрибутів візуального оформлення UI. Це дозволяє забезпечити консистентний вигляд додатків та легко змінювати їхній вигляд відповідно до дизайну. UI-фреймворки надають зручні інструменти для розміщення елементів на екрані, такі як контейнери, сітки, флекси тощо. Це допомагає організувати інтерфейс таким чином, щоб він був зручним для користувача та легко адаптувався до різних розмірів екранів. Багато фреймворків надають підтримку для реактивного програмування і анімацій. Це дозволяє створювати інтерактивні елементи, які реагують на дії користувача, а також додавати анімації для покращення візуального досвіду. Деякі UI-фреймворки, такі як ReactNative та Flutter, підтримують розробку крос-платформних додатків, тобто додатки, які працюють однаково на різних платформах, таких як iOS та

Android. Це дозволяє економити час і зусилля розробників. Деякі фреймворки надають можливості для легкої інтеграції з іншими сервісами, такими як бази даних, API, аналітика тощо. Це дозволяє розширювати функціональність додатків та підтримувати їхню взаємодію з іншими системами.

Загалом, UI-фреймворки в мобільній розробці спрощують процес створення та управління інтерфейсами користувача, дозволяючи розробникам зосередитися на функціональності та дизайні додатків без необхідності повторюваного написання коду для кожного елементу UI.

1.2. Переваги декларативних UI-фреймворків

Декларативні UI-фреймворки набувають все більшої популярності серед розробників мобільних додатків завдяки численним перевагам, які вони пропонують. Ці фреймворки дозволяють значно спростити розробку інтерфейсів та забезпечують ефективність у роботі з кодом.

Наприклад, за даними розробників з усього світу від сайту Statista [12] щодо популярних вебфреймворків, на момент 2023 року лідируючими є Node.js, React і jQuery. Однак варто також звернути увагу на популярні мобільні та кросплатформні UI-фреймворки, які активно використовуються для створення мобільних додатків (рис. 1.1):

– Flutter швидко набирає популярність завдяки високій продуктивності та підтримці Google. Він дає змогу створювати красиві та високопродуктивні програми для iOS та Android;

– NativeScript підтримується компанією Progress і постачає розробникам можливості використовувати нативні API платформи без написання нативного коду. NativeScript використовується такими компаніями, як SAP та SmartThings;

– Jetpack Compose – новий, але швидко зростаючий фреймворк для створення нативних Android-додатків із використанням декларативного підходу. Розроблено та підтримується Google; [7]

– Vue Native використовується розробниками, знайомими з Vue.js, і надає можливість створювати кросплатформні програми для iOS та Android.

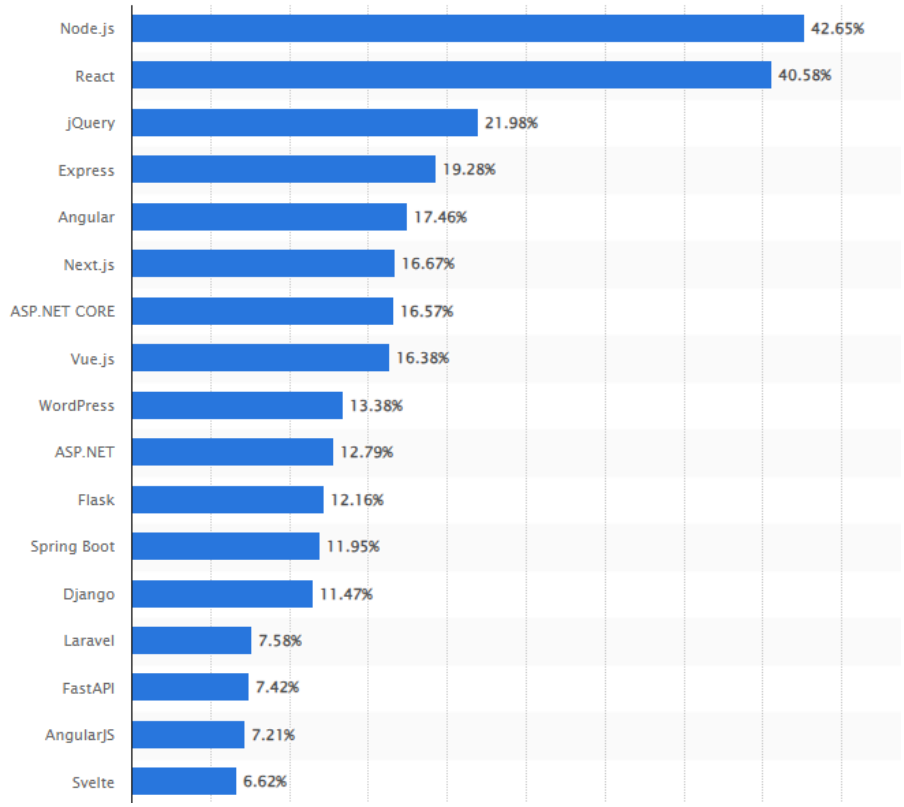

Рисунок 1.1 – Найпопулярніші вебфреймворки за даними Statista

Ці дані показують, що мобільні та кросплатформні UI-фреймворки стають все більш популярними серед розробників, забезпечуючи потужні та гнучкі інструменти для створення сучасних мобільних додатків. Нижче розглянемо детально основні переваги декларативних UI-фреймворків.

Однією з головних переваг декларативних UI-фреймворків є їх простота та зручність у використанні. Замість того, щоб вказувати конкретні кроки для створення елементів інтерфейсу (імперативний підхід), розробники описують, як інтерфейс повинен виглядати та реагувати на зміни стану (декларативний підхід). Це дає змогу зосередитися на описі кінцевого результату, а не на процесі досягнення цього результату.

Декларативні UI-фреймворки допомагають значно зменшити кількість коду, необхідного для створення інтерфейсів. Це робить код більш компактним, зрозумілим і легким для підтримки.

Багато декларативних UI-фреймворків оптимізовані для високої продуктивності. Вони використовують різні техніки для мінімізації кількості необхідних оновлень інтерфейсу та ефективного управління станом. Це дає змогу створювати додатки, які працюють швидко та плавно, забезпечуючи відмінний користувацький досвід.

Декларативні UI-фреймворки часто підтримують реактивність, що дозволяє автоматично оновлювати інтерфейс у відповідь на зміни в стані даних. Це забезпечує динамічність та інтерактивність додатків.

1.3. Огляд сучасних UI-фреймворків

React Native – це фреймворк для розробки мобільних додатків, що базується на бібліотеці React, розробленій Facebook. Історія появи React Native пов’язана з історією React та потребою в уніфікованому інструменті для створення крос-платформенних мобільних додатків. У 2013 році, після успіху React в галузі веброзробки, компанія Facebook почала досліджувати можливість використання React для мобільних платформ. Вони розробили прототип під назвою ReactNativeforiOS, який надавав можливості писати мобільні додатки для iOS за допомогою JavaScript та React. У березні 2015 року Facebook анонсував випуск React Native на конференції F8. Цей фреймворк дозволяв розробникам створювати кросплатформні мобільні додатки для iOS та Android за допомогою React та JavaScript. React Native має унікальні особливості, такі як можливість перевикористання коду між платформами, високу продуктивність завдяки використанню нативних компонентів та можливість гарячого перезавантаження (Hot Reload) для швидкого тестування змін у реальному часі. [10]

React Native відрізняється від інших фреймворків для мобільної розробки декількома ключовими особливостями. Однією з основних переваг React Native є можливість розробки кросплатформних додатків для iOS та Android з використанням одного коду. Він застосовує нативні компоненти, що дозволяє додаткам мати природний вигляд та взаємодіювати з платформами безпосередньо. Також ReactNative використовує декларативний підхід до

побудови UI, що дає змогу розробникам описувати, як додаток повинен виглядати, на основі стану додатку та вхідних даних. Одна з найпопулярніших функцій ReactNative – гаряче перезавантаження, яке допомагає розробникам швидко бачити зміни в коді UI без необхідності повного перезавантаження додатка. Це полегшує тестування та відладку додатків. Вигляд середовища розробки з кодом та робочим емулятором продемонстровано на рис. 1.2.

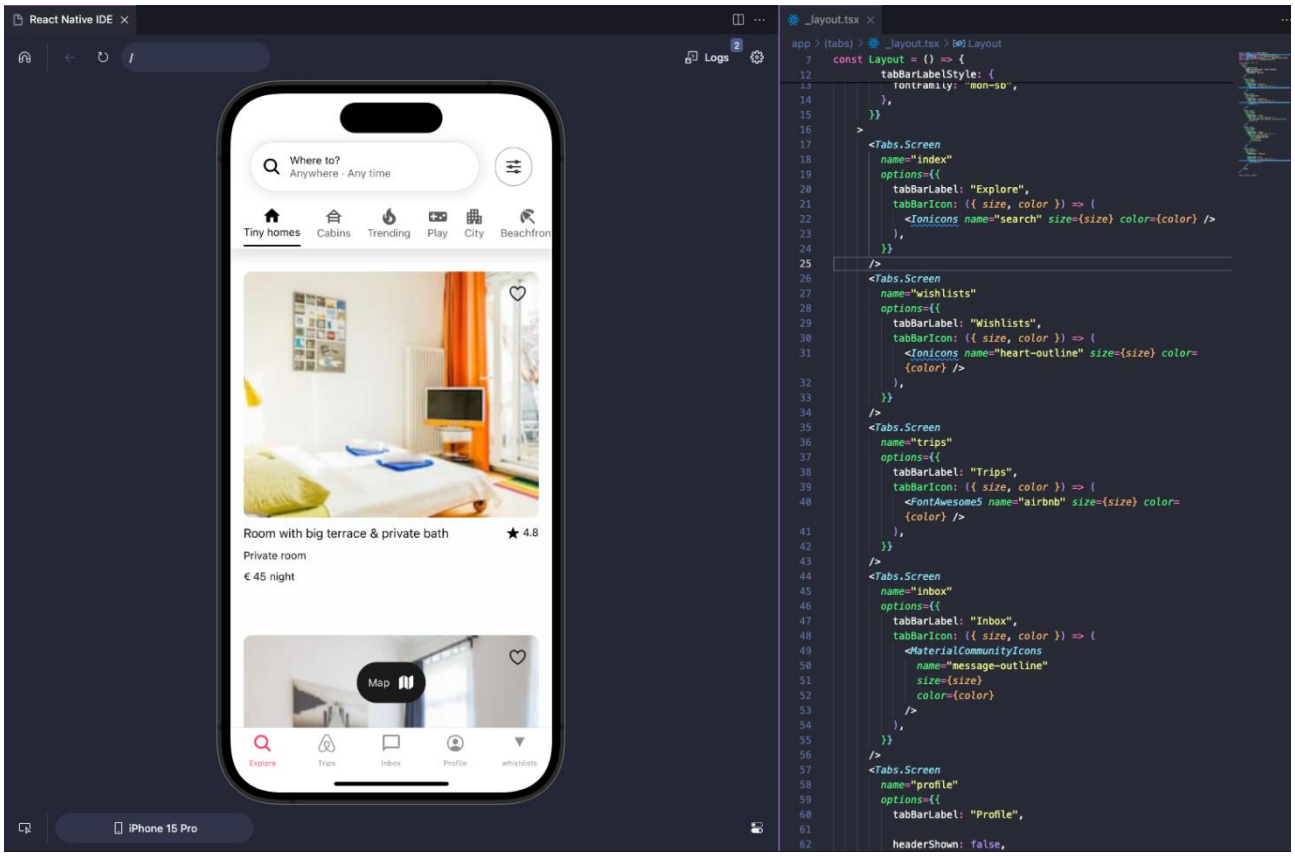

Рисунок 1.2 – Розроблення додатків засобами React Native

У React Native наявна велика кількість готових компонентів та віджетів, які розробники можуть використовувати для швидкої побудови інтерфейсів. Сюди відносять кнопки, тексти, списки, каруселі, карти та багато іншого.

React Native має велике та активне співтовариство розробників, що призводить до швидкого вирішення проблем, надання порад та підтримки. Крім того, існує велика кількість сторонніх бібліотек, плагінів та інструментів для підтримки та розширення функціональності додатків на ReactNative. Фреймворк підтримує використання сторонніх модулів та бібліотек, що сприяє інтеграції

додатків з різними сервісами, такими як бази даних, мережеві запити, аналітика тощо.

У цілому, React Native є потужним інструментом для розробки мобільних додатків, який відзначається високою продуктивністю, швидкістю розробки та можливістю створення кросплатформних додатків з виглядом та функціональністю нативних додатків.

Фреймворк Flutter – це відкритий фреймворк для розробки кросплатформних мобільних додатків від компанії Google. Основою Flutter є мова програмування Dart, яка володіє сучасним синтаксисом, схожим на Java або JavaScript. Мова Dart підтримує асинхронне програмування, що сприяє швидкій роботі додатків. [11]

Flutter дає розробникам змогу створювати кросплатформні додатки, які працюють однаково на iOS та Android. Він використовує один код для створення інтерфейсу користувача, логіки та управління станом додатка.

У Flutter є велика кількість вбудованих віджетів для побудови інтерфейсу користувача. Ці віджети включають в себе кнопки, тексти, списки, анімації, карти, форми та багато іншого. Розробник може легко налаштовувати та перевикористовувати ці віджети. Подібно до ReactNative, Flutter використовує декларативний підхід до побудови інтерфейсу користувача. Розробники описують, як повинен виглядати їхній UI за допомогою віджетів та композицій. Однією з найбільших переваг Flutter, як и у ReactNative є гарячий перезавантаження, яке дозволяє розробникам бачити зміни у реальному часі без необхідності повного перезавантаження додатка. Це полегшує процес розробки та відладки. Flutter відзначається високою продуктивністю та швидкістю роботи. Він використовує технологію «Skia» для відрисовки користувацького інтерфейсу, що сприяє швидкій та плавній роботі інтерфейсу додатків.

Flutter має широке співтовариство розробників та активну підтримку від Google. Це означає, що розробники можуть легко знаходити відповіді на свої питання, використовувати сторонні плагіни та бібліотеки.

Власне технологія Flutter спроєктована для створення мобільних додатків з використанням візуальних ефектів та анімацій. Мова Dart була розроблена з орієнтацією на веброзробку, але згодом знайшла застосування і в інших сферах, включаючи мобільну розробку. У березні 2021 року Google випустив Flutter 2.0, який підтримує створення кросплатформних додатків для різних платформ, включаючи веб, десктопні ОС та інші. Зараз Flutter вважається одним з найбільш привабливих фреймворків для розробки мобільних додатків завдяки його зручності використання, ефективності та можливостям створювати красиві та функціональні додатки для різних платформ. Вигляд середовища розробки з кодом та працюючим емулятором показано на рис. 1.3.

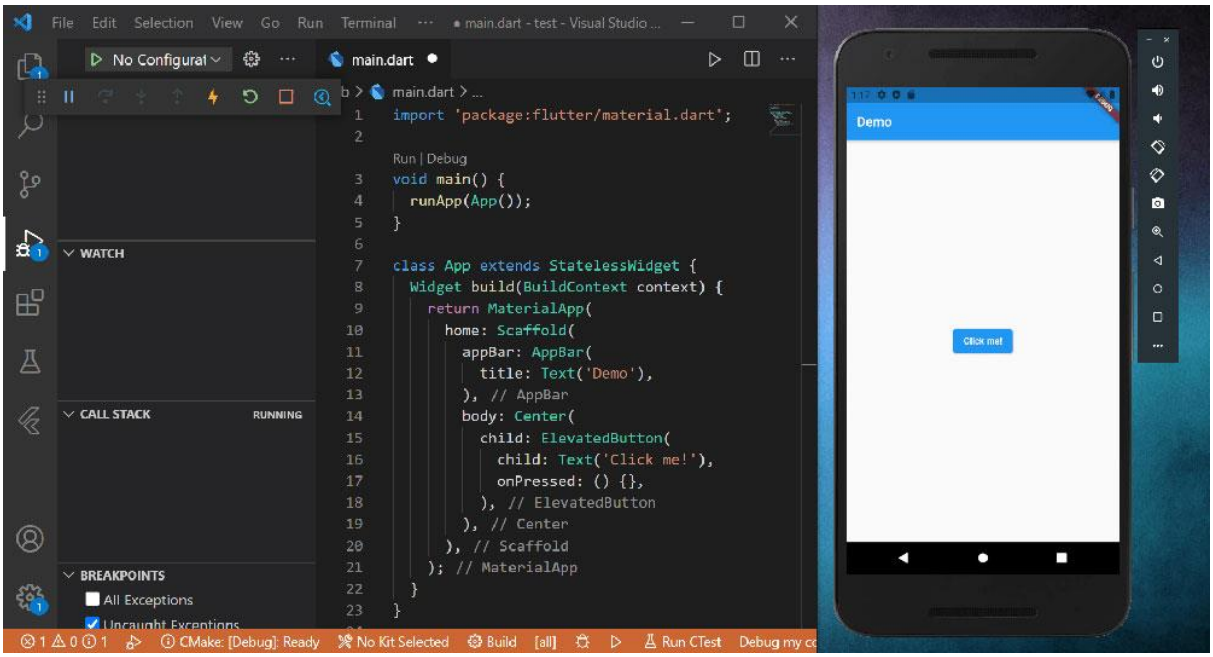

Рисунок 1.3 – Розроблення додатків засобами Flutter

NativeScript – це відкритий фреймворк для розробки кросплатформних мобільних додатків з використанням нативних технологій. NativeScript дозволяє розробникам використовувати відомі мови програмування, такі як JavaScript, TypeScript, Angular та Vue.js. Це дає можливість вибрати найбільш зручний інструмент для кожного конкретного проєкту. Однією з головних переваг NativeScript є можливість використання нативних API та функцій платформ (iOS та Android) безпосередньо з JavaScript. Це дає змогу створювати додатки з відчуттям нативного досвіду користувача. NativeScript дозволяє розробникам

створювати кросплатформні додатки для iOS та Android з використанням спільного коду. Це зменшує зусилля та час, потрібний для розробки та підтримки додатків на різних платформах.

NativeScript постачає для розробників функціональність, яка включає розширені можливості, такі як доступ до камери, геолокації, push-сповіщень, датчиків та інших апаратних функцій пристроїв безпосередньо з JavaScript. Фреймворк підтримує спільні підходи до створення інтерфейсу користувача через використання XML та CSS для опису UI. Це дозволяє розробникам зосередитися на дизайні та взаємодії замість деталей реалізації. Маючи інтеграцію з Angular та Vue.js, фреймворк дозволяє використовувати їх для розробки кросплатформних додатків з нативними можливостями. Вигляд середовища розробки з кодом користувацького інтерфейсу показано на рис. 1.4.

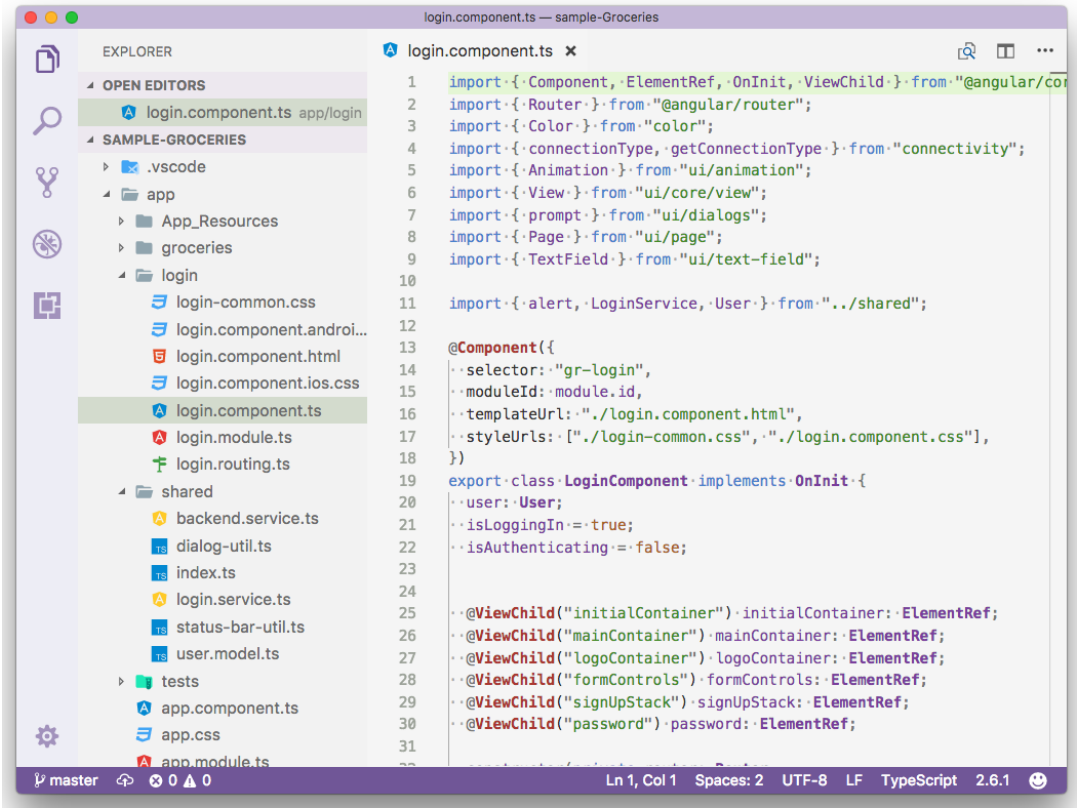

Рисунок 1.4 – Розроблення додатків засобами NativeScript

SwiftUI – це декларативний фреймворк для розробки інтерфейсів користувача (UI) для програм, написаних мовою програмування Swift, яка використовується для розробки додатків для платформ iOS, macOS, watchOS та tvOS. SwiftUI використовує декларативний підхід до створення інтерфейсу

користувача, що означає, що ви описуєте, як повинен виглядати ваш інтерфейс, а не як його створювати крок за кроком. Це забезпечує більшу зрозумілість та зручність в розробці.

SwiftUI повністю інтегрований з мовою програмування Swift, що дає змогу використовувати всі її можливості для створення інтерфейсу [2]. Фреймворк надає велику кількість вбудованих віджетів та компонентів, таких як кнопки, тексти, списки, форми, зображення та багато інших. Ці компоненти легко налаштовувати та комбінувати для створення складних інтерфейсів (рис. 1.5).

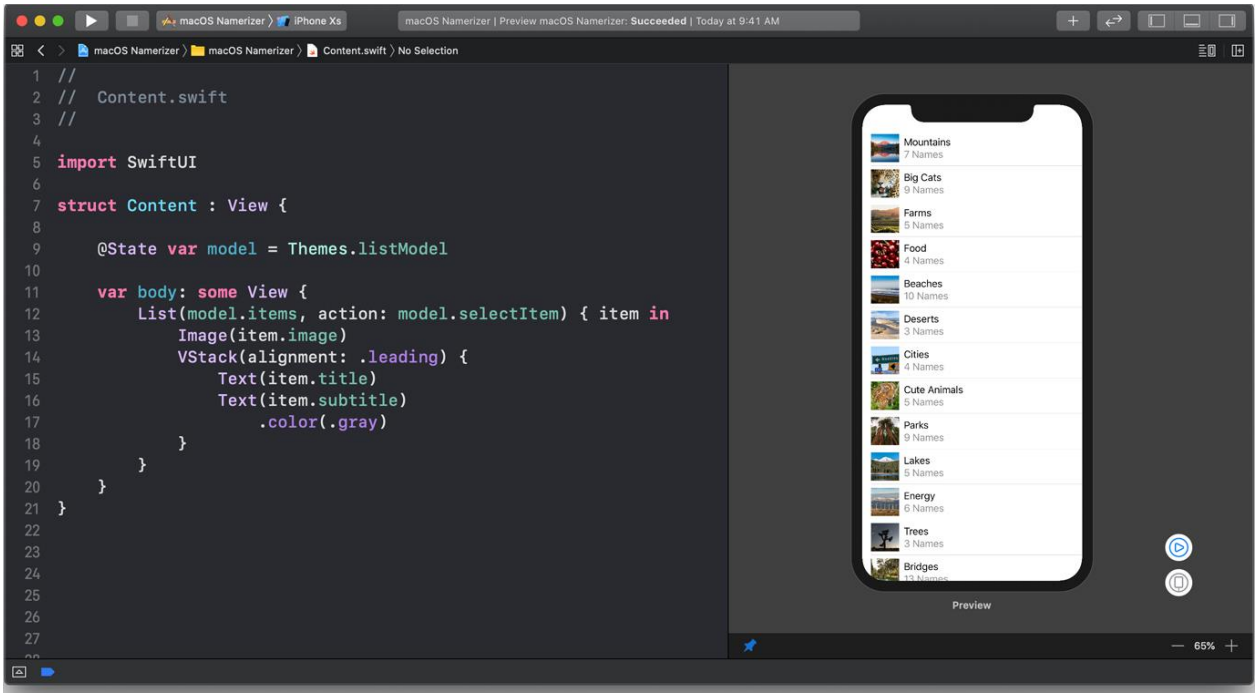

Рисунок 1.5 – Розроблення додатків засобами SwiftUI

SwiftUI побудований на принципах реактивного програмування, що дає змогу автоматично оновлювати інтерфейс користувача відповідно до змін в даних. Це полегшує роботу зі змінними станами додатка та дозволяє швидко реагувати на події. SwiftUI підтримує розробку для різних платформ, таких як iOS, macOS, watchOS та tvOS. Звідси, розробники можуть використовувати один код для створення інтерфейсу для різних пристроїв. SwiftUI може взаємодіяти з іншими фреймворками, такими як UIKit для iOS та AppKit для macOS. Це дозволяє поєднувати нові можливості SwiftUI з існуючими додатками та бібліотеками.

Vue Native – це фреймворк для розробки кросплатформних мобільних додатків з використанням Vue.js та JavaScript. Vue Native побудований на компонентній архітектурі, що робить розробку мобільних додатків більш організованою та простою (рис. 1.6). Ви можете створювати компоненти, які можна використовувати повторно, підвищуючи ефективність розробки.

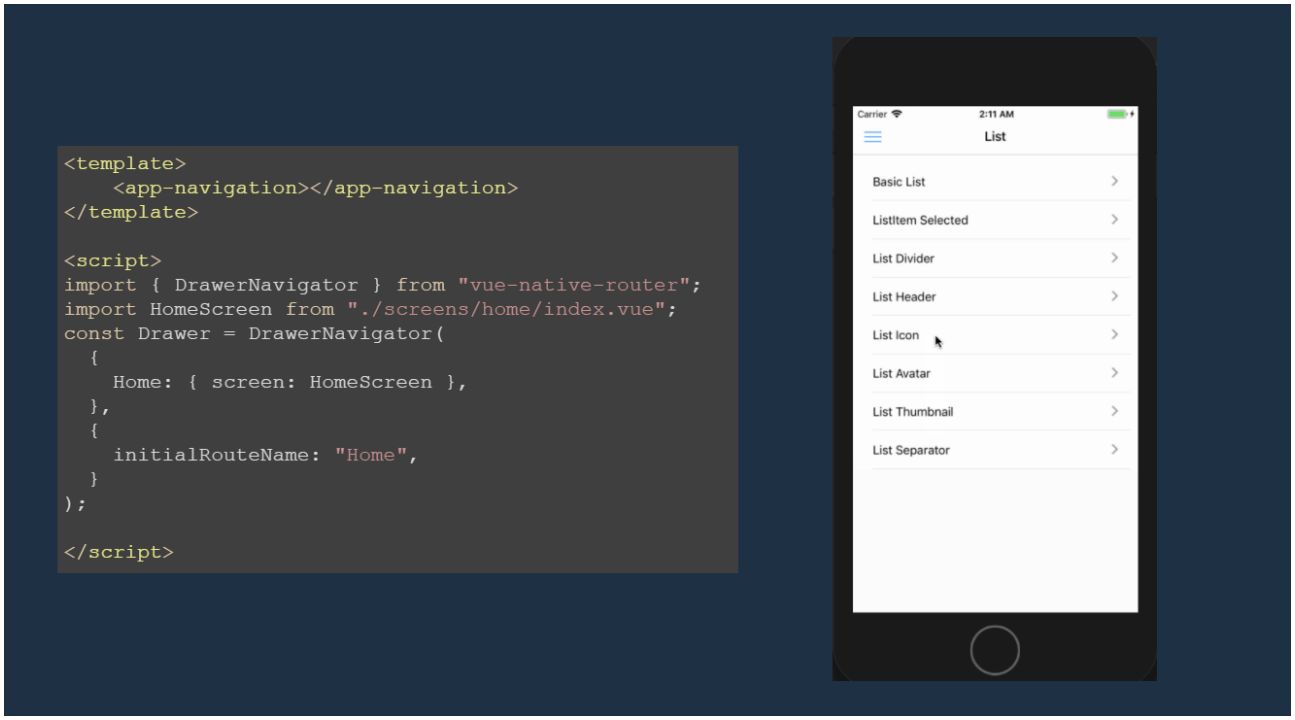

Рисунок 1.6 – Вигляд UI-коду для фреймворку Vue Native

Основною мовою програмування у Vue Native є JavaScript разом з синтаксисом Vue.js. Це робить розробку більш зрозумілою для тих, хто вже працював з Vue.js або знає JavaScript.Vue Native [5] використовує реактивність для автоматичного оновлення інтерфейсу користувача при зміні стану даних, що дає змогу створювати динамічні та респонсивні додатки. Vue Native дозволяє розробляти кросплатформні мобільні додатки для iOS та Android, використовуючи один і той же код. У VueNative існує багато плагінів, які додають нові функціональні можливості та інтеграції з іншими сервісами. Це дозволяє розширювати можливості додатка та прискорювати розробку.Є активна спільнота розробників Vue Native, яка активно обмінюється досвідом та надає підтримку новачкам та досвідченим розробникам. Vue Native надає добру

швидкість роботи додатків та продуктивність розробки завдяки своїй легкості в освоєнні та використанні.

JetpackCompose – це декларативний UI-фреймворк для розробки Android-додатків, розроблений компанією Google. Jetpack Compose використовує декларативний підхід до розробки інтерфейсу користувача. Замість імперативного кодування UI-елементів через XML, ви описуєте, як UI повинен виглядати відповідно до стану даних та логіки за допомогою Kotlin (рис. 1.7).

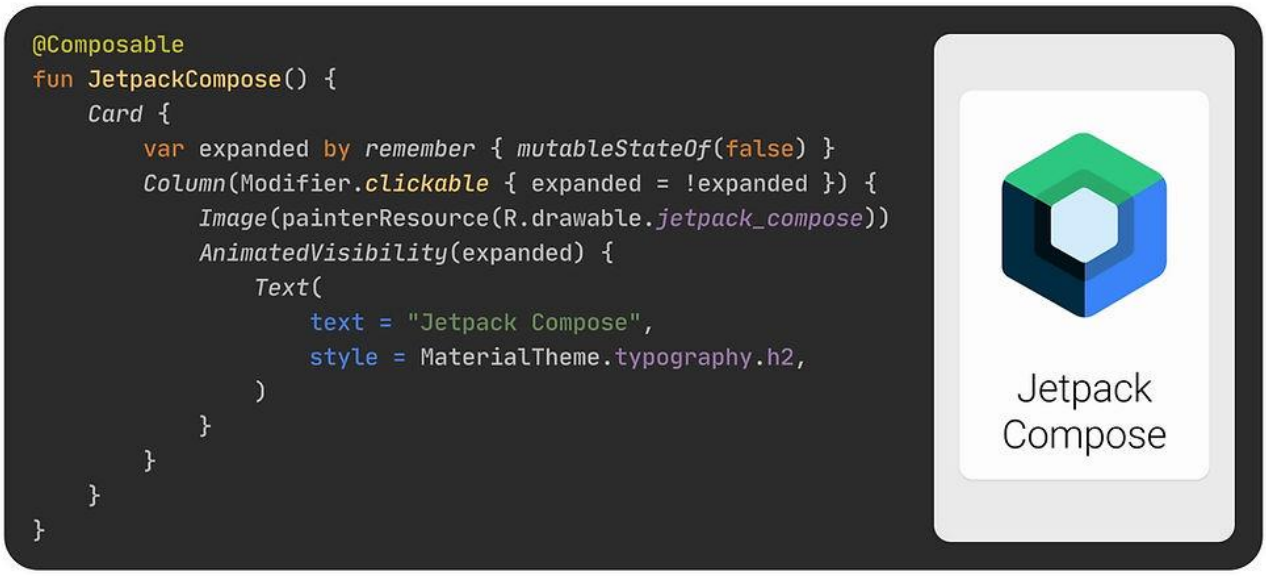

Рисунок 1.7 – Опис інтерфейсу звасобами Jetpack Compose

Jetpack Compose базується на мові програмування Kotlin. Він використовує Kotlin DSL (Domain Specific Language), що дозволяє вам описувати UI зрозумілим та зручним способом, використовуючи функції та комбінації елементів. У Jetpack Compose є багатий набір готових компонентів інтерфейсу користувача, таких як кнопки, тексти, списки, карти тощо. Ви також можете створювати власні компоненти та компонувати їх для створення складних UI-структур. Jetpack Compose підтримує реактивний підхід, де UI оновлюється автоматично при зміні стану даних. Фреймворк має вбудовану підтримку Material Design, що дозволяє легко створювати додатки з сучасним та стильним дизайном за допомогою готових матеріальних компонентів.

JetpackCompose інтегрується з існуючими Android-проєктами, що дає змогу поступово переходити на новий UI-фреймворк, не пошкоджуючи

існуючий код. Фреймворк має активну спільноту розробників, що сприяє обміну досвідом, вирішенню проблем та розвитку фреймворку. Крім того, він отримує регулярні оновлення та підтримку від Google.

Отже, за результатами огляду предметної області було встановлено, що існує ціла низка UI-фреймворків, які надають аналогічну функціональність та подібні інструментальні засоби. Проте зручність та ефективність розробки користувацьких інтерфейсів також залежить від архітектурних підходів, закладених при проєктування фреймворку, принципах побудови синтаксису та інших аспектах. Це зумовлює необхідність подальшого, більш глибокого аналізу.

РОЗДІЛ 2

ТЕХНІЧНИЙ АНАЛІЗ ДЕКЛАРАТИВНИХ UI-ФРЕЙМВОРКІВ

2.1. Аналіз архітектури UI-фреймворку React Native

Компонентами архітектури фреймворку React Native є:

– генератор нативного коду Codegen;

– JSI (JavaScript Interface)

– JavaScript-рушій Hermes Engine, який виконує JS-код на пристроях;

– Turbo Module (реалізує нативний модуль за допомогою JSI та нативного коду);

– рендер-рушій Fabric;

– формувальник рендер-пайплайну Fabric renderer;

– рушій Yoga.

Архітектура React Native будує свій робочий цикл на двох фазах: етапі збірки додатку та етапі виконання додатку. На першому етапі відбуваються такі кроки:

1) команда на збирання компілює JavaScript-код у байткод;

2) під час збирання створюється нативний код за допомогою Codegen;

3) байткод та нативний код включаються до пакету додатку, який може бути встановлений на пристрій.

Codegen працює лише під час збірки додатку, але не під час його виконання. Він виконує дві основні задачі у новій архітектурі React Native: перевірку типів на етапі компіляції та генерацію нативного коду. Codegen забезпечує перевірку типів між JavaScript та C++, що вирішує проблеми комунікації між динамічно типізованим JavaScript та статично типізованим C++. Також він генерує нативний код для компонентів Turbo Module та Fabric, що полегшує комунікацію між JavaScript та нативним кодом без використання мосту, що робить цей процес швидшим та надійнішим.

JSI є найглибшою частиною нової архітектури React Native та використовується всіма компонентами нової архітектури: Hermes Engine, Turbo Module, Fabric. Цей інтерфейс написаний на C++ та замінює міст зі старої

архітектури, надаючи прямий нативний інтерфейс до JavaScript-об’єктів та функцій. Отримуємо такі переваги:

– паралелізм. JavaScript може викликати функції, що виконуються на різних потоках;

– синхронне виконання: можливість виконувати синхронно ті функції, які не повинні бути асинхронними;

– зниження накладних витрат: нова архітектура не потребує серіалізації або десеріалізації даних;

– спільний код: використання C++ дає змогу розробляти платформо-незалежний код, який легко шериться між платформами;

– типобезпечність: для забезпечення правильного виклику методів JavaScript на C++ об’єктах додається прошарок коду, який генерується на основі специфікацій JavaScript, типізованих за допомогою Flow або TypeScript;

– JSI є легковаговим універсальним JavaScript інтерфейсом, написаним на C++, який може бути використаний JavaScript-рушієм для прямого виклику методів у нативному середовищі;

– відокремлення (decoupling) інтерфейсу від JavaScript-рушія: стара архітектура використовувала JavaScriptCore (JSC) Engine, проте нова дає змогу використовувати інші JavaScript-рушії, такі як Chakra, v8 та Hermes.

Hermes є JavaScript-рушієм, що виконує код на пристроях, коли користувач запускає додаток. Він завантажує байткод, що містить JavaScript-код та нативний код додатку, та використовує JSI для прямого доступу до нативних функцій та об’єктів, без використання мосту, що забезпечує ефективну та швидку роботу додатку. Hermes не лише корисний для додатків React Native, але й зменшує розмір пакету та час завантаження, а також надає інтерфейс для візуалізації метрик продуктивності вашого додатку під час розробки.

За результатами дослідження, проведеного супроводжувачами React Native [3], Hermes є найпродуктивнішим двигуном JavaScript для побудови додатків React Native. У дослідженні було оцінено три метрики:

– TTI – тривалість між запуском додатку та моментом, коли користувачі можуть взаємодіяти з ним;

– розмір бінарного файлу – це розмір пакету додатку React Native у форматі APK (Android) або IPA (iOS);

– споживання пам’яті – обсяг пам’яті, що використовується під час роботи додатку.

Turbo Modules – це система нативних модулів, яка використовує технологію JSI (JavaScript Interface), написану на C++, замість серіалізації даних JSON. Нативні модулі – це частини нативного коду, написані мовою нативної платформи, такі як Java для Android або Objective-C для iOS, які можуть бути викликані з JavaScript. Система нативних модулів – це система, за допомогою якої JavaScript та нативні модулі взаємодіють один з одним. Вона дозволяє JavaScript-коду використовувати нативні функції та бібліотеки, які недоступні в JavaScript, такі як робота з камерами, датчиками або шифрування.

Turbo Modules впроваджує «ліниве завантаження» для нативних модулів (наприклад, Bluetooth, геолокація та файлове сховище), що дає змогу завантажувати їх тільки при запуску додатку користувачем. Система дозволяє JavaScript-коду утримувати посилання на ці нативні модулі, тому конкретний нативний модуль завантажується тільки тоді, коли він потрібен, що покращує час запуску додатку.

Fabric – це UI-менеджер, відповідальний за рендеринг UI на пристроях. Замість того, щоб спілкуватися з JavaScript через місток, Fabric надає свої функції через JavaScript, дозволяючи JavaScript і нативному коду спілкуватися безпосередньо через функції-посилання.

Fabric використовує JSI для взаємодії з Hermes та нативним кодом без використання містка. Він покращує інтероперабельність між React Native та платформами-хостами (наприклад, Android або iOS), поліпшуючи комунікацію між JavaScript та нативними потоками. Це забезпечується через дерево представлення UI-елементів на платформі-хості, що дозволяє React Native рендерити поверхні синхронно.

Завдяки інтеграції React Suspense завантаження даних у додатках стало простішим. Інші функції, доступні в React 18, тепер доступні в Fabric Renderer, такі як Concurrent Features, що підтримують UI додатків респонсивним під час тривалих переходів станів.

Fabric Renderer – це нова система рендерингу React Native, яка об'єднує більше логіки рендерингу в C++, покращує інтероперабельність з платформами-хостами та відкриває нові можливості для React Native. Рендер-конвеєр Fabric проходить у три фази, як показано на рис. 2.1: фаза рендерингу, фаза фіксації та фаза монтування.

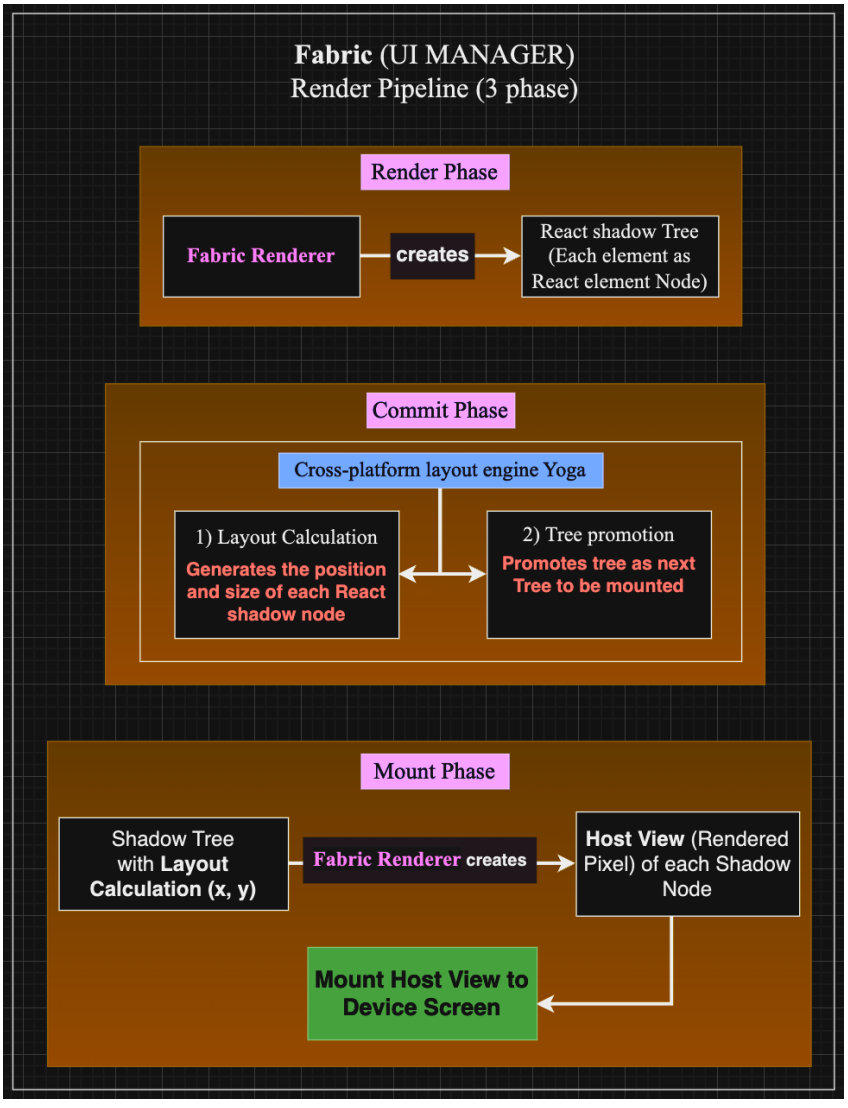

Рисунок.2.1 – Рендер-конвеєр Fabric

Протягом фази рендерингу React виконує код для створення дерев React-елементів – простих JavaScript-об'єктів з описом того, що відображається на

екрані. Дерево React-елементів використовується для рендерингу тіні дерева React в C++. Рендерер Fabric створює тіньове дерево, яке складається з тіньових React-вузлів, що представляють компоненти для монтування (рис. 2.2). Ці вузли створюються синхронно лише для хост-компонентів React, але не для композитних компонентів, таких як <View>. Коли <View> перетворюється у Shadow Block, він транслюється в об'єкт <ViewShadowNode>. Процес на рис. 2.2 показує, як збирається тіньове React-дерево; коли воно завершене, рендерер ініціює фіксацію (коміт) дерева елементів.

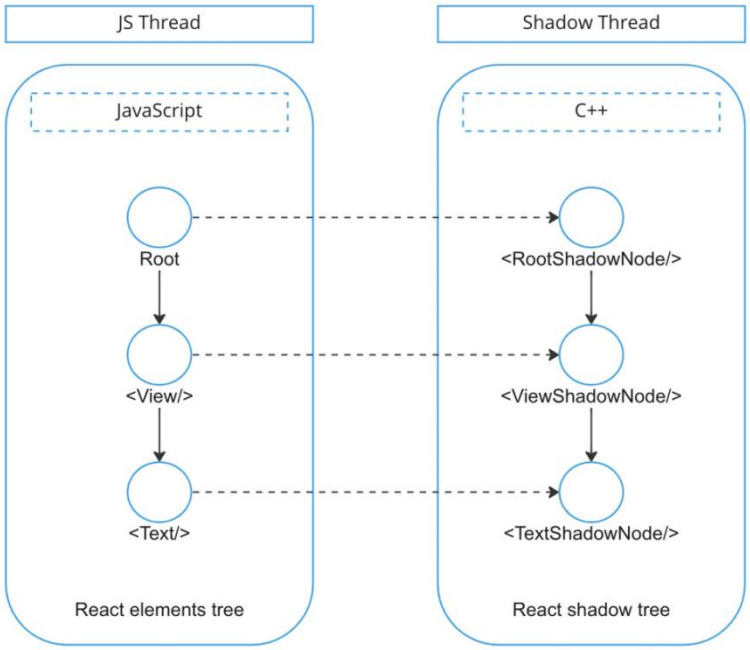

Рисунок 2.2 – Представлення фази рендерингу

Механізм кросплатформної компонування Yoga виконує операції, які відбуваються під час фази фіксації: обчислення макету і підвищення дерева (tree promotion). Обчислення макету за допомогою Yoga генерує положення та розмір кожного тіньового вузла React. Операція Tree Promotion просуває нове тіньове React-дерево як наступне дерево для монтування. Це представляє останній стан дерева елементів React.

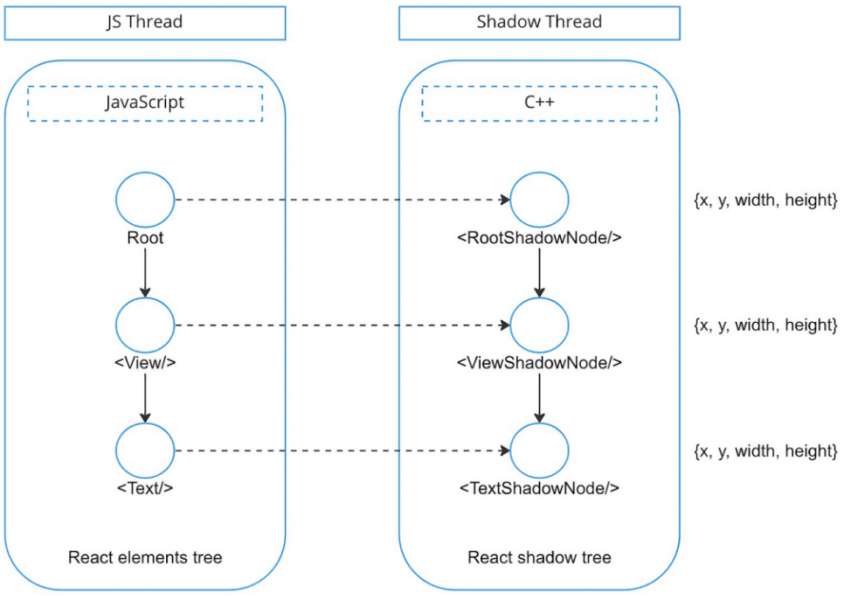

Рисунок 2.3 – Дерево React-елементів

Фаза монтування, на якій тіньове React-дерево перетворюється на дерево перегляду хоста (пристрою, з якого було запущено додаток) із відтвореними пікселями на екрані. Fabric Renderer створює відповідне представлення хоста для кожного тіньового вузла React і монтує його на екрані. Засіб візуалізації Fabric відповідає за створення представлень хоста та їх монтування на екрані.

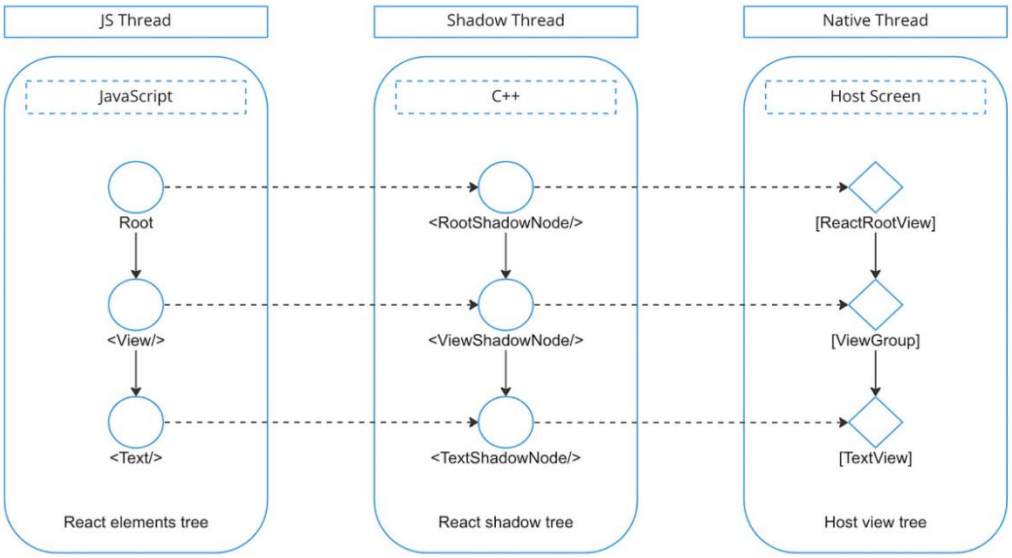

Рисунок 2.4 - Фаза монтування

Узагальнений процес відображено на рис. 2.5. Додаток починає свою роботу, завантажуючи файл байткоду з кодом JavaScript, який обробляє рушій

Hermes на Android та iOS для поліпшення розміру додатка та швидкості запуску. Далі код JavaScript спілкується з нативним кодом через JSI. Нативний код створює нативні модулі з використанням Turbo Modules. Код JavaScript також визначає інтерфейс через React-компоненти, які перетворюються в нативний код за допомогою Fabric, підвищуючи швидкість реагування та забезпечуючи кращу інтеграцію з функціями пристрою. Fabric використовує Yoga для dbpyfxtyyz розташування елементів інтерфейсу та Fabric Renderer для їх відображення на екрані, забезпечуючи оптимальну респонсивність та зручність користування.

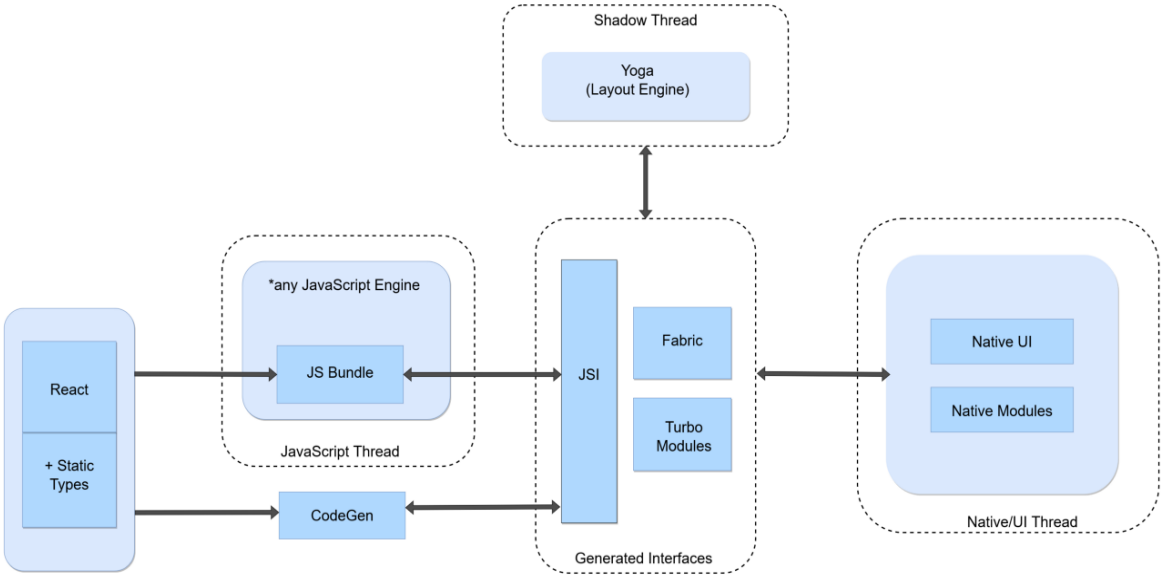

Рисунок 2.5 – Загальні інтерфейси

Нарешті, Fabric використовує Fabric Renderer для відображення елементів інтерфейсу користувача на екрані, застосовуючи нативні графічні API, такі як Core Animation на iOS або SurfaceView на Android. Fabric Renderer обробляє анімації, жести, доступність та інші функції, які роблять елементи інтерфейсу користувача інтерактивними.

2.2. Аналіз архітектури UI-фреймворку: Jetpack Compose

Для Jetpack Compose справедливе рівняння: $UI = f(state)$. Це означає, що інтерфейс користувача є результатом функції, застосованої до певного стану.

В основі архітектури Compose та Reactive UI загалом лежать важливі аспекти обробки стану: підйом стану та однонаправлений потік даних.

Підйом стану — це техніка, яка використовується в розробці програмного забезпечення, зокрема в програмуванні інтерфейсу користувача, де відповідальність за керування та маніпулювання станом компонента переміщується до компонента вищого рівня або більш централізованого місця. Метою підйому стану є покращення організації коду, можливості повторного використання та обслуговування.

Однонаправлений потік даних (UDF) — це шаблон проєктування, де стан «стікає» вниз, а події — вгору. Дотримуючись однонаправленого потоку даних, ви можете відокремити складники, які відображають стан в інтерфейсі користувача, від частин вашої програми, які зберігають і змінюють стан. Суть полягає в тому, що ми хочемо, щоб наші компоненти інтерфейсу споживали стан і випромінювали події. Те, що наші компоненти обробляють події, що виникли ззовні, порушує це правило, запроваджуючи кілька джерел істини. Важливо те, що будь-яка «подія», яку ми впроваджуємо, має базуватися на стані.

Станом може бути все, що має сенс для розробника і його сценарію використання. Це може бути клас даних, що має всі властивості, які можуть знадобитися вашому інтерфейсу користувача, або закритий інтерфейс, який представляє всі можливі сценарії. У будь-якому випадку стан — це статичне представлення компонента або всього інтерфейсу користувача з екрану.

Отже, екран – це функція f з вище згаданого рівняння. Щоб слідувати шаблону підйому стану, потрібно зробити цей компонент без стану та виставити взаємодії користувача як зворотні виклики. Це зробить екран придатним для перевірки, попереднього перегляду та повторного використання. Ми також включаємо інші стани Composable, які нам можуть знадобитися, тому вони виведені за межі екрана. Уривки коду, які демонструють принцип побудови інтерфейсу як набору Composable-елементів, показано в лістингу 2.1.

Лістинг 2.1 – Зразок опису Composable-елемента

```
data class PlacesState(val places: List<Place> = emptyList(),val error: String? = null)

@Composable
	fun PlacesScreen(
	state: PlacesState,
	pagerState: PagerState,
	onFavoritesButtonClick: (Place) -> Unit,
	onNavigateToPlaceButtonClick: (Place) -> Unit) {
		Scaffold {
		PlacesPager( pagerState = pagerState, state = state,
		onFavoritesButtonClick = onFavoritesButtonClick,
		onNavigateToPlaceButtonClick = onNavigateToPlaceButtonClick
		)
	}
}
```

Кожен введений в додаток елемент повинен бути представлений як подія: натискання, зміни тексту і навіть таймери або інші оновлення [15]. Оскільки ці події змінюють стан інтерфейсу користувача, ViewModel повинен бути тим, хто їх обробляє та оновлює стан інтерфейсу. Прошарок інтерфейсу користувача ніколи не повинен змінювати стан поза обробником подій, оскільки це може призвести до несумісностей та помилок у додатку.

Наприклад, composable-об’єкт, що приймає String і лямбду як параметри, може бути викликаний з багатьох контекстів і є високоповторюваним. Припустимо, що верхня панель додатку завжди відображає текст і має кнопку «назад». Ви можете визначити більш загальний composable-об’єкт MyAppTopAppBar, який отримує текст і обробник кнопки «назад» як параметри (лістинг 2.2).

Використовуючи ViewModel та mutableStateOf, можна також впровадити однонаправлений потік даних у вашому додатку, якщо виконується одна з наступних умов:

– Стан вашого інтерфейсу користувача доступний через спостережувані тримачі стану, такі як StateFlow або LiveData.

– ViewModel обробляє події, що надходять з інтерфейсу користувача або інших шарів вашого додатку, та оновлює тримач стану на основі подій.

Наприклад, при реалізації екрану входу, натискання на кнопку «Увійти» повинно спричинити відображення індикатора прогресу та виклик мережі.

Якщо вхід пройшов успішно, тоді ваш додаток переходить на інший екран; у разі помилки додаток відображає Snackbar. Екран має чотири стани:

– Singed Out: коли користувач ще не увійшов у систему.

– In Process: коли ваш додаток намагається виконати вхід користувача, здійснюючи мережевий виклик.

– Error: коли сталася помилка під час входу.

– Singed In: коли користувач увійшов у систему.

Лістинг 2.2 – composable-об’єкт MyAppTopAppBar

```
@Composable
fun MyAppTopAppBar(topAppBarText: String, onBackPressed: () -> Unit) {
   TopAppBar(
      title = {
         Text(
            text = topAppBarText,
            textAlign = TextAlign.Center,
            modifier = Modifier
               .fillMaxSize()
               .wrapContentSize(Alignment.Center)
         )
      },
      navigationIcon = {
         IconButton(onClick = onBackPressed) {
            Icon(
               Icons.Filled.ArrowBack,
               contentDescription = localizedString
            )
         }
      },
      // ...
   )
}
```

Ви можете змоделювати ці стани як запечатаний клас. ViewModel надає стан як State, встановлює початковий стан та оновлює стан за необхідністю. ViewModel також обробляє подію входу, надаючи метод onSignIn(). Приклад відповідного коду наведено в лістингу 2.3.

Лістинг 2.3 – Власна реалізація ViewModel

```
class MyViewModel : ViewModel() {
   private val _uiState = mutableStateOf<UiState>(UiState.SignedOut)
   val uiState: State<UiState>
      get() = _uiState

   // ...
}
```

На додачу до API mutableStateOf, Compose надає розширення для LiveData, Flow та Observable для реєстрації як слухача та представлення значення як стану (лістинг 2.4).

Лістинг 2.4 – Розширення для LiveData, Flow та Observable

```
class MyViewModel : ViewModel() {
   private val _uiState = MutableLiveData<UiState>(UiState.SignedOut)
   val uiState: LiveData<UiState>
      get() = _uiState
   // ...
}
@Composable
fun MyComposable(viewModel: MyViewModel) {
   val uiState = viewModel.uiState.observeAsState()
   // ...
}
```

Також Jetpack Compose не є єдиним монолітним проєктом, він створений з кількох модулів, які збираються разом, утворюючи повну стекову систему. Розуміння різних модулів, що складають Jetpack Compose, дозволяє вам використовувати відповідний рівень абстракції для створення додатку або бібліотеки та розуміти, коли ви можете «спуститися» на нижчий рівень для більшого контролю або налаштування, а також мінімізувати ваші залежності [16].

Кожен такий прошарок побудований на «нижчих рівнях», комбінуючи функціональність для створення компонентів вищого рівня. Кожен шар будується на публічних API нижчих шарів, щоб перевірити межі модуля і дозволити вам замінити будь-який шар, якщо це необхідно. Розглянемо ці шари знизу вгору (рис. 2.6).

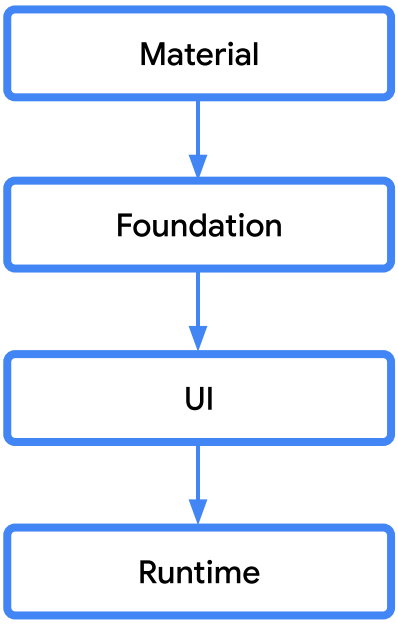

Рисунок 2.6 – Прошарки API Jetpack Compose

Модуль Runtime забезпечує основи виконання Compose, такі як remember, mutableStateOf, анотація @Composable та SideEffect. Ви можете розглянути можливість побудови безпосередньо на цьому шарі, якщо вам потрібні тільки можливості управління деревом Compose, а не його інтерфейс користувача.

Прошарок UI складається з кількох модулів (ui-text, ui-graphics, ui-tooling тощо). Ці модулі реалізують основи інструментарію інтерфейсу користувача, такі як LayoutNode, Modifier, обробники вводу, користувацькі макети і малювання. Ви можете розглянути можливість побудови на цьому шарі, якщо вам потрібні тільки основні концепції інструментарію інтерфейсу користувача.

Модуль Foundation забезпечує будівельні блоки для Compose UI, які не залежать від системи дизайну, такі як Row і Column, LazyColumn, розпізнавання певних жестів тощо. Ви можете розглянути можливість побудови на шарі Foundation для створення власної системи дизайну.

Модуль Material надає реалізацію системи Material Design для Compose UI, забезпечуючи систему тем, стилізовані компоненти, індикацію, іконки. Використовуйте цей шар при застосуванні Material Design у вашому додатку [17].

Реалізація Jetpack Compose набагато простіша і стисліша в порівнянні з традиційним підходом – не потрібно мати справу з адаптерами, власниками

представлень або складними XML-макетами. Натомість визначаються composable-функції, які представляють компоненти інтерфейсу користувача, і складаються разом для побудови ієрархії інтерфейсу. Jetpack Compose використовує стислий та виразний синтаксис Kotlin, дозволяючи створювати динамічні та інтерактивні інтерфейси з мінімальною кількістю шаблонного коду.

2.3. Аналіз архітектури UI фреймворку: Flutter

Фреймворк Flutter [4] розроблено як розширювану 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 замінювана (рис. 2.7).

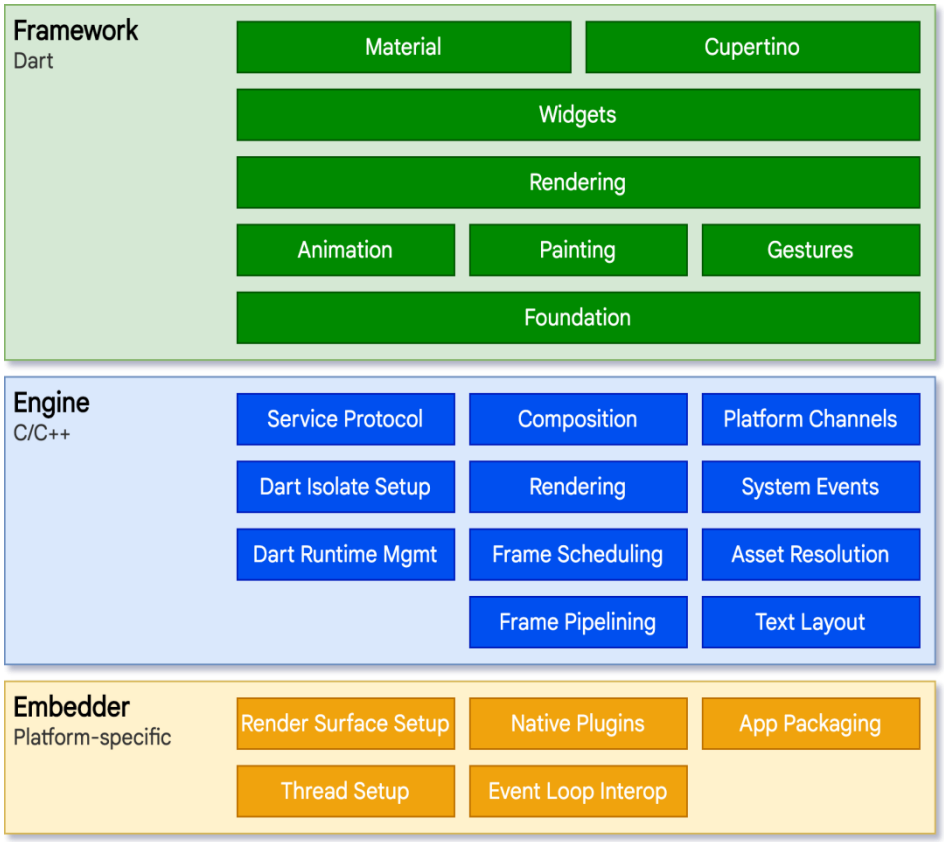

Рисунок 2.7 – Архітектурні прошарки

Flutter-додатки пакуються як нативні програми для кожної операційної системи. Embedder-прошарок забезпечує точку входу, координує доступ до системних сервісів (рендеринг, введення тощо) і керує циклом подій. Він реалізований нативною мовою платформи. Embedder дозволяє інтегрувати

Flutter-код у наявну програму як модуль або використовувати його як основний контент.

Ядром Flutter є Flutter Engine, написаний на C++. Рушій відповідає за растеризацію композитних сцен, реалізацію основного API Flutter, графіку (Impeller для iOS та Android, Skia – для інших платформ), текстову розмітку, файловий та мережевий ввід/вивід, підтримку доступності, архітектуру плагінів, Dart Runtime та інструменти компіляції.

Flutter Engine доступний через dart:ui, що огортає базовий C++ код у Dart-класи, надаючи примітиви для управління введенням, графікою і рендерингом тексту. Розробники взаємодіють з Flutter через Flutter Framework, який надає сучасний реактивний фреймворк на мові Dart. Він включає платформні, макетні та основні бібліотеки, складені з кількох шарів:

– базові фундаментальні класи – анімація, малювання, жести;

– прошарок рендерингу – абстракція для роботи з макетом і динамічна маніпуляція деревом об’єктів рендерингу;

– прошарок віджетів – абстракція композиції, модель реактивного програмування;

– бібліотеки Material і Cupertino – набори управлінських елементів (контролів) для реалізації дизайнів Material або iOS.

Фреймворк Flutter невеликий, багато функцій реалізовані як пакети, включаючи платформні плагіни та кросплатформні функції, що базуються на основних бібліотеках Dart і Flutter.

Рисунок 2.8 дає огляд компонентів звичайного Flutter-додаткаи, створеного за допомогою flutter create. Вона показує розташування Flutter Engine в цій структурі, підкреслює межі API і визначає сховища, де знаходяться окремі компоненти.

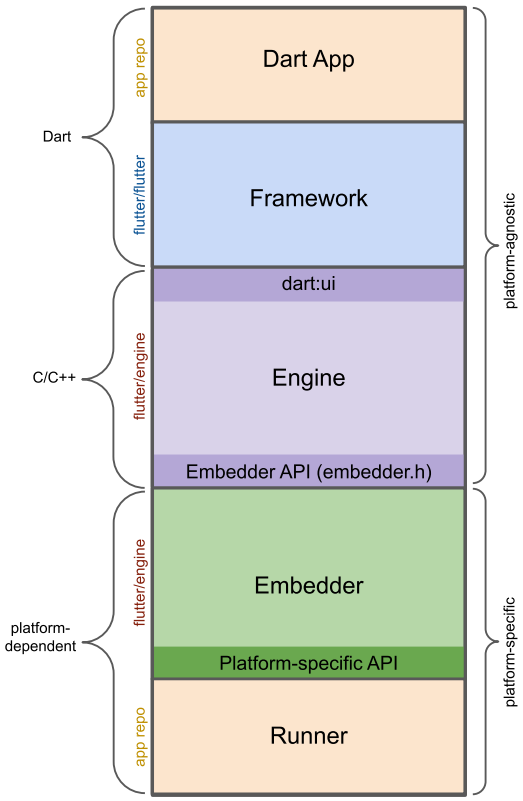

Рисунок 2.8 – Анатомія Flutter-додатка

Прошарок Dart App комбінує віджети для створення бажаного інтерфейсу користувача та реалізує бізнес-логіку. Він контролюється розробником програми.

Рівень Framework надає високорівневий API для створення якісних додатків (віджети, hit-testing, виявлення жестів, доступність, введення тексту). Він комбінує дерево віджетів додатка в сцену.

Прошарок Engine відповідає за растеризацію комбінованих сцен. Він надає низькорівневу реалізацію основних API Flutter (наприклад, графіка, текстова розмітка, виконавче середовище Dart), а також забезпечує функціональність фреймворку за допомогою dart:ui API та інтегрується з конкретною платформою за допомогою Embedder API.

Вже згаданий Embedder координується з операційною системою для доступу до таких сервісів, як поверхні рендерингу, доступність і введення. Він керує циклом подій та надає платформоспецифічний API для інтеграції Embedder в додатки.

Рівень Runner комбінує компоненти, які надає платформоспецифічний API Embedder, в пакет програми, що запускається на цільовій платформі. Частина шаблону додатка, створеного за допомогою flutter create, контролюється розробником програми.

Flutter – це реактивний, декларативний UI-фреймворк, в якому розробник надає відображення від стану додатка до стану інтерфейсу, а фреймворк відповідає за оновлення інтерфейсу під час зміни стану програми. У більшості традиційних UI-фреймворків початковий стан інтерфейсу користувача описується один раз, а потім оновлюється кодом користувача під час виконання у відповідь на події. Однією з проблем цього підходу є те, що зі збільшенням складності програми розробнику потрібно знати, як зміни стану впливають на весь інтерфейс користувача [1].

У Flutter віджети (аналогічні компонентам у React) представлені незмінюваними класами, що використовуються для конфігурації дерева об’єктів. Ці віджети керують окремим деревом об’єктів для макета і окремим деревом компонуючих об’єктів. Flutter забезпечує ефективний обхід змінених частин дерев та поширення змін через них.

Віджет декларує свій інтерфейс користувача, замістивши метод build(), який відображає стан на користувацький інтерфейс. Метод build() виконується швидко і без побічних ефектів, дозволяючи фреймворку викликати його за потреби. Мова Dart доречна для швидкого створення і видалення об’єктів.

Віджети є основними будівельними блоками інтерфейсу користувача у Flutter-програмі. Вони формують ієрархію на основі композиції. Кожен віджет вкладається у свого батька і може отримувати контекст від батька. Ця структура тягнеться до кореневого віджета, який містить Flutter-додаток (зазвичай MaterialApp або CupertinoApp)., як показано у лістингу 2.5.

У наведеному коді всі створені класи є віджетами. Програми оновлюють свій інтерфейс користувача у відповідь на події (такі як взаємодія користувача), замінюючи віджет у ієрархії іншим віджетом. Фреймворк порівнює нові і старі

віджети і ефективно оновлює інтерфейс користувача. Flutter має власні реалізації кожного UI-контролю, замість системних [13].

Лістинг 2.5 – Демонстрація опису віджета

```
import ‘package:flutter/material.dart’;
import ‘package:flutter/services.dart’;

void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: const Text(‘My Home Page’),
        ),
        body: Center(
          child: Builder(
            builder: (context) {
              return Column(
                children: [
                  const Text(‘Hello World’),
                  const SizedBox(height: 20),
                  ElevatedButton(
                    onPressed: () {
                      print(‘Click!’);
                    },
                    child: const Text(‘A button’),
                  ),
                ],
              );
            },
          ),
        ),
      ),
    );
  }
}
```

Переваги цього підходу:

– розробник може створювати будь-які варіанти контролів без обмежень точками розширення ОС;

– Flutter може одночасно компонувати всю сцену;

– програма виглядає і працює однаково на всіх версіях ОС, навіть якщо система змінила реалізації своїх контролів.

Віджети у Flutter складаються з невеликих, однопрофільних компонентів, що комбінуються для створення складних інтерфейсів. Flutter використовує мінімум концепцій для створення широкого словника. Наприклад, концепт Widget використовується для відображення на екрані, макетування, взаємодії з користувачем, керування станом, темізацією, анімаціями та навігацією.

У шарі рендерингу RenderObjects описують макетування, малювання, тестування попадання і доступність. Класова ієрархія неглибока і широка для максимізації комбінацій. Основні можливості є абстрактними, навіть базові функції, такі як відступи та вирівнювання, реалізовані як окремі компоненти.

Фреймворк вводить два основні класи віджетів: stateful і stateless. Багато віджетів не мають змінного стану: їхні властивості не змінюються з часом (наприклад, іконка або мітка). Такі віджети успадковуються від StatelessWidget. Якщо характеристики віджета повинні змінюватися на основі взаємодії користувача або інших факторів, цей віджет є stateful. Наприклад, якщо віджет має лічильник, що збільшується при натисканні кнопки, то значення лічильника є станом для цього віджета. Коли це значення змінюється, віджет повинен бути перебудований для оновлення своєї частини інтерфейсу. Такі віджети успадковуються від StatefulWidget і зберігають змінний стан у окремому класі, який успадковується від State. StatefulWidgets не мають методу build(); замість цього їхній інтерфейс створюється через об’єкт State. Коли змінюється об’єкт State (наприклад, при збільшенні лічильника), необхідно викликати setState(), щоб сигналізувати фреймворку про оновлення інтерфейсу користувача шляхом повторного виклику методу build() об’єкта State.

Стан керується і передається системою через конструктор у віджеті для ініціалізації його даних, щоб метод build() міг переконатися, що будь-який дочірній віджет створений з необхідними даними.

Зі збільшенням глибини дерев віджетів передача інформації про стан стає обтяжливою. Тому третій тип віджета, InheritedWidget, забезпечує легкий спосіб

отримати дані від спільного предка. InheritedWidget можна використовувати для створення віджета стану, який обгортає спільного предка в дереві віджетів.

Виклик методу of(context) отримує контекст побудови (вказівник на поточне розташування віджету) і повертає найближчого предка в дереві. InheritedWidgets також пропонують метод updateShouldNotify(), який Flutter викликає, щоб визначити, чи має зміна стану спричинити перебудову дочірніх віджетів, які його використовують. Flutter широко використовує InheritedWidget як частину каркасу для спільного стану, такого як візуальна тема додатка, яка включає властивості, такі як колір і стилі шрифту.

Із ростом додатків стають привабливими більш продумані підходи до управління станом, що спрощують процес створення та використання віджетів із збереженням стану. Багато додатків Flutter використовують утилітарні пакети, такі як provider, який надає обгортку навколо InheritedWidget.

Flutter мінімізує абстракції, обходячи бібліотеки системних віджетів інтерфейсу користувача на користь власного набору віджетів. Dart-код, який відрисовує візуальні елементи Flutter, компілюється в нативний код, який використовує Skia (або Impeller) для рендерингу. Flutter також вбудовує свою власну копію Skia як частину рушія, що дає змогу розробнику оновлювати свій додаток до останніх покращень продуктивності, навіть якщо смартфон не був оновлений до нової версії Android. Те ж саме стосується Flutter на інших платформах, таких як Windows або macOS. Flutter має простий конвеєр для потоку даних до системи, як показано на рис. 2.10.

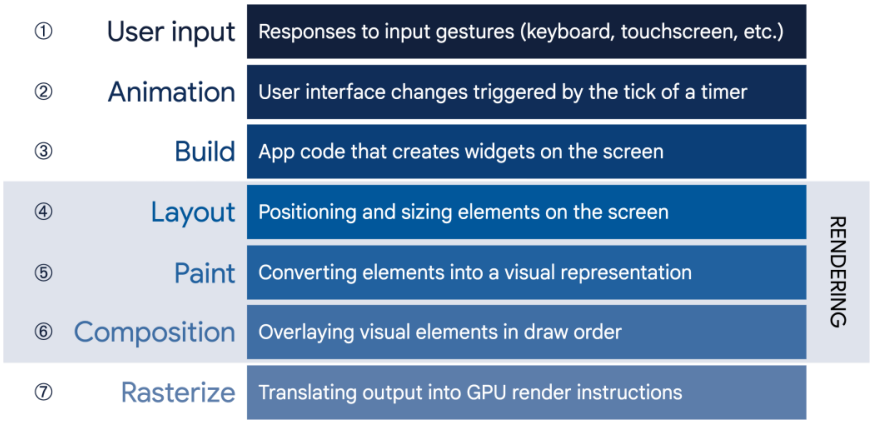

Рисунок 2.10 – Потік даних до системи

Розглянемо фрагмент коду з лістингу 2.6. Коли Flutter потрібно відобразити цей фрагмент, він викликає метод build(), який повертає піддерево віджетів для відтворення інтерфейсу користувача на основі поточного стану додатка. Під час цього процесу метод build() може додавати нові віджети відповідно до свого стану. Наприклад, у наведеному коді Container має властивості color і child.

Лістинг 2.6 – Опис контейнера

```
Container(
  color: Colors.blue,
  child: Row(
    children: [
      Image.network(‘https://www.example.com/1.png’),
      const Text(‘A’),
    ],
  ),
);
```

Відповідно, віджети Image і Text можуть додавати дочірні віджети, такі як RawImage і RichText, під час процесу побудови. У результаті ієрархія віджетів може бути глибшою, ніж показано у коді (рис. 2.11). На етапі створення Flutter перетворює віджети, виражені в коді, у відповідне дерево елементів, з одним елементом для кожного віджета. Кожен елемент представляє конкретний екземпляр віджета в заданому місці ієрархії дерева. Існує два основних типи елементів:

– ComponentElement, хост для інших елементів;

– RenderObjectElement, елемент, який бере участь у фазах макетування або відрисовки.

Важлива частина будь-якого UI-фреймворку — це здатність ефективно розташовувати ієрархію віджетів, визначаючи розмір і позицію кожного елемента перед їх рендерингом на екрані. Базовий клас для кожного вузла в дереві рендерингу — RenderObject, який визначає абстрактну модель для макету і малювання. Це дуже загальна модель: вона не прив’язана до фіксованої кількості вимірів або навіть до декартової системи координат. Кожен RenderObject знає свого батька, але мало знає про своїх дітей, крім того, як їх

відвідати та їхні обмеження. Це надає RenderObject достатню абстракцію для обробки різних випадків використання. Під час фази побудови Flutter створює або оновлює об’єкт, що наслідує RenderObject для кожного RenderObjectElement у дереві елементів. RenderObject-и є примітивами: RenderParagraph відрисовує текст, RenderImage — зображення, а RenderTransform застосовує трансформацію перед відрисовкою свого дочірнього елемента.

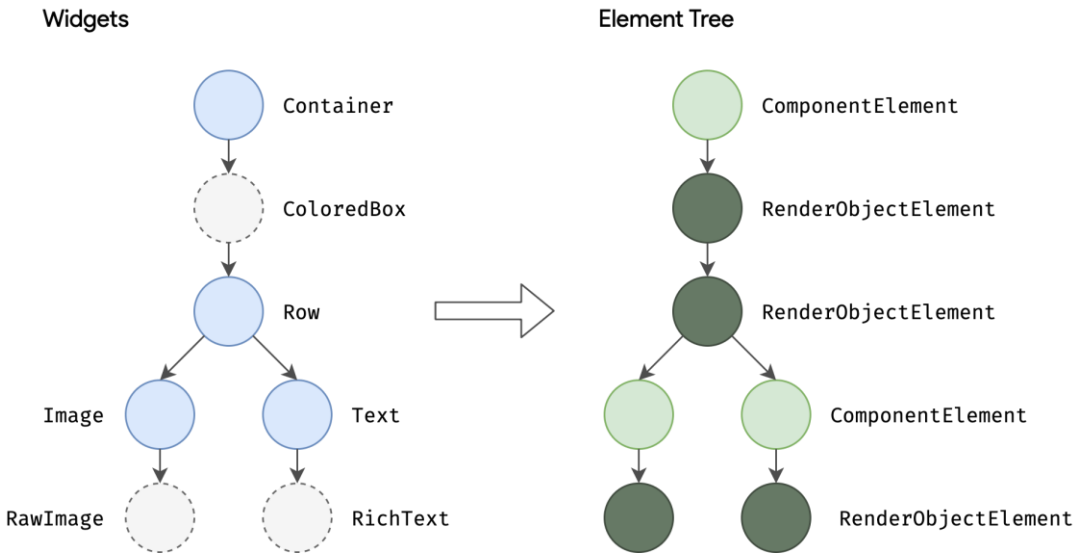

Рисунок 2.11 – Віджети та дерево елементів, Element Tree

Відмінності між ієрархією віджетів, деревами елементів та рендерингу продемонстровано на рис. 2.12.

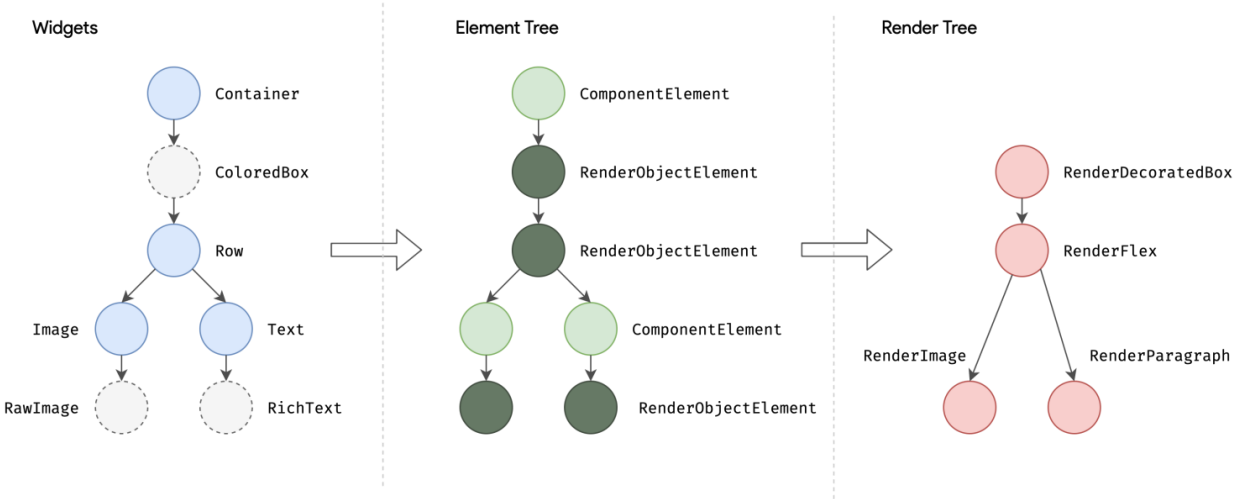

Рисунок 2.12 – Віджети, дерево елементів та дерево відрисовки

Більшість віджетів Flutter відрисовуються об’єктом, який наслідує клас RenderBox, що представляє RenderObject фіксованого розміру в декартовому 2D-просторі. RenderBox надає основу для моделі обмежень, встановлюючи мінімальну та максимальну ширину і висоту для кожного віджета. Щоб виконати макет, Flutter проходить дерево рендерингу у глибину, передаючи обмеження розмірів від батьківського елемента до дочірнього. Визначаючи свій розмір, дочірній елемент повинен дотримуватися обмежень, встановлених батьком. Обмеження спускаються вниз, розміри піднімаються вгору. Наприкінці проходження дерева кожен об’єкт має визначений розмір у межах обмежень батька і готовий до відрисовки методом paint(). Модель обмежень дає змогу розташовувати об’єкти за час $O(n)$.

Flutter будує, розташовує, компонує і відрисовує інтерфейси користувача самостійно. Платформа-завантажувач — це нативний додаток ОС, який хостить весь контент Flutter, і виступає як сполучна ланка між операційною системою і Flutter. Під час запуску додатка завантажувач надає точку входу, ініціалізує рушій Flutter, отримує потоки для UI і растеризації та створює текстуру, яку Flutter може використовувати. Завантажувач також відповідає за життєвий цикл додатка, включаючи обробку жестів введення, зміну розміру вікна, управління потоками та повідомленнями платформи.

Flutter включає завантажувачі для Android, iOS, Windows, macOS і Linux (рис. 2.13); ви також можете створити власний завантажувач платформи, як у прикладах для віддалених сесій або Raspberry Pi. Кожна платформа має свій набір API та обмежень

– На iOS та macOS Flutter завантажується як UIViewController або NSViewController відповідно, використовуючи Metal або OpenGL;

– На Android Flutter завантажується як Activity, а перегляд контролюється FlutterView;

– На Windows Flutter розміщується в традиційному Win32-додатку і відрисовує інтерфейс за допомогою ANGLE, який перетворює виклики API OpenGL на еквіваленти DirectX 11.

Flutter дозволяє викликати нативний код через платформний канал, який є механізмом для спілкування між вашим кодом Dart і платформно-специфічним кодом вашого хост-додатку. Це дозволяє надсилати та отримувати повідомлення між Dart і нативним компонентом.

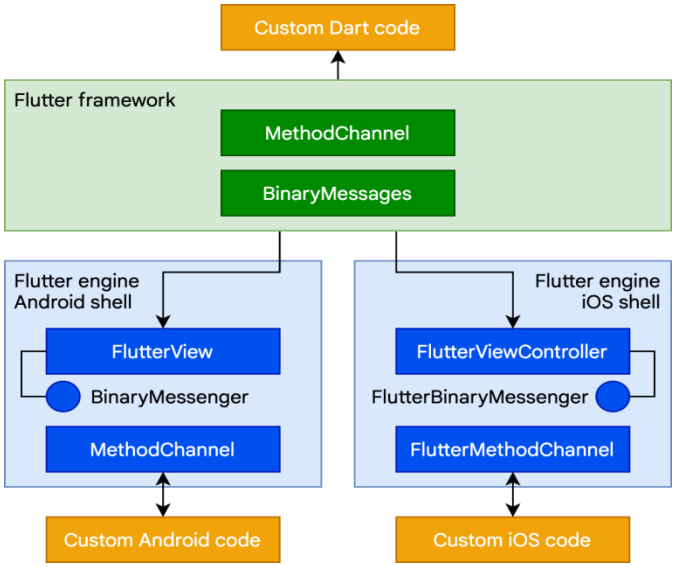

Рисунок 2.13 – Схема інтеграції

Для API на базі C, включаючи ті, що можуть бути згенеровані для коду, написаного сучасними мовами, як Rust або Go, Dart надає прямий механізм для прив'язки до нативного коду, використовуючи бібліотеку dart:ffi. Модель інтерфейсу сторонніх функцій (FFI) може бути значно швидшою, ніж платформні канали, тому що не потрібно серіалізувати дані для передачі. Натомість, час виконання Dart надає можливість виділяти пам'ять у купі, яка підтримується об'єктом Dart, та здійснювати виклики до статично або динамічно пов'язаних бібліотек. FFI доступний для всіх платформ, крім вебу, де js package виконує аналогічну функцію.

Загальне технічне порівняння фреймворків, що були розглянуті, наведено в таблиці 2.1. Кожна з розглянутих платформ має свої переваги та підходить для різних випадків використання. Jetpack Compose є чудовим вибором для розробки додатків під Android, забезпечуючи простоту і ефективність. React Native пропонує потужну міжплатформну інтеграцію та високу продуктивність завдяки новій архітектурі. Flutter забезпечує максимальну кросплатформність та

відмінний користувацький досвід завдяки власним віджетам та високопродуктивному рендерингу. Обираючи платформу для розробки додатку, важливо враховувати вимоги проекту, зручність використання, продуктивність та можливості тестування, щоб забезпечити найкращий результат для кінцевого користувача.

Таблиця 2.1 – Порівняння UI-фреймворків

Аспект	Jetpack Compose	React Native	Flutter
Композиція	Функції на Kotlin, що використовують @Composable анотацію для опису UI	Компоненти на JavaScript/JSX, що описують структуру UI через декларативні вирази	Віджети на Dart, що представляють UI у вигляді ієрархії віджетів
Стан віджетів	Зберігання стану у об'єктах State та MutableState, що дозволяє реактивно оновлювати UI	Використовує setState для локального стану, а також сторонні бібліотеки (Redux, MobX) для глобального стану	Зберігання стану у StatefulWidget та керування станом через Provider, Riverpod
Управління станом	ViewModel та LiveData з архітектури MVVM, що забезпечує розділення логіки та представлення	Контекст API для локального стану, Redux і MobX для глобального стану	State для локального стану, InheritedWidget для передачі стану, Provider та Riverpod для керування станом
Макети та рендеринг	Декларативні функції композиції, що визначають макети через @Composable функції	Декларативні компоненти, що описують макети через JSX/TSX синтаксис	Декларативні віджети, що описують макети через дерево віджетів
Рендеринг	Нативний Android рендеринг, що використовує платформні компоненти та API	Нативні компоненти через міст, що дозволяє використовувати нативні елементи UI	Власний рендеринг-рушій, що оминає нативні компоненти та використовує власні графічні бібліотеки (Skia)
Інтеграція з платформою	Глибока інтеграція з Android, що дозволяє використовувати всі можливості платформи та існуючий Android-код	Обмежена інтеграція, що потребує використання нативних модулів для доступу до платформних API	Високий рівень інтеграції через платформні канали, що дозволяє взаємодіяти з нативним кодом та API
Інтеграція зі стороннім кодом	Легко інтегрується з існуючим Android-кодом, підтримує використання стандартних Android-бібліотек	Використовує нативні модулі та міст для інтеграції з існуючим кодом, може вимагати додаткових зусиль для інтеграції	Підтримує інтеграцію з нативним кодом через платформні канали, дозволяє використовувати існуючі бібліотеки та SDK

РОЗДІЛ 3
ТИПОВІ СЦЕНАРІЇ ВИКОРИСТАННЯ ДЕКЛАРАТИВНИХ UI-ФРЕЙМВОРКІВ

3.1. Порівняння фреймворків за сценаріями використання

Маючи таке різноманіття фреймворків, неможливо виділити якийсь загальний сценарій використання всіх фреймворків, все залежить від мети, підходу та знань розробника мобільного додатку.

Так як React Native є кросплатформним фреймворком, то перший сценарій використання випливає з потреби розробки програми, яка виглядатиме і поводитиметься однаково як на Android, так і на IOS. Якщо розробник знайомий із розробкою на JavaScript та React.JS, йому буде значно простіше влитися у процес розробки на React Native. Але це не означає, що розробнику не потрібно знати Java/Kotlin та Swift. Якщо перед ним стоїть нетривіальне завдання, з яким не можуть допомогти вбудовані функції React Native, йому доведеться зіткнутися з нативними модулями. Нативні модулі React Native допомагають розширити можливості програми, надавши доступ до нативних функцій і методів для платформи розробки. Нативні модулі пишуться рідними мовами для платформ, якими є для Android: Java/Kotlin, а для IOS: Swift. В ідеалі нативний модуль повинен на обох платформах працювати однаково, щоб у самому React Native не робити додаткових перевірок і використовувати єдиний код. Також React Native часто вибирають за можливість швидкого прототипування програми за рахунок вже озвучених його переваг у вигляді власних компонентів, які виглядають єдино на всіх платформах.

Flutter варто використовувати при розробці кросплатформних програм для Android, iOS, веб і настільних платформ. Visual Studio Code та Android Studio надають інтеграцію з Flutter, пропонуючи інструменти для попереднього перегляду та налагодження інтерфейсів, створених за допомогою Flutter, що дозволяє бачити зміни в реальному часі та швидше тестувати різні компоненти. Flutter значно скорочує кількість шаблонного коду, який часто зустрічається у традиційних Android- та iOS-додатках, написаних з використанням нативних

інструментів. Також Flutter можна інтегрувати в існуючі Android- та iOS-програми, що дає змогу поступово переходити на новий фреймворк без необхідності повної переробки програми.

Flutter пропонує вбудовані інструменти для створення складних анімацій і темізації інтерфейсів, що спрощує створення привабливих і динамічних інтерфейсів користувача. Він інтегрується з різними бібліотеками та плагінами, що спрощує побудову архітектури програми та управління станом. Наприклад, такі пакети, як provider та riverpod, допомагають ефективно керувати станом.

Вибір Vue Native для розробки мобільного додатка може бути кращим, якщо розробник вже має досвід розробки з Vue.js. Vue Native побудований на React Native, що дає змогу використовувати плагіни та бібліотеки звідти і це також відкриває можливість кросплатформності при розробці додатків. Vue Native добре показує себе на етапі прототипування додатків.

NativeScript вибирають найчастіше ті, хто хочуть використовувати при розробці JavaScript або TypeScript як основні мови програмування. Так, NativeScript дозволяє використовувати Angular у розробці, що спрощує процес навчання та робить фреймворк доступним для широкого кола розробників. Він надає повний доступ до нативних API платформ Android та iOS, що дозволяє розробляти програми з нативною продуктивністю та інтерфейсом користувача. Ви можете використовувати існуючі JavaScript-бібліотеки, а також створювати власні нативні модулі, якщо вам потрібна специфічна функціональність. Загалом, NativeScript є потужним інструментом для тих, хто хоче розробляти кросплатформні мобільні додатки з нативною продуктивністю та доступом до нативних API, використовуючи сучасні вебтехнології.

SwiftUI варто вибирати насамперед, якщо йде розробка суто під екосистему Apple, а саме iOS, macOS, watchOS та tvOS. Це дозволяє створювати універсальні програми із загальними компонентами, що знижує витрати на розробку та підтримку. Використання SwiftUI дозволяє уніфікувати стиль програмування інтерфейсів з використанням однієї мови Swift. Це робить код більш читабельним та простим для розуміння. SwiftUI інтегрований з Xcode,

надаючи можливості для інтерактивного проєктування. Розробники можуть бачити зміни в реальному часі за допомогою Canvas, що прискорює процес розробки та тестування.

SwiftUI можна поступово впроваджувати в існуючі проекти, оскільки він добре інтегрується з UIKit (для iOS) та AppKit (для macOS). Це дозволяє використовувати новий підхід до розробки інтерфейсів без повної переробки існуючого коду. SwiftUI тісно інтегрований з Combine – фреймворком для роботи з асинхронними подіями. Це дає змогу створювати реактивні програми, де UI автоматично оновлюється під час зміни даних [6].

Jetpack Compose варто використовувати при розробці під Android, він написаний на Kotlin і використовує всі можливості, включаючи корутини й розширені функції. Це дозволяє створювати більш компактний, виразний та безпечний код. Android Studio надає інструменти для попереднього перегляду та налагодження інтерфейсів, створених за допомогою Compose, що дозволяє бачити зміни в реальному часі та швидше тестувати різні компоненти. Compose значно скорочує кількість шаблонного коду, який часто зустрічається у традиційних Android-додатках, написаних з використанням XML. Фреймворк можна інтегрувати в існуючі Android-додатки, що дозволяє поступово переходити на новий фреймворк без необхідності повної переробки програми. Compose UI може співіснувати з класичними XML-верстками та компонентами.

Jetpack Compose пропонує вбудовані інструменти для створення складних анімацій і темізації інтерфейсів, що спрощує створення привабливих і динамічних інтерфейсів користувача. Compose інтегрується з іншими бібліотеками з екосистеми Jetpack, такими як Navigation, LiveData та ViewModel, що спрощує побудову архітектури програми та управління станом. Jetpack Compose надає сучасні інструменти та підходи для розробки UI на Android, спрощуючи процес створення, тестування та підтримки додатків, що робить його чудовим вибором для сучасних Android-розробників.

3.2. Порівняння фреймворків за іншими критеріями

Як зазначалось у другому розділі, розглянуті UI-фреймворки мають свої ключові особливості. Зокрема, засобами Jetpack Compose користувацький інтерфейс описується в вигляді функцій Kotlin, код компілюється в байткод, виконуваний на JVM, і потім працює на Android. Оскільки Jetpack Compose створений для Android, він інтегрується безпосередньо з платформними компонентами та API Android.

У той же час, декларативний підхід React Native визначає опис користувацького інтерфейсу за допомогою JSX (JavaScript XML). JavaScript-код виконується в окремому потоці (JavaScriptCore на iOS і Hermes на Android), який взаємодіє з нативними компонентами через міст (bridge). Для доступу до нативних функцій та API використовується система нативних модулів.

У свою чергу, Flutter описує UI у вигляді коду на Dart. Він використовує власний рушій рендерингу для відображення користувацького інтерфейсу, оминаючи нативні компоненти платформи. Всі елементи інтерфейсу у Flutter представлені віджетами, включаючи кнопки, текстові поля та анімації.

Ці фреймворки мають різні архітектурні підходи, що впливають на їх продуктивність, споживання ресурсів та загальну зручність використання. Для детального порівняння основних характеристик та продуктивності цих фреймворків, розглянемо таблицю 3.1, яка надає більш детальну інформацію про кожен з них.

Таблиця 3.1 – Порівняння основних фреймворків за різними критеріями

Критерій	**Jetpack Compose**	**React Native**	**Flutter**
Хостинг контенту	Android (експериментально: Desktop, Web)	iOS, Android, (експериментально: Windows, macOS)	iOS, Android, Web, Desktop
Рендеринг нативних контролів	Так, використовує нативні Android контролі для відображення UI елементів	Так, використовує нативні контролі через міст для відображення UI елементів	Ні, використовує власні віджети та рендеринг-движок для відображення UI елементів
Використовувані інтерфейси	Kotlin функції з використанням @Composable анотацій	JavaScript/JSX компоненти, що описують UI через декларативні вирази	Dart віджети, що представляють UI через ієрархію віджетів

Продовження таблиці 3.1

Архітектурні шари	ViewModel, LiveData, Compose Runtime, що забезпечують реактивну модель управління станом	JavaScriptCore (або Hermes), Native Modules, Bridge, що забезпечують взаємодію між JavaScript та нативними компонентами	Flutter Framework, Engine, Platform Channels, що забезпечують власний рендеринг та взаємодію з нативним кодом
Анатомія програми	Activity/Fragment, Composable функції, що представляють структуру та логіку UI	JavaScript код, React компоненти, Native Modules, що визначають структуру та логіку UI	main.dart, віджети, платформенні канали, що представляють структуру та логіку UI

Окремо розглянуто порівняння продуктивності фреймворків за показниками продуктивності використання обчислювальних ресурсів.

Таблиця 3.2 – Порівняння продуктивності

Параметр	**Jetpack Compose**	**React Native**	**Flutter**
Пам’ять	Оптимальне використання пам’яті завдяки нативному виконанню та оптимізації для Android	Може бути високим через використання моста та JavaScriptCore, що додає накладні витрати	Оптимальне використання пам’яті, але залежить від кількості та складності віджетів та анімацій
Процесорний час	Висока продуктивність завдяки нативному виконанню та оптимізації для Android	Середня продуктивність, можливі затримки через міст та JavaScript	Висока продуктивність завдяки використанню власного рендеринг-движка та оптимізації для високої продуктивності
Розмір бінарного файлу	Невеликий, залежить від Android SDK та використовуваних бібліотек	Середній, збільшується за рахунок включення JavaScriptCore та нативних модулів	Зазвичай більший, через включення Flutter Engine та всіх необхідних бібліотек
Енергоспоживання	Ефективне, оптимізоване для Android, мінімізує використання ресурсів	Може бути вищим через використання JavaScriptCore та моста, що додає накладні витрати	Ефективне, але залежить від використання анімацій та частоти оновлення UI
Швидкість запуску програми	Швидка, завдяки нативному виконанню та оптимізації для Android	Середня, через необхідність ініціалізації моста та JavaScriptCore	Швидка, завдяки ефективному виконанню та оптимізації Flutter Engine

3.3. Бенчмаркінг декларативних UI-фреймворків

Бібліотека Macrobenchmark вимірює більші взаємодії кінцевого користувача, такі як запуск, взаємодія з інтерфейсом користувача та анімаціями. Бібліотека забезпечує прямий контроль над середовищем продуктивності, яке ви тестуєте. Він дає вам змогу контролювати компіляцію, а також запускати та зупиняти свою програму, щоб безпосередньо вимірювати фактичний запуск або прокручування програми.

Бібліотека Macrobenchmark вводить події та відстежує результати ззовні з тестової програми, створеної на основі ваших тестів. Таким чином, під час написання контрольних тестів ви не викликаєте код програми безпосередньо, а натомість переходите в програмі як користувач [14].

Бібліотека Microbenchmark дозволяє порівнювати код програми безпосередньо в циклі виконання. Вона розроблена для вимірювання роботи ЦП, яка оцінює продуктивність у найкращому випадку, як-от кешування звернень до диска, які ви можете побачити за допомогою внутрішнього циклу або певної гарячої функції. Бібліотека може вимірювати лише код, який ви можете викликати безпосередньо ізольовано.

Якщо вашій програмі потрібно обробляти складну структуру даних або мати певний важкий алгоритм, який викликається кілька разів під час роботи програми, це може бути хорошим прикладом для порівняльного аналізу. Ви також можете вимірювати частини свого інтерфейсу користувача. Наприклад, ви можете виміряти вартість прив’язки елемента RecyclerView, скільки часу потрібно для роздування макета або наскільки вимогливим є проходження макета та вимірювання вашого класу View з точки зору продуктивності. Однак ви не можете виміряти, як контрольні випадки сприяють загальному досвіду користувача. У деяких сценаріях порівняльний аналіз не скаже вам, чи ви покращуєте вузьке місце, як-от джек або час запуску програми. З цієї причини дуже важливо спочатку визначити ці вузькі місця за допомогою Android Profiler. Після того, як ви знайдете код, який хочете дослідити та оптимізувати, цикл із порівняльним тестуванням можна запускати швидко та легше, щоб отримати

результати з меншим шумом, що дозволить вам зосередитися на одній області вдосконалення.

Бібліотека Macrobenchmark повідомляє лише інформацію про вашу програму, а не про систему в цілому. Тому найкраще аналізувати продуктивність ситуацій, пов'язаних із програмою, а не тих, які можуть бути пов'язані із загальними проблемами системи.

Інструментальні тестування з використанням цієї бібліотеки не викликають ваш код додатку безпосередньо, а натомість навігують по додатку, як це робив би користувач. Тести Macrobenchmark запускаються в окремому процесі, щоб дозволити перезапуск або попередню компіляцію вашого додатку. Це означає, що механізми, такі як Espresso, не працюватимуть; натомість ви можете використовувати UiAutomator для взаємодії з цільовим додатком.

Вимірювання часу запуску додатку. Час запуску додатку, або час, необхідний для того, щоб користувачі почали використовувати ваш додаток, є ключовим показником залучення користувачів. Щоб виміряти час запуску додатку за допомогою макробенчмарків, напишіть тест @Test таким чином, як на рис. 3.1.

```
1  // Copyright 2022 Google LLC.
2  // SPDX-License-Identifier: Apache-2.0
3
4  @RunWith(AndroidJUnit4::class)
5  class StartupBenchmark {
6     @get:Rule
7     val benchmarkRule = MacrobenchmarkRule()
8
9     @Test
10    fun startup() = benchmarkRule.measureRepeated(
11        packageName = "com.google.samples.apps.sunflower",
12        metrics = listOf(StartupTimingMetric()),
13        iterations = 5,
14        startupMode = StartupMode.COLD,
15    ){
16         pressHome()
17         startActivityAndWait()
18    }
19 }
```

Рисунок 3.1 – Шаблон вимірювання часу в Android Studio

Макробенчмарки — це регулярні інструментовані модульні тести, тому вони використовують синтаксис JUnit — @RunWith, @Rule, @Test тощо. Під час написання бенчмарка, точкою входу є функція measureRepeated класу MacrobenchmarkRule, де необхідно вказати принаймні ці параметри:

– packageName – оскільки бенчмарки виконуються в окремому процесі, потрібно вказати, який додаток вимірювати.

– metrics – основний тип інформації, що фіксується. У нашому випадку це час запуску.

– iterations – скільки разів повторюється цикл. Більше ітерацій означає більш стабільні результати, але за рахунок більш тривалого часу виконання.

– measureBlock (останній параметр-лямбда) – Macrobenchmark буде відслідковувати і записувати визначені метрики під час цього блоку. Дії, які потрібно виміряти, виконуються тут.

Опційно можна також вказати CompilationMode та StartupMode. CompilationMode визначає, як додаток буде попередньо скомпільовано в машинний код і має такі опції:

– None() – не компілює додаток заздалегідь, але JIT все ще ввімкнений під час виконання додатка;

– Partial() – попередньо компілює додаток з використанням Baseline Profiles і/або прогрівочних запусків;

– Full() – повністю попередньо компілює додаток. Це єдина опція для Android 6 (API 23) і нижче.

Параметр StartupMode дозволяє визначити, як додаток має запускатися на початку бенчмарку. Доступні опції: COLD, WARM і HOT. Можна почати з StartupMode.COLD, що представляє найбільший обсяг роботи, яку має виконати додаток.

Під час виконання бенчмарка додаток буде запускатися та зупинятися кілька разів (залежно від кількості ітерацій), а потім результати будуть виведені в Android Studio, як показано на рис. 3.2.

Рисунок 3.2 – Результат виконання бенчмарка

Результати надаються як час у мілісекундах, який потрібен для запуску вашого додатка (timeToInitialDisplayMs). Кожен результат є посиланням із системним трасуванням, яке можна відкрити в Android Studio для подальшого аналізу, як виглядав запуск і вжити заходів для його оптимізації.

Зазвичай вимірюють час повного завантаження вмісту додатка і можливості взаємодії з ним користувача, також званий time to full display. Щоб повідомити систему, коли це відбудеться, необхідно викликати Activity.reportFullyDrawn(). У такому разі бенчмарк автоматично захоплює timeToFullDisplayMs. Важливо дочекатися завантаження вмісту в бенчмарках, інакше бенчмарк завершиться з першим відображеним кадром і може пропустити метрику.

За результатами опису інструментальної основи для виконання порівняння декларативних UI-фреймворків визначено основні аспекти, які можуть оцінюватись і порівнюватись при аналізі їх застосовності. Велику роль для такої оцінки відіграють не лише технічні показники ефективності використання обчислювальних ресурсів, а й орієнтування фреймворку на визначені платформи та знайомство розробника з інструментальними засобами, які забезпечують функціонування фреймворку.

ВИСНОВКИ

У цій роботі ми досліджували декларативні UI-фреймворки, які використовуються для розробки мобільних додатків. Декларативний підхід дозволяє розробникам описувати, що додаток має робити, а не як це робити, що значно спрощує процес розробки. Ми розглянули основні принципи декларативних фреймворків, їхні переваги та недоліки, а також порівняли найбільш популярні з них: React Native, Vue Native та Flutter.

Декларативні UI-фреймворки пропонують численні переваги, включаючи простоту коду, легкість у підтримці та читабельність. Завдяки декларативному підходу розробники можуть зосередитися на логіці додатку, не турбуючись про низькорівневі деталі реалізації. Кожен фреймворк має свої переваги та недоліки. Наприклад, React Native відомий своєю великою спільнотою та численними бібліотеками, але може мати проблеми з продуктивністю в складних додатках. Vue Native є легким у використанні, але має меншу екосистему порівняно з React Native. Flutter забезпечує високу продуктивність завдяки компіляції в нативний код, але вимагає вивчення нової мови програмування Dart. React Native використовує JavaScript та JSX для опису інтерфейсу, Vue Native використовує знайомий синтаксис Vue.js, а Flutter використовує мову Dart та віджети для створення інтерфейсів. Кожен з цих фреймворків має свої сильні та слабкі сторони, залежно від вимог проекту. Порівняння коду, синтаксису та структури показує, що React Native підходить для тих, хто вже має досвід з React, Vue Native – для розробників, які працюють з Vue.js, а Flutter забезпечує високий рівень продуктивності і багаті можливості для створення інтерфейсів, але вимагає вивчення мови Dart.

Отже, сучасний ландшафт мобільної розробки є багатогранним і динамічним, пропонуючи розробникам широкий спектр інструментів та технологій для створення високоякісних мобільних застосунків. Незалежно від обраного підходу, головною залишається здатність адаптуватися до швидко змінюваного технологічного середовища, забезпечуючи користувачам швидкий, безпечний і функціональний доступ до інформації та сервісів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Niel T. Mobile Design Pattern Gallery: UI Patterns for Smartphone Apps, 2nd Edition. NY: O'Reilly, 2014. 404c.
2. Coron T. Apple Game Frameworks and Technologies: Build 2D Games with SpriteKit & Swift. The Pragmatic Programmers, 2021. 504c. (дата звернення 23.05.2024)
3. Core Components and APIs. URL: https://reactnative.dev/docs/components-and-apis (дата звернення 23.05.2024).
4. Flutter documentation. URL: https://docs.flutter.dev (дата звернення 23.05.2024).
5. Introduction. URL: https://nativescript-vue.org/en/docs/introduction/ (дата звернення 23.05.2024).
6. SwiftUI. URL: https://developer.apple.com/xcode/swiftui/ (дата звернення 23.05.2024).
7. Build better apps faster with Jetpack Compose URL: https://developer.android.com/develop/ui/compose (дата звернення 23.05.2024).
8. Data flow у SwiftUI. Або чому не все так просто, як здається. URL: https://dou.ua/forums/topic/44036/ (дата звернення 23.05.2024).
9. What Is React Native? Complex Guide for 2024. URL: https://www.netguru.com/glossary/react-native (дата звернення 23.05.2024).
10. Measure and improve performance with Macrobenchmark. URL: https://medium.com/androiddevelopers/measure-and-improve-performance-with-macrobenchmark-560abd0aa5bb (дата звернення 23.05.2024).
11. The Good and the Bad of Flutter App Development. URL: https://www.altexsoft.com/blog/pros-and-cons-of-flutter-app-development/ (дата звернення 23.05.2024).
12. Stack Overflow (укр. Композиція Flutter) URL: https://stackoverflow.com/ (дата звернення 23.05.2024).
13. Statista. URL: https://www.statista.com (дата звернення 23.05.2024).

14. Capture Macrobenchmark metrics. URL: https://developer.android.com/topic/performance/benchmarking/macrobenchmark-metrics (дата звернення 23.05.2024).
15. Architecting your Compose UI. URL: https://developer.android.com/develop/ui/compose/architecture (дата звернення 23.05.2024).
16. Jetpack Compose architectural layering. URL: https://developer.android.com/develop/ui/compose/layering (дата звернення 23.05.2024).
17. Jetpack Compose vs XML: A comprehensive comparison for Android UI development. URL: https://www.auberginesolutions.com/blog/jetpack-compose-vs-xml-a-comprehensive-comparison-for-android-ui-development/ (дата звернення 23.05.2024).