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 *Ситника Романа Сергійовича*
(ПІБ)

академічної групи *121М-19-1*
(шифр)

спеціальності *121 Інженерія програмного забезпечення*
(код і назва спеціальності)

на тему: *Розробка програмного забезпечення для дослідження ефективності засобу Flutter при розробці мобільних додатків*

_____ *Р.С. Ситник*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
розділів кваліфікаційної роботи				
спеціальний	Проф. Алексєєв М. О.			
економічний	Доц. Касьяненко Л.В.			

Рецензент				

Нормоконтролер	Доц. Сироткіна О.І.			

Дніпро
2020

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

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)
_____ І.М. Удовик
(підпис) (прізвище, ініціали)
« » _____ 20 20 Року

ЗАВДАННЯ
на виконання кваліфікаційної роботи магістра

спеціальності *121 Інженерія програмного забезпечення*
(код і назва спеціальності)

студенту *121М-19-1* *Ситнику Р. С.*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи *Розробка програмного забезпечення для дослідження ефективності засобу Flutter при розробці мобільних додатків*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 22.10.2020 р. № 888 -с

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – процес розробки мобільних додатків для операційних систем iOS та Android та практичне використання засобу Flutter з метою підвищення ефективності розробки мобільних додатків.

Предмет досліджень – методи та підходи засобу Flutter у розробці програмного забезпечення для мобільних пристроїв.

Мета роботи – вдосконалення та пришвидшення процесу розробки якісних мобільних додатків на замовлення бізнесу під популярні мобільні операційні системи за допомогою засобу Flutter.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна полягає у тому, що вдосконалено методи визначення ефективності мобільних додатків з використанням засобу Flutter, отримали подальший розвиток процеси ефективної розробки мобільних додатків за допомогою засобу Flutter та декларативного підходу до побудови мобільних інтерфейсів.

Практична цінність полягає у тому, що в результаті роботи, було створено прототип бібліотеки, яка дозволяє визначати ефективність розроблених кросплатформенних мобільних додатків засобом Flutter під операційними системами iOS та Android. Використані методи та підходи можуть застосовуватися при розробці програмного забезпечення під мобільні пристрої на замовлення бізнесу, щоб розробка мобільних додатків проходила швидшими та ефективнішими методами.

4 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець**
Збір інформації для дослідження предметної області	01.09.2020 – 03.10.2020
Дослідження методів для вирішення поставленого завдання	03.10.2020 – 10.11.2020
Експериментальні дослідження	10.11.2020 – 28.11.2020
Економічна частина	28.11.2020 – 01.12.2020

Завдання видав _____ *Алексєєв М.О.*
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання _____ *Ситник Р.С.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *01.09.2020 р.*

Термін подання кваліфікаційної роботи до ЕК *03.12.2020 р.*

РЕФЕРАТ

Пояснювальна записка: 105 стор., 32 рис., 3 додатки, 75 джерел.

Об’єкт досліджень: процес розробки мобільних додатків для операційних систем iOS та Android та практичне використання засобу Flutter з метою підвищення ефективності розробки мобільних додатків.

Предмет досліджень: методи та підходи засобу Flutter у розробці програмного забезпечення для мобільних пристроїв.

Мета магістерської роботи: вдосконалення та пришвидшення процесу розробки якісних мобільних додатків на замовлення бізнесу під популярні мобільні операційні системи за допомогою засобу Flutter.

Методи дослідження: теорія системного аналізу, принципи об’єктно-орієнтованого програмування, принципи декларативного програмування.

Наукова новизна: вдосконалено методи визначення ефективності мобільних додатків з використанням засобу Flutter, отримали подальший розвиток процеси ефективної розробки мобільних додатків за допомогою засобу Flutter та декларативного підходу до побудови мобільних інтерфейсів.

Практична цінність результатів полягає у тому, що в результаті роботи, було створено прототип бібліотеки, яка дозволяє визначати ефективність розроблених кросплатформенних мобільних додатків засобом Flutter під операційними системами iOS та Android. Використані методи та підходи можуть застосовуватися при розробці програмного забезпечення під мобільні пристрої на замовлення бізнесу, щоб розробка мобільних додатків проходила швидшими та ефективнішими методами.

Це дає можливість розробляти мобільні додатки одразу під обидві платформи з єдиною кодовою базою та з використанням гнучкого декларативного підходу до побудови інтерфейсу.

У розділі «Економіка» проведені розрахунки трудомісткості розробки програмного забезпечення, витрат на створення ПЗ і тривалості його розробки.

Список ключових слів: мобільний додаток, фреймворк, Flutter, Dart, Android, iOS.

ABSTRACT

Explanatory note: 105 pages, 32 images, 3 appendices, 75 sources.

Objects of research: the process of developing mobile applications for iOS and Android operating systems and the practical use of Flutter to increase the efficiency of mobile application development.

Subject of research: Flutter methods and approaches in mobile software development.

Purpose of the master's work: improving and accelerating the process of developing high-quality mobile applications to order business for popular mobile operating systems using Flutter.

Methods of research: theory of systems analysis, principles of object-oriented programming, principles of declarative programming.

Scientific novelty: the methods for determining the effectiveness of mobile applications using the Flutter tool have been improved, the processes of effective development of mobile applications using the Flutter tool and the declarative approach to the construction of mobile interfaces have been further developed.

Practical significance: as a result of the work, a prototype library was created, which allows to determine the effectiveness of the developed cross-platform mobile applications using Flutter under the operating systems iOS and Android. The methods and approaches used can be used in the development of software for mobile devices to order business, so that the development of mobile applications is faster and more efficient methods.

In the section "Economics" the calculations of the complexity of software development, the cost of creating software and the duration of its development.

List of keywords: mobile application, framework, Flutter, Dart, Android, iOS.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД – база даних;

ART – Android Runtime, середовище виконання Android додатків;

DEX – Dalvik Executable, файл з байткодом для Dalvik VM;

HTML – HyperText Markup Language;

CSS – Cascading Style Sheets;

PWA – Progressive Web App, прогресивний вебзастосунок;

AOT – Ahead-of-Time компіляція, компіляція до роботи програми;

FPS – Frames Per Second, кількість кадрів на секунду;

JIT – Just-In-Time компіляція, компіляція під час роботи програми;

UI – User Interface, інтерфейс користувача;

JSX – JavaScript XML;

BLoC – Business Logic Components.

ЗМІСТ

ВСТУП

Актуальність теми. В сучасному світі кількість мобільних девайсів, смартфонів та планшетів, перевищила кількість звичайних персональних комп'ютерів, і кожного року ця різниця зростає ще більше на користь мобільних девайсів.

На сьогодні існує дві основні мобільні операційні системи – Android та iOS, які займають ~74% та ~25% світового ринку відповідно.

Для бізнесу все важливіше надавати своїм клієнтам швидкий доступ до своїх послуг та сервісів за допомогою мобільних додатків. А деякі підприємства навіть стають mobile-first, де мобільні додатки стають основним інструментом взаємодії з клієнтами. Прикладом цього є, наприклад, перший мобільний український банк – monobank, який за кілька років став одним з найпопулярніших банків України, хоча не має жодного відділення та навіть браузерного інтерфейсу, а вся робота з банком відбувається за допомогою додатків для мобільних пристроїв.

Тому мати свій мобільний додаток – вже частіше стає необхідністю для багатьох підприємств. Але розробляти та підтримувати одразу два незалежні один від іншого додатки під кожну популярну мобільну платформу двома різними командами розробників – часами буває занадто дорого та довго для бізнесу.

Для вирішення цієї проблеми існує перелік різноманітних кросплатформних рішень, на кшталт Cordova або React Native, але вони мають ряд вад, які роблять розробку ще більш складнішою, а досвід користування додатком – стає неприємним для користувача. Однією з причин цього є, наприклад, використовування мосту з інтерпретатором JavaScript, або рендеринг браузерних сторінок всередині додатку.

Щоб вирішити цю проблему та мати єдину кодову базу для додатків на різних платформах, компанія Google випустила фреймворк Flutter, який сам

відтворює на екрані кожний піксель інтерфейсу користувача без використання браузеру або посередників з іншими віртуальними машинами.

Мета і задачі дослідження. Метою роботи є вдосконалення та пришвидшення процесу розробки якісних мобільних додатків на замовлення бізнесу під популярні мобільні операційні системи за допомогою засобу Flutter.

Для досягнення мети дослідження необхідно розв’язати наступні задачі:

1. Здійснити аналіз існуючих методів та інструментів, які дозволяють займатися одночасною розробкою під Android та iOS.
2. Проаналізувати нове рішення від Google – фреймворк Flutter як засіб для розробки мобільних додатків.
3. Здійснити порівняння та побудувати структурні моделі усіх можливих рішень для розробки програмного забезпечення для мобільних операційних систем.
4. Розробити плагін для засобу Flutter для визначення ефективності програмного забезпечення для мобільних пристроїв.
5. Зробити висновки щодо найкращих методів та інструментів для розробки мобільного програмного забезпечення.

Об’єктом досліджень є процес розробки мобільних додатків для операційних систем iOS та Android та практичне використання засобу Flutter з метою підвищення ефективності розробки мобільних додатків.

Предметом досліджень є методи та підходи засобу Flutter у розробці програмного забезпечення для мобільних пристроїв.

Методи дослідження. Для розв’язання поставлених завдань використано: теорія системного аналізу, принципи об’єктно-орієнтованого програмування, принципи декларативного програмування.

Наукова новизна роботи:

Наукова новизна полягає у тому, що вдосконалено методи визначення ефективності мобільних додатків з використанням засобу Flutter, отримали подальший розвиток процеси ефективної розробки мобільних додатків за

допомогою засобу Flutter та декларативного підходу до побудови мобільних інтерфейсів.

Практичне значення полягає у тому, що в результаті роботи, було створено прототип бібліотеки, яка дозволяє визначати ефективність та швидкодію розроблених кросплатформенних мобільних додатків засобом Flutter під операційними системами iOS та Android. Використані методи та підходи можуть застосовуватися при розробці програмного забезпечення під мобільні пристрої на замовлення бізнесу, щоб розробка мобільних додатків проходила швидшими та ефективнішими методами.

Особистий внесок автора. Було створено прототип бібліотеки для Flutter, за допомогою якої можна досліджувати ефективність та швидкодію при розробці мобільних додатків для iOS та Android засобами Flutter, також було здійснено дослідження фреймворку Flutter та його порівняння з іншими засобами розробки кросплатформенних та нативних додатків для мобільних пристроїв.

Структура та обсяг кваліфікаційної роботи. Робота складається з вступу, чотирьох розділів і висновків. Містить 105 сторінки друкованого тексту, в тому числі 77 сторінка тексту основної частини з 32 рисунками, списку використаних джерел з 75 найменуваннями на 5 сторінках, 3 додатках на 10 сторінках.

РОЗДІЛ 1
АНАЛІЗ ІНСТРУМЕНТІВ, МЕТОДІВ ТА ЗАСОБІВ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ МОБІЛЬНИХ ПРИСТРОЇВ

1.1. Аналіз поточної ситуації в розробці мобільних додатків

Станом на 2020 рік, на світовому ринку операційних систем для мобільних пристроїв домінують дві основні системи – Android від Google та iOS від Apple (рис 1.1.).

Обидві операційні системи мають різну структуру, ядра, будову ОС та різні підходи до розробки програмного забезпечення під них, та не мають підтримки додатків для іншої платформи [1].

Рис. 1.1. Логотипи iOS та Android.

Тобто, для розробки мобільного додатку, який б охоплював увесь ринок мобільних операційних систем, треба розробити дві окремі програми за допомогою різних мов програмування та різних SDK для кожної платформи окремо.

І таким чином в індустрії мобільного програмного забезпечення так склалося, що для розробки концептуально однакового додатку треба дві різні команди розробників, які розробляють однаковий функціонал за допомогою різних мов, підходів та SDK. А професія “мобільного розробника” поділяється на дві незалежні одна від одної спеціалізації – Android-розробника та iOS-розробника, і вони майже не переплітаються між собою [2].

Такий підхід вже став класичним у світі мобільної розробки, але невеликому бізнесу та організаціям може бути доволі дорого фінансувати дві окремі команди розробників, їхній менеджмент та тестування. І доволі часто це є завадою для запуску мобільних додатків невеликому бізнесу та організаціям.

Також це створює перепони під час розробки і великому та середньому бізнесу. Іноді одна версія додатку для однієї операційної системи створюється швидше, ніж інша версія для другої ОС, і замовникам доводиться чекати поки одна з команд зможе наздогнати іншу, щоб випустити нову версію програмного забезпечення для усіх платформ одночасно.

З цією проблемою регулярно стискаються, наприклад, крупні банки, на кшталт ПриватБанку та monobank. Версії їхніх мобільних додатків для різних операційних систем можуть виходити у різний час, що може викликати складнощі як у користувачів, так і бізнесу.

1.1.1. Нативні методи та засоби розробки мобільного ПЗ

Операційна система Android написана мовою програмування C та C++, і також використовує Linux в якості системного ядра ОС. Але для того, щоб програмне забезпечення не залежало від архітектури процесору та іншого апартного забезпечення, для запуску прикладних програм використовувалася віртуальна машина Dalvik, а починаючи від Android 5.0 – середа виконання Android Runtime (ART) [3].

Обидві середи виконання запускають прикладне програмне забезпечення для мобільної операційної системи Android у вигляді спеціального байт-коду у форматі DEX, який є подібним на байт-код віртуальної машини Java (рис 1.2).

Трансляцію у байт-код формата DEX забезпечують такі мови програмування, як Java та мова програмування Kotlin.

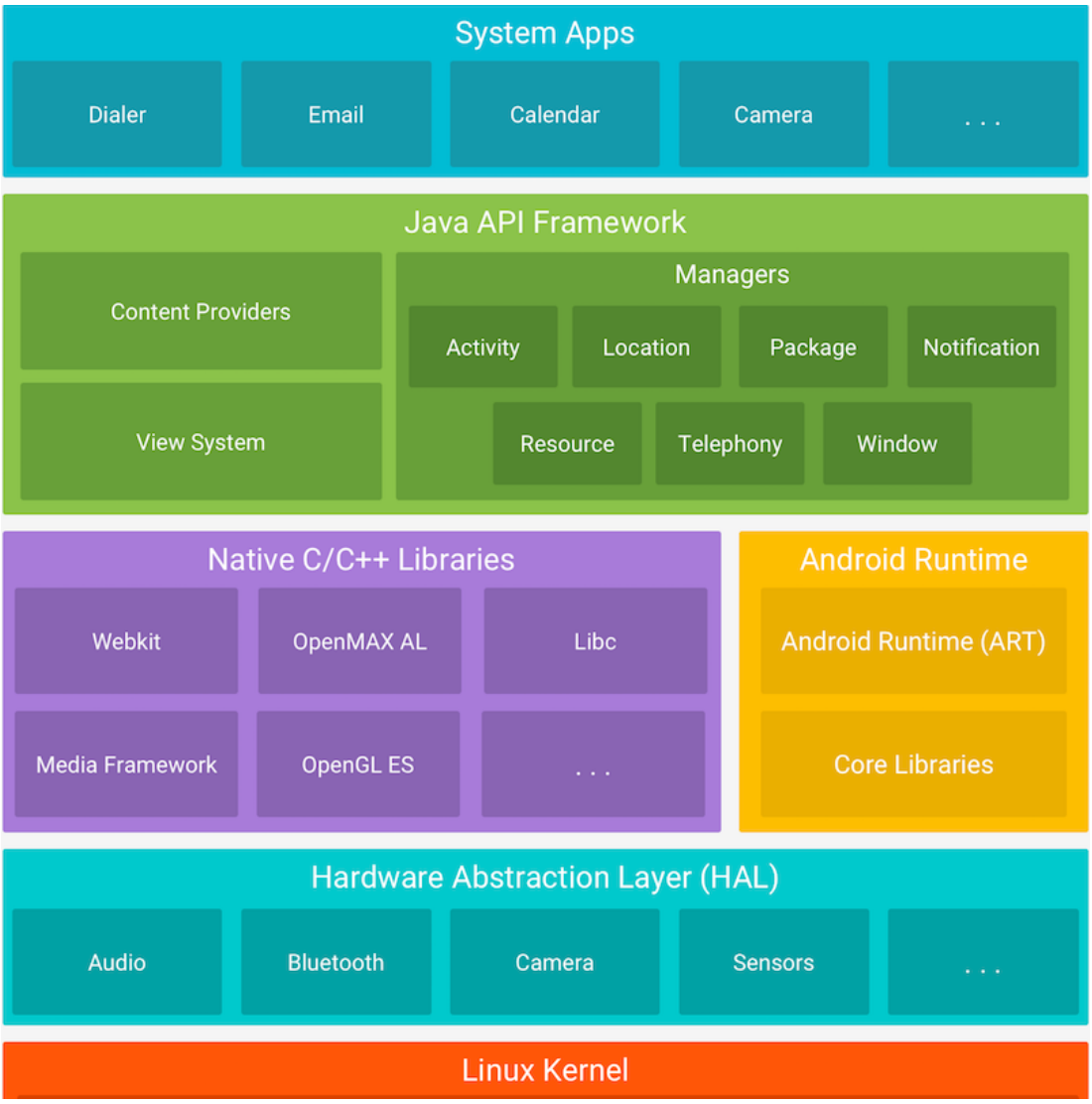

Рис. 1.2. Архітектура ОС Android.

Java довгий час використовувалася як основна мова програмування під Андроїд, але починаючи від 2017 року все більш популярною мовою для розробки додатків ставала Kotlin, яку розробив JetBrains.

А вже у 2019 компанія Google під час своєї конференції Google I/O 2019 об’явила мову програмування Kotlin основною мовою для розробки для Android,

і що всі нові бібліотеки, фреймворки та інструкції будуть виходити в першу чергу мовою Kotlin [4].

Але перехід з однієї мови програмування на іншу є доволі довгим процесом, і так склалося, що від більшості Android-розробників на сьогоднішній день вимагають знання одразу двох мов – і Java, і Kotlin, що підвищує поріг входу до цієї професії.

На сьогоднішній день більшість більшість Android-розробників використовують мову програмування Kotlin [5].

Подібна ситуація склалася і з операційною системою iOS від Apple.

iOS також написана мовою програмування C, C++ та Objective-C, а в якості ядра використовується власною UNIX-подібною розробкою Apple – ядром під назвою Darwin, яка складається з компонентів NeXTSTEP та FreeBSD (рис 1.3.) [6].

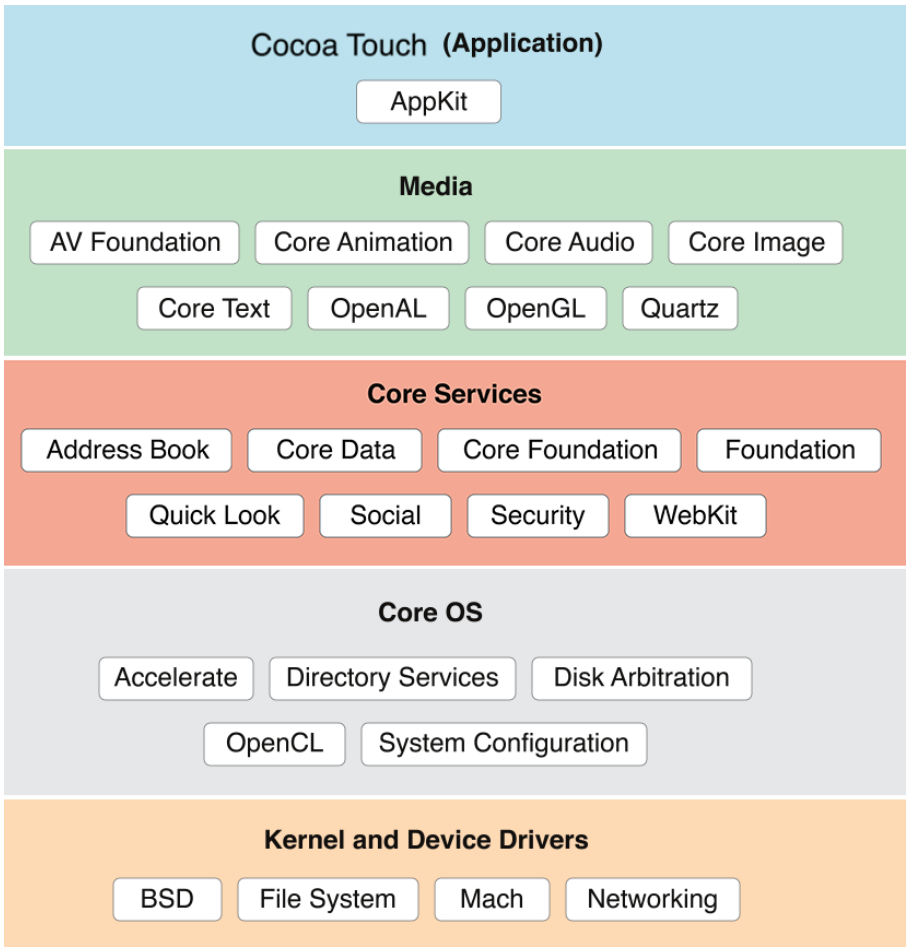

Рис. 1.3. Архітектура iOS.

Довгий час основною мовою програмування додатків для операційних систем Apple вважалася мова Objective-C, яку також розробила компанія Apple у 1986 році.

Objective-C є розширенням мови C з додаванням ООП, але з підходом, який відрізняється від ООП у мові C++.

ООП в Objective-C включає інтерфейси, класи та категорії. У мові було реалізовано одиничне невіртуальне наслідування, на відміну від C++ [7].

Але розробляти мобільні додатки для iOS цією мовою було доволі складним та довгим процесом. Розробники мобільних додатків були змушені самостійно дбати про використання пам’яті, працювати з вказівками (pointers) тощо.

І тому у 2014 році компанія Apple представила нову основну мову для програмування для iOS – Swift.

Swift успадковує найкращі принципи мови Objective-C, тому синтаксис цієї мови є звичним для розробників, які в минулому використовували Objective-C, але водночас відрізняється використанням засобів автоматичного розподілу пам'яті і контролю переповнення змінних і масивів, що значно збільшує надійність і безпеку коду [8].

Також Swift-додатки компілюються у машинний код операційної системи від Apple, що дозволяє забезпечити високу швидкість роботи програмного забезпечення, написанного цією мовою.

В якості інструментарію, Android-розробники використовують Android Studio, в основі якої лежить IntelliJ IDEA від JetBrains, та Android SDK. iOS-розробники використовують фірменну середу розробки Xcode від Apple, яка працює тільки під управлінням операційної системи macOS [9].

1.1.2. Кросплатформні методи та засоби розробки мобільного ПЗ

Як альтернатива нативній розробці під iOS та Android рідними мовами, деякі компанії почали розробляти свої власні рішення, які б дозволяли писати

програмне забезпечення одразу під обидві платформи за допомогою однієї мови програмування.

Майже всі такі рішення засновані на використанні якогось третього сполучника, який дозволяє під'єднати нативні біндинги мобільної операційної системи (мовами Kotlin та Swift) до іншої мови програмування або фреймворку, та таким чином дозволити розробляти додатки за допомогою інших мов та інструментів, які будуть працювати поверх рідного коду ОС [10].

1.1.3. Розробка додатків за допомогою браузерних технологій

Перші кросплатформні фреймворки, на кшталт Apache Cordova, почали використовувати в якості рушія вбудований елемент вікна браузеру (WebView), поверх якого і працює весь кросплатформений додаток (рис 1.4.).

Таке програмне забезпечення розробляється тими ж самим інструментами, що і звичайні браузерні вебсторінки: HTML, CSS та JavaScript, і потім відтворюється за допомогою елементу вікна браузеру (WebView), який модифікований додатковими інструментами та біндингами від Apache Cordova [11].

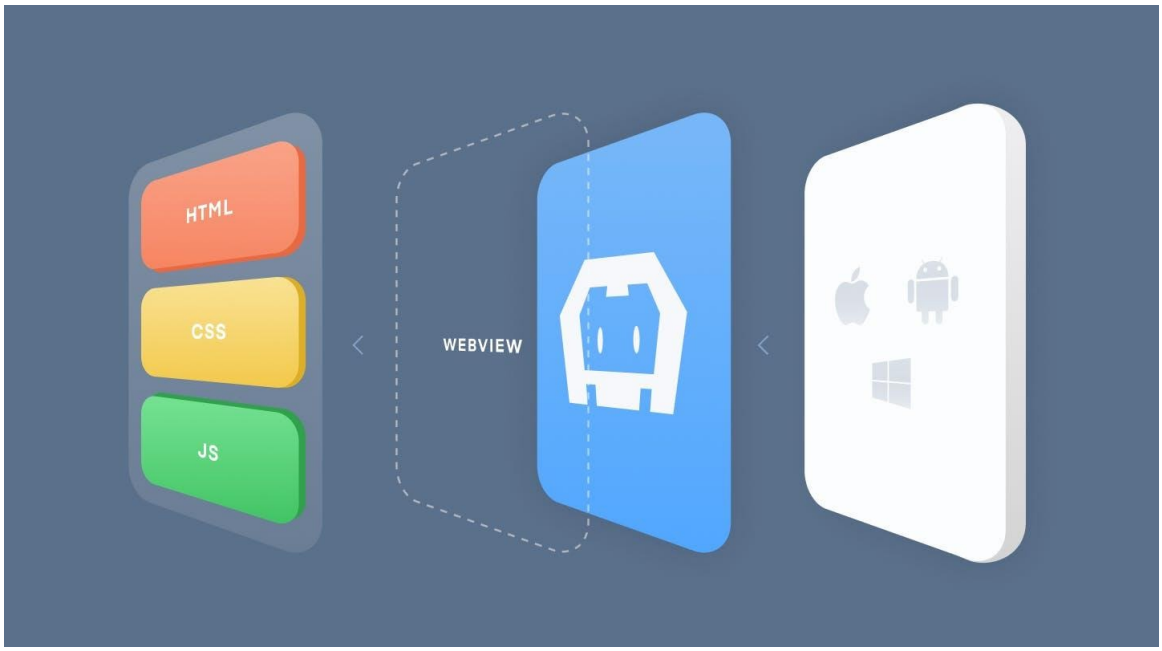

Рис. 1.4. Принцип роботи Apache Cordova.

Це дозволяє залучити до розробки програмістів, які мають досвід роботи з вебтехнологіями і розробляти додатки одночасно під обидві платформи, використовуючи браузерні технології всередині програми. Також це допомагає перевикористати якісь готові елементи вебсайтів під час створення мобільного додатку.

Але цей підхід має свої дуже серйозні вади:

1. Швидкодія написаних таким чином додатків є дуже низькою. Загальна повільність всього інтерфейсу може відчуватися навіть на потужних девайсах. При побудові ж складних та комплексних інтерфейсів сильніше уповільнення швидкодії стає все більш помітним, що знижує загальний досвід користувача (User Experience) та заважає працювати з програмами [12].
2. Зовнішній вигляд таких додатків доволі складно пристосувати до справжнього мобільного вигляду, і він більше схожий на браузерний (яким він насправді і є зсередини).

Це може завадити в побудові гарних користувальницьких інтерфейсів та може призвести до негативної реакції користувачів через вади UI-дизайну [13].

Найбільше такий підхід до розробки мобільних додатків підходить для розробки дуже маленьких та нескладних мобільних додатків, наприклад програми-довідника або невеликої інформаційної панелі.

За схожим принципом працюють і прогресивні вебдодатки (Progressive Web Applications, PWA) (рис 1.5.).

Це вебові додатки (вебсайти), які є гібридом звичайної Інтернет-сторінки браузеру та мобільного додатку.

Вони створюється за допомогою інструментів, які надають сучасні мобільні браузери – Service Workers, Manifests, кешування скриптів та сторінок тощо.

Рис. 1.5. Логотип технології PWA.

Технологія PWA дає можливість зробити офлайн кеш вебкомпонентів та скриптів сайту, виконуючи їхній код за допомогою спеціального сервісу Service Worker навіть в офлайн режимі, без використання мережі, та надати цьому вигляд звичайного мобільного додатку [14].

Але цей підхід має приблизно ті ж самі вади, що і мобільні додатки на основі WebView: повільна швидкість роботи, повільний запуск, обмежений розмір додатку (до 10 мб), а очистка кешу мобільного браузеру призведе до неможливості роботи PWA-додатку в офлайн-режимі. Також, прогресивні вебдодатки погано співпрацюють з основним браузером iOS – Safari від Apple, і потребують використання інших браузерів, які встановлені за замовчуванням на операційній системі від Apple [15].

Технологія не отримала широкого використання за роки її існування, але продовжує розвиватися та поліпшувати якість роботи. Наприклад, компанія Twitter надає можливість завантажити PWA-додаток на телефон, якщо зайти на їхній ресурс через браузер. Але більшість користувачів все ще віддають перевагу більш функціональному та швидкому додатку з магазину додатків AppStore та Google Play.

1.1.4. Розробка кросплатформних додатків іншими мовами програмування за допомогою посередників та мостів

Наступним етапом еволюції кросплатформних додатків стало впровадження повноцінних мов програмування до розробки мобільних додатків без використання рендерингу через браузерні рушії.

Для реалізації цього підходу, розробники фреймворків почали впроваджувати архітектуру на основі шаблону «посередник» (або ж, як позначається цей посередник в документації фреймворків – «міст»), який зв'язує виклики нативного API мобільних операційних систем з командами іншої мови програмування та віртуальної машини, які використовуються для виконання коду кросплатформного додатку.

Посередник – це шаблон проєктування програмного забезпечення, який дозволяє визначити об'єкт, який взаємодіє з групою інших об'єктів за заданою поведінкою.

Це працює таким чином: код додатка, написаний ненативною мовою, виконується у віртуальній машині або інтерпретаторі, який поставляється разом з кросплатформним додатком. Це виконання відбувається в окремому потоці роботи додатку. Після цієї обробки команди віддаються через посередника (мосту) до нативних біндингів мобільної операційної системи, які вже вконуються у головному потоці (main thread) додатку, який відповідає за відтворення інтерфейсу користувача та обробку команд від користувача [16].

Цей підхід дозволяє писати один код для обох платформ одночасно, виконуючи його у віртуальній машині, та передаючи команди до ОС через «міст-посередник».

І вже ці трансльовані команди до ОС будують інтерфейс і описують поведінку додатку, що зовні не відрізняється від нативних реалізацій.

Однією з успішних таких розробок презентувала компанія Facebook у 2015, фреймворк – React Native, який базується на принципах роботи бібліотеки для створення вебдодатків React.JS [17].

Підхід цього фреймворку для розробки мобільних додатків сильно відрізняється від підходів до розробки додатків, які використовуються у фреймворках на основі рендерингу на браузерному рушії.

React Native дає можливість розробляти програмне забезпечення для iOS та Android за допомогою динамічної мови програмування JavaScript та бібліотеки React.

Але у відмінності від браузерних рушіїв, React Native відтворює інтерфейс користувача за допомогою рідних компонентів операційної системи. Тобто, наприклад, форма з полем вводу та кнопка будуть справжніми нативними елементами UI, а не будуть пародіювати їх, як у Corvoda або PWA (рис 1.6.).

Вигляд елементів інтерфейсів також буде різний для iOS та Android, згідно з дизайн-системами цих ОС [18].

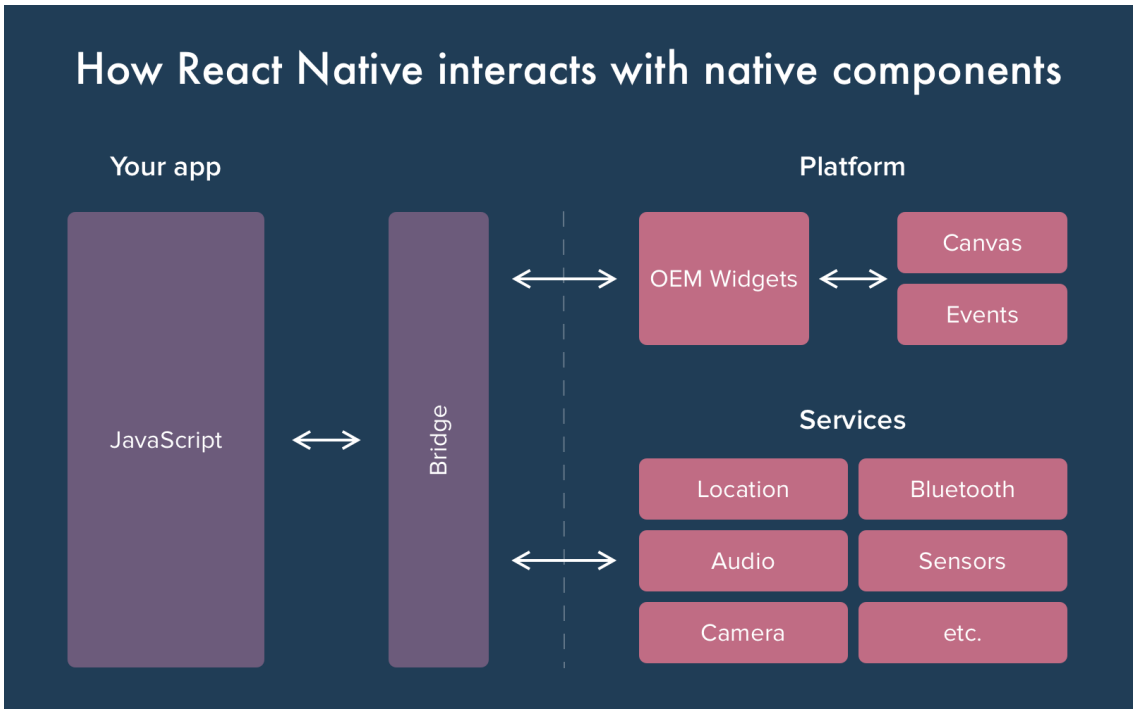

Рис. 1.6. Архітектура React Native.

Для реалізації цього підходу React Native використовує рушій для інтерпретації JavaScript та спеціальний міст, який передає команди від рушія мови до системних команд рідними мовами системи (Kotlin, Swift, Java, Objective-C), які вже будують інтерфейс та задають поведінку елементів.

Така архітектура дозволяє писати увесь код мобільного додатку на JavaScript та використовувати бібліотеки для цієї мови. Цей підхід приближає розробку мобільного програмного забезпечення до розробки вебдодатків для браузеру.

Але цей підхід також має доволі багато недоліків.

Швидкодія таких додатків стає набагато кращою за швидкодію програм, які використовують браузерний рушій для рендерінгу та роботи всього додатку, але все одно цей підхід створює додаткові перепони на шляху до забезпечення плавної та швидкої роботи мобільного програмного забезпечення.

Кожен мобільний додаток, який використовує React Native, запускає у фоні віртуальну машину для обробки коду додатка, який написаний на JavaScript. Але ще більше сповільняє роботу цих додатків міст – RN Bridge, який є відповідальним за взаємодію коду на JavaScript з усією операційною системою та девайсом [19].

Через це такі додатки вимагають багато оперативної пам’яті, займають багато пам’яті на диску (кожен мобільний додаток встановлюється разом з віртуальною машиною та фреймворком), та сповільнює обробку команд через використання спеціального мосту в якості посередника.

За доволі схожим принципом працюють інші, менш популярні фреймворки для розробки кросплатформних мобільних додатків, наприклад, Xamarin.

Фреймворк Xamarin, яким зараз володіє компанія Microsoft, дозволяє вести розробку мовою C#, використовуючи вільну реалізацію стандартів .NET (CLR, CLI та інших інтерфейсів) і дозволяє створювати додатки на iOS та Android [20].

Але він має ті ж самі проблеми зі швидкодією, що і React Native.

Xamarin використовує спеціальний міст, який зв’язує нативне API операційних систем з кодом на C#.

Крім того, також у Xamarin існують проблеми з підтримкою фреймворку товариством розробників, своєчасними оновленнями, зручними інструментами для розробки та іншим.

Процес розробки та поліпшення цього фреймворку є періодичним та непостійним через те, що компанія Microsoft більше уваги акцентує розвитку мови C# та .Net в інших напрямках: .Net Core, Unity, Win.Forms, ASP.NET та інших технологіях, які розвиваються паралельно з Xamarin (рис 1.7.).

Також, мова C# та середа розробки Visual Studio мають вищий поріг входу, аніж JavaScript, тому вивчення фреймворку Xamarin може займати багато часу у новачків, які не працювали з інструментами та фреймворками для розробки від Microsoft раніше [21].

Тому Xamarin не зазнав великого поширення за межами компаній, які спеціалізуються на .NET розробці, та широко використовують інструменти Microsoft для розробників.

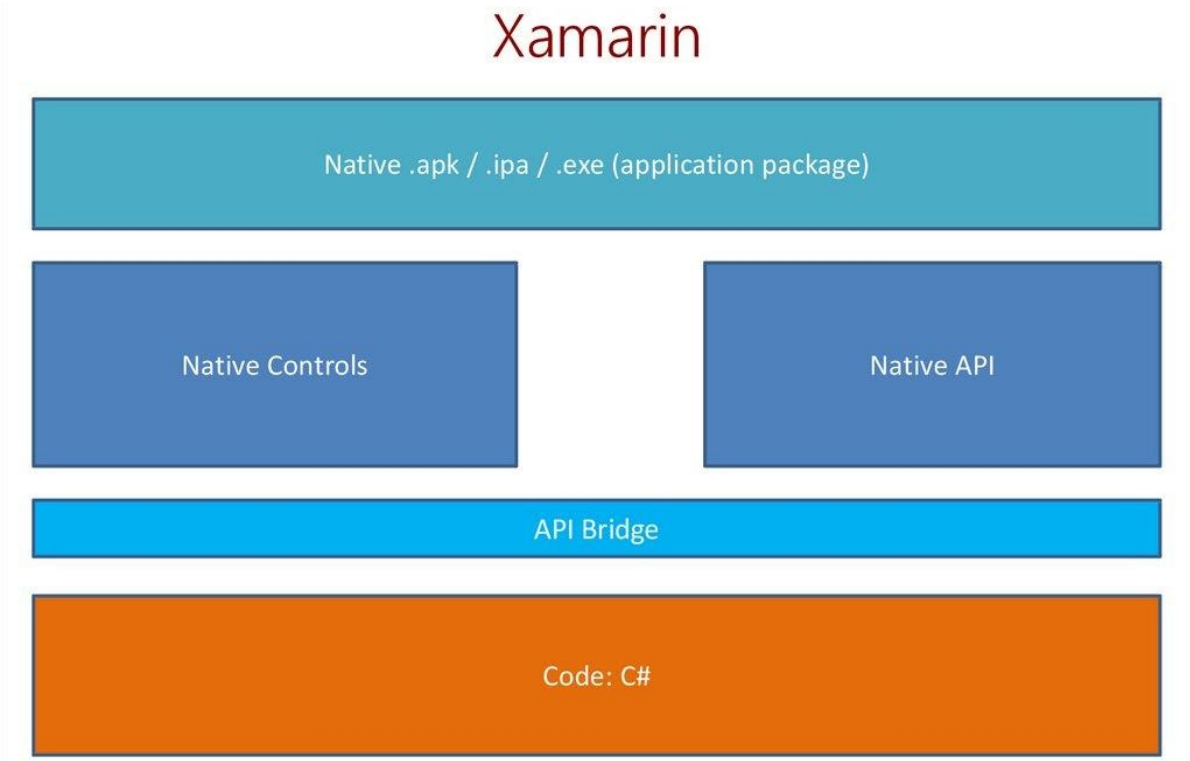

Рис. 1.7. Архітектура Xamarin.

1.1.5. Розробка кросплатформних додатків іншими мовами програмування без використання посередників та мостів

У 2018 році компанія Google представила новий інструмент для розробки кросплатформного програмного забезпечення, який не використовує ані посередників та мостів, ані браузерних технологій – фреймворк Flutter (рис 1.8.).

Рис. 1.8. Логотип Flutter.

Google представила зовсім інший підхід до побудови таких додатків. Flutter не використовує системні компоненти мобільних операційних систем (такі як кнопки, текстові поля та інші), а повністю сам займається відтворенням кожного пікселя на екрані. Тобто, фреймворк Flutter використовує свій рушій для рендерингу інтерфейсу користувача [22].

Такий підхід робить набагато простішим використання UI-елементів, які поводяться однаково на всіх платформах, на відміну від фреймворків на кшталт React Native, де доволі часто доводиться дописувати поведінку та/або стилі візуальних елементів для кожної операційної системи окремо, бо на кожній платформі використовується свій біндинг для візуальних елементів.

Також Flutter не використовує посередників для роботи додатків, а компілюється за допомогою Ahead-of-Time (AOT) компіляції у нативні бібліотеки ARM, x86 та інші [23].

Під час запуску на системі, додаток підгружає цю нативну бібліотеку Flutter, та починає використовувати її для відтворення програми.

Ahead-of-Time (AOT) – це вид транслятора, який використовує метод компіляції перед виконанням програми. Метод AOT не вимагає виділення додаткової пам'яті і AOT-компіляція проходить з мінімальним навантаженням на систему. Процес компіляції повністю виконується перед виконанням програми.

В якості основної мови програмування Flutter використовує мову Dart, який працює на віртуальній машині DartVM, яка представлена як система виконання Dart-коду всередині предкомпільованих нативних бібліотек. Тобто, ця середа виконання Dart є частиною нативної бібліотеки, в яку компілюється Flutter [24].

Цей підхід дозволяє створювати швидкі додатки, які здатні без додаткових налаштувань відтворюватися зі швидкістю 60 FPS (кадрів на секунду) на усіх девайсах на системах Android та iOS. Також тривають роботи щодо впровадження підтримки Flutter для створення вебдодатків та програмного забезпечення для Windows, macOS та Linux.

Візуалізація зображення на екрані забезпечується апаратним забезпеченням (дисплеєм), яке регулярно (зазвичай 60 раз в секунду, але існують сучасні екрани, які оновлюються 120 раз на секунду) оновлює дисплей. Це називається "частотою оновлення" і виражається в Гц (Герцах).

Дисплей отримує інформацію для відображення від GPU (Graphics Processing Unit), що представляє собою спеціалізовану електронну схему, оптимізовану і призначену для швидкого формування зображення з деяких даних (полігонів і текстур).

Кількість разів в секунду, які графічний процесор може генерувати "зображення" (=буфер кадрів) для відображення і відправки його на апаратне забезпечення, називається кадровою частотою (прим: frame rate). Це вимірюється за допомогою блоку кадрів в секунду (наприклад, 60 кадрів в секунду або 60 fps).

Разом з фреймворком, Flutter надає велику вбудовану бібліотеку віджетів (UI-елементів), анімацій, жестів тощо. Також підтримуються різноманітні сторонні бібліотеки.

Але Flutter також має кілька недоліків.

Однією з вад Flutter є те, що інтерфейси, які побудовані на ньому, виглядають приблизно однаково на всіх платформах, якщо під час розробки програмного забезпечення використовувати один набір віджетів (візуальних елементів).

Це заважає створювати унікальні інтерфейси для кожної системи окремо, згідно з їхніх дизайн-патернів. Але цю проблему можна вирішити за допомогою використання додаткового шару абстракції для UI-віджетів, який буде відтворювати для різних платформ різний візуальний вигляд елементів. Але, на відміну від React Native, ці компоненти будуть мати єдині стандарти поведінки, і вони не будуть потребувати якихось додаткових налаштувань.

Інша проблема Flutter – використання доволі непопулярної мови програмування Dart, яка до появи цього фреймворку не була широко представлена на ринку. Хоча мова Dart є доволі схожою на інші C-подібні мови програмування, на кшталт Java, Kotlin, C++ або Swift, тому вона не потребує багато часу на опанування, якщо розробник раніше вже програмував на будь-якій C-подібній мові.

Так, згідно з даних Google Trends, з листопада 2019 року, користувачі почали шукати Flutter частіше за React Native [25].

1.2. Висновки до першого розділу

В цьому розділі були проаналізовані дві найбільші мобільні операційні системи – Android та iOS, та методи за якими можливо проводити розробку програмного забезпечення для цих ОС.

Були розібрані такі підходи до розробки, як: нативна розробка під кожну ОС окремо, розробка за допомогою рішень на основі браузерного рушія, а також засобами React Native та Flutter.

На сьогоднішній день Flutter стрімко розвивається, та починає випереджати найближчого конкурента, який був на ринку кросплатформних до появи цього фреймворку – React Native.

РОЗДІЛ 2
АНАЛІЗ ТА РОЗБІР РОБОТИ ФРЕЙМВОРКІВ FLUTTER ТА REACT NATIVE

2.1. Аналіз методів роботи фреймворку Flutter

Flutter – молода, але дуже перспективна платформа. Вона цікава насамперед своєю простотою розробки і швидкістю роботи на рівні з нативними додатками для Android та iOS.

Висока продуктивність програм і швидкість розробки досягається за рахунок декількох особливостей цього фреймворку:

- На відміну від багатьох відомих на сьогоднішній день мобільних платформ, Flutter не використовує JavaScript ні в якому вигляді. В якості мови програмування для Flutter був обраний Dart, який компілюється в бінарний код, за рахунок чого досягається швидкість виконання операцій порівняна з Objective-C, Swift, Java або Kotlin [26].
- Flutter не використовує нативні компоненти ні в якому вигляді, що дозволяє не писати ніяких прошарків для комунікації з ними (на відміну від того ж React Native). Замість цього, подібно до ігрових двигунів, Flutter відтворює весь інтерфейс самостійно. Кнопки, текст, медіа-елементи, фон – все це відтворюється всередині графічного двигуна в самому Flutter. Після вищесказаного варто відзначити, що "Hello World" додаток на Flutter займає зовсім небагато місця: iOS ≈ 2.5Mb і Android ≈ 4Mb [27].
- Для побудови UI під Flutter використовується декларативний підхід, натхненний вебфреймворком ReactJS, на основі віджетів (які також іноді називають “компонентами”). Для ще більшого приросту в швидкості роботи інтерфейсу віджети перемальовуються по необхідності – тільки коли в них щось змінилося, подібно до того як це робить Virtual DOM в світі вебдодатків [28].

Dart – це мова програмування загального призначення від компанії Google, яка призначена насамперед для розробки клієнтських додатків (вебдодатків, мобільних додатків), хоча на ньому можна розробляти і серверну частину. Це також означає, що одну і ту ж програму на Dart можна компілювати під різні платформи – Windows (x86/64), Android, iOS, macOS.

Dart – це також об'єктно-орієнтована мова програмування. Всі значення, які використовуються в програмі на Dart, представляють собою об'єкти. У своєму розвитку Dart зазнав впливу більш ранніх мов, таких як Smalltalk, Java, JavaScript. Його синтаксис схожий на синтаксис інших C-подібних мов [29].

У Windows, macOS та Linux Flutter працює у віртуальній машині Dart, яка має Just-In-Time (JIT) механізм виконання. Під час написання та налагодження програми Flutter використовує компіляцію Just-In-Time (JIT), дозволяючи виконувати "гаряче перезавантаження" ("Hot Reload"), за допомогою якої модифікації вихідних файлів можуть бути введені в запущений додаток. В більшості випадків зміни вихідного коду негайно відображаються у запущеному додатку, не вимагаючи перезапуску або втрати стану [30].

Релізні версії додатків Flutter (ті, що встановлюються на пристрої користувачів з магазину додатків) компілюються із достроковою компіляцією (Ahead-Of-Time, AOT) на Android і iOS, що робить можливим високу продуктивність Flutter на мобільних пристроях [31].

Рушій Flutter, написаний головним чином на C++, забезпечує підтримку низькорівневого візуалізації за допомогою графічної бібліотеки Skia від Google [32].

Крім того, він взаємодіє зі специфічними для платформи SDK, такими як Android та iOS. Рушій Flutter – це портативна середа виконання для програм, написаних на Flutter та Dart. Він реалізує основні бібліотеки Flutter, включаючи анімацію та графіку, файлові та мережеві введення-виведення, підтримку доступності, архітектуру плагінів, а також середовище виконання та компіляції Dart. Більшість розробників взаємодіють з Flutter за допомогою Flutter

Framework, який забезпечує реактивну структуру та набір віджетів платформи [33].

Фреймворк Flutter містить два набори віджетів, які відповідають певним мовам дизайну: віджети Material Design, що реалізують однойменну дизайн-систему від Google, та віджети Cupertino – вказівки щодо інтерфейсу від Apple.

Як правило, розробники взаємодіють з Flutter через Flutter Framework, який забезпечує сучасну реактивну структуру, написану мовою Dart. Він включає багатий набір бібліотек платформи, верстки та основоположних бібліотек, що складається з низки шарів. Знизу вгору:

Шар рендерингу забезпечує абстракцію для роботи з дизайном. За допомогою цього шару можна побудувати дерево об'єктів, що відображаються на екрані.

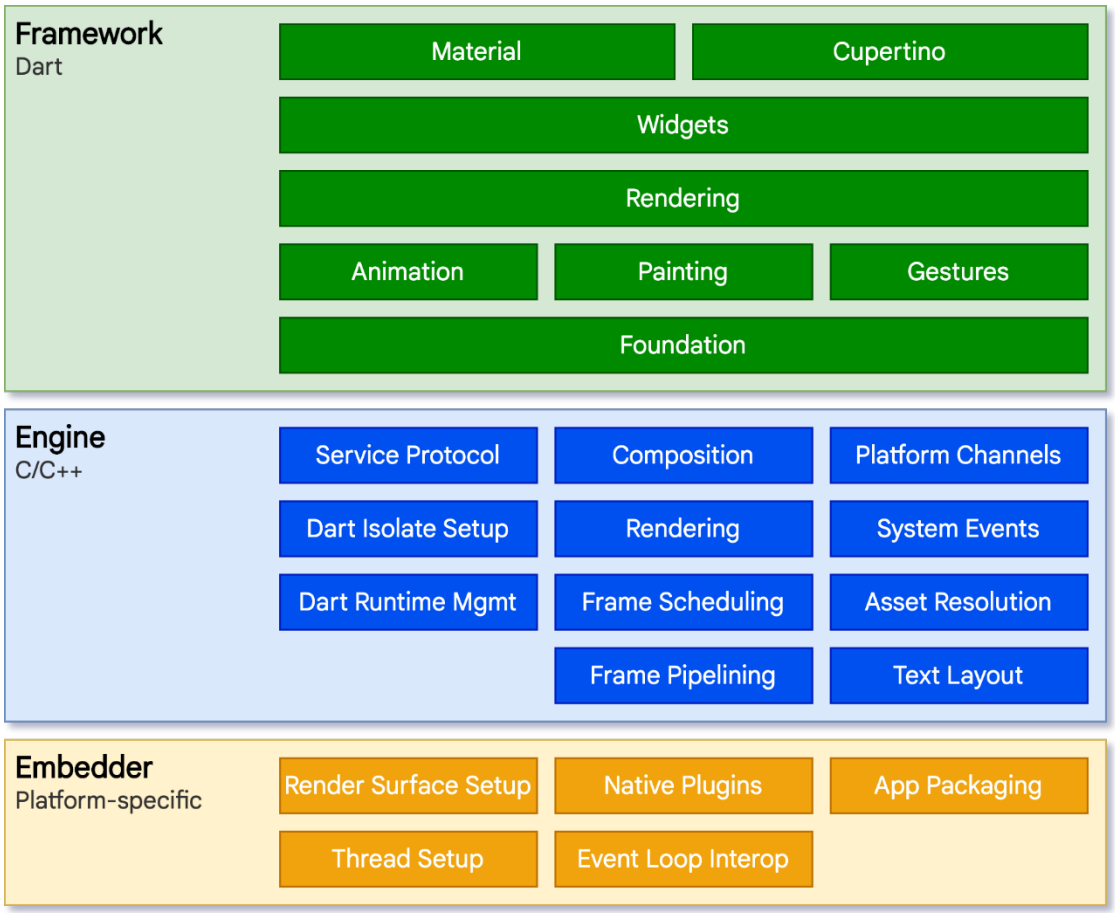

Рис. 2.1. Архітектура Flutter.

Коли розробник пише додаток на Flutter, використовуючи мову Dart, він залишається на рівні Flutter Framework (виділено зеленим кольором на рис. 2.1.).

Flutter Framework взаємодіє з Flutter Engine (синім кольором на рис. 2.1.) через шар абстракції, званий Window. Цей рівень абстракції надає ряд API для непрямого взаємодії з пристроєм [34].

Також через цей рівень абстракції Flutter Engine повідомляє Flutter Framework, коли:

- Подія, що представляє інтерес, відбувається на рівні пристрою (зміна орієнтації, зміна налаштувань, проблема з пам'яттю, стан роботи програми тощо).
- Подію, що відбувається на рівні жестів.
- Канал платформи відправляє деякі дані.
- Flutter Engine готовий до рендерингу нового кадру.

На перший погляд, Flutter – це реактивна, декларативна структура інтерфейсу користувача, в якій розробник забезпечує відображення від стану програми до стану інтерфейсу, а платформа бере на себе завдання оновлення інтерфейсу під час виконання, коли стан програми змінюється. Ця модель натхненна фреймворком ReactJS від Facebook, який включає переосмислення багатьох традиційних принципів дизайну [35].

У більшості традиційних фреймворків початковий стан інтерфейсу користувача описується один раз, а потім окремо оновлюється кодом під час виконання у відповідь на різноманітні події. Однією із проблем цього підходу є те, що, оскільки додаток ускладнюється, розробник повинен знати, як стан змінює каскад у всьому інтерфейсі користувача.

У Flutter віджети (подібно до компонентів у React) представлені незмінними (immutable) класами, які використовуються для конфігурації дерева об'єктів. Ці віджети використовуються для управління окремим деревом об’єктів для макетування, яке потім використовується для управління окремим деревом об’єктів для композитування [36].

Перетворення стану об'єкту на UI-елемент у Flutter можна відобразити наступної формулою:

$$UI = f(state), \tag{2.1}$$

де:

UI – це візуальні елементи додатку,

f – build-методи віджетів (функції, які будують візуальний інтерфейс),

state – стан додатку.

Крім мобільних платформ, Flutter також має підтримку компіляції у нативні програми від настільні операційні системи, такі як Windows, macOS та ОС на базі ядра Linux. Наприклад, компанія Canonical, яка займається розробкою ОС Ubuntu, створила нову хост-бібліотеку на основі GTK+ для підтримки додатків Flutter в усіх дистрибутивах Linux [37].

Крім підтримки десктопів, Flutter також підтримує вебплатформу. При побудові вебдодатку Flutter транслює весь Dart код у JavaScript, Canvas, WebGL, WebAssembly за допомогою dart2js та інших вспомогалих пакетів (рис. 2.2.) [38].

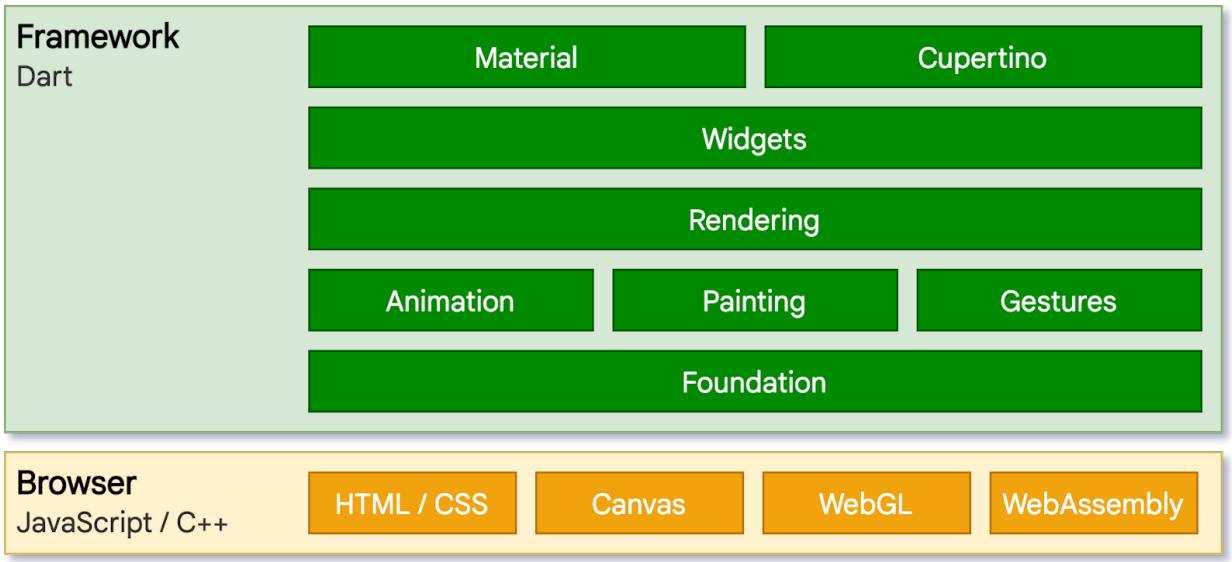

Рис. 2.2. Схема архітектури Flutter for Web.

Для побудови вебдодатку на Flutter достатньо зробити це за допомогою команди:

flutter build web

Підтримка вебдодатків та настільних програм наразі у альфа-версії, та не всі функції працюють стабільно та швидко.

Також Flutter можливо використовувати для розробки embedded-систем (або інтерфейсу до нього) (рис. 2.3.). Наприклад, компанія Sony використовує Flutter в своїх embedded-системах [39].

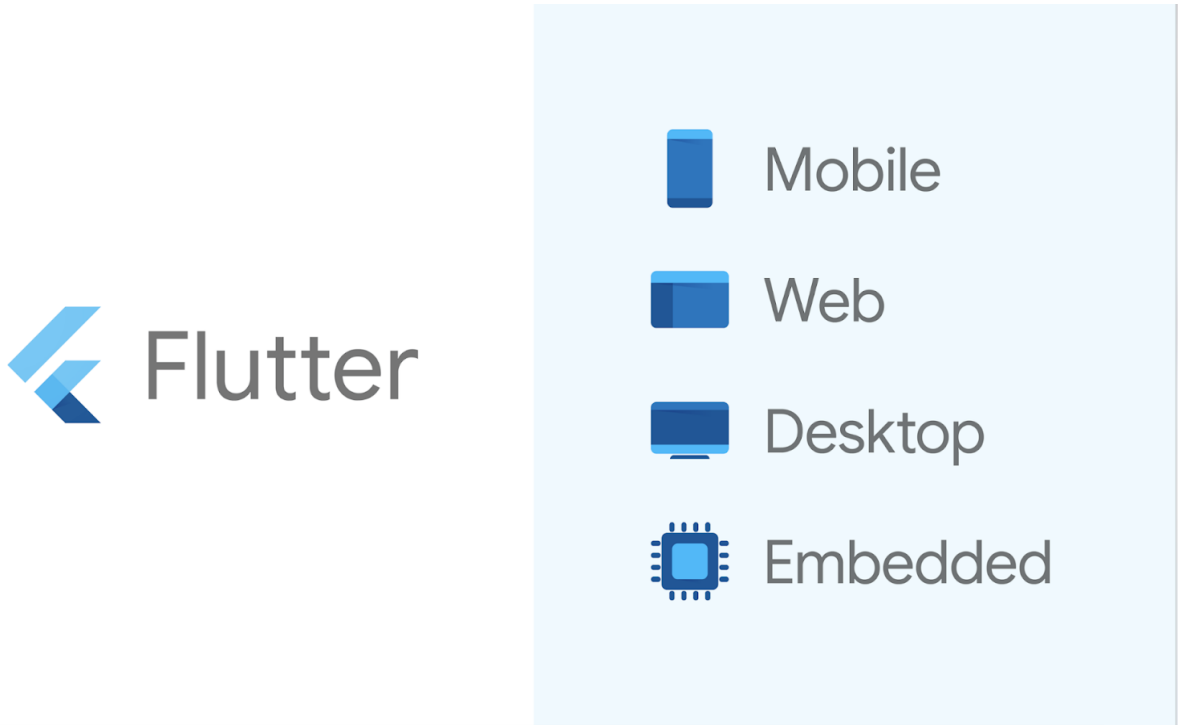

Рис. 2.3. Платформи, які підтримують Flutter.

Наразі фреймворк Flutter в свої продуктах використовують такі компанії як Sony, Google, Tencent, The New York Times, Square, ebay, BMW, nuBank, ByteDance та інші.

Прикладами додатків на Flutter є Groupon, nuBank, Philips Hue, Greentea, eBay Motors [40].

2.1.1. Основи композиції елементів у Flutter

Flutter наголошує на віджетах як на одиниці композиції фреймворку. Віджети є будівельними блоками користувацького інтерфейсу програми Flutter, і кожен віджет є незмінною декларацією частини користувальницького інтерфейсу [41].

Віджети є центральною ієрархією класів у рамках Flutter. Віджет – це незмінний опис частини користувальницького інтерфейсу. Віджети можна роздути на елементи, які керують базовим деревом рендеринга. У той же час навколо нього безліч додаткових механізмів, які допомагають фреймворку справлятися зі своїм завданням. Наприклад, Елемент – це деяке уявлення віджета в певному місці дерева [42].

Віджет описує конфігурацію деякої частини призначеного для користувача інтерфейсу, але один і той же віджет може використовуватися в різних місцях дерева. Кожне таке місце буде представлено відповідним елементом. Але з часом, віджет, який пов'язаний з елементом може помінятися. Це означає, що елементи більш живучі і продовжують використовуватися, лише оновлюючи свої зв'язки.

Елемент створюється за допомогою виклику методу Widget.createElement і конфігурується екземпляром віджета, у якого був викликаний метод.

В оголошенні класу можна бачити, що елемент імплементує BuildContext інтерфейс. BuildContext – це щось, що управляє позицією віджета в дереві віджетів, як випливає з його документації. Майже в точності збігається з описом елемента. Даний інтерфейс використовується, щоб уникнути прямого маніпулювання елементом, але при цьому дати доступ до необхідних методів контексту [43].

Остання ланка – це RenderObject. Як випливає з назви – це об'єкт дерева візуалізації. У нього є батьківський об'єкт, а також поле з даними, що батьківський об'єкт використовує для зберігання специфічної інформації, що

стосується самого цього об'єкта, наприклад, його позицію. Даний об'єкт відповідає за реалізацію базових протоколів відтворення і розташування [44].

RenderObject не обмежує модель використання дочірніх об'єктів: їх може не бути, може бути один або безліч. Також не обмежується система позиціонування: картезіанська система, полярні координати, все це доступно для використання. Немає обмежень і на використання протоколів розташування: підстроювання по ширині або висоті, обмеження розміру, завдання розмірів і розташування батьком або при необхідності використання даних батьківського об'єкта [45].

З цього слідує, що візуальні елементи у Flutter складаються з трьох рівнів дерев (рис. 2.4.):

- Дерево віджетів, в зону відповідальності якого належить декларування конфігурацій і зберігання властивостей.
- Дерево елементів, які керують життєвим циклом і пов'язують віджети в деяку ієрархію і об'єкти рендеринга з ними.
- Дерево об'єктів візуалізації, відповідальність яких – відтворення на екрані, облік розмірів, положення і обмежень (кордонів) елементів.

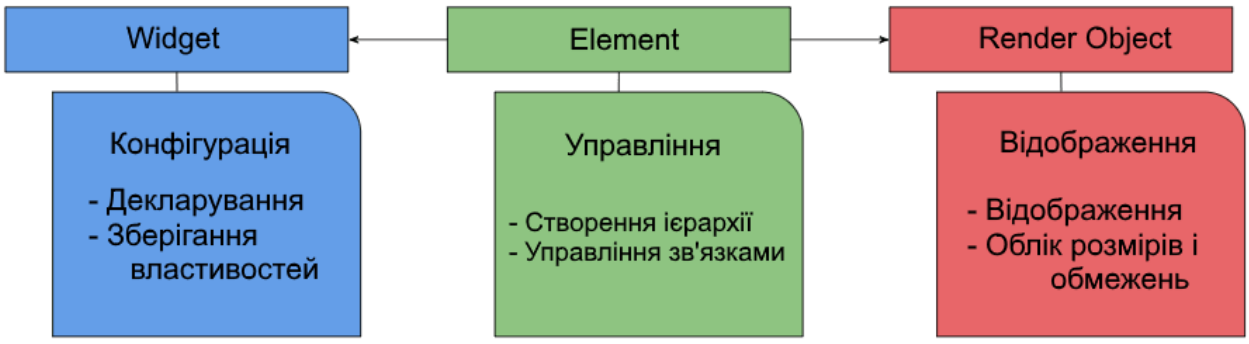

Рис. 2.4. Взаємодія Widget – Element – RenderObject.

Віджети утворюють ієрархію на основі композиції. Кожен віджет гніздиться всередині свого батька (parent) і може отримувати контекст від батьків.

Ця структура йде аж до кореневого віджета (контейнера, в якому розміщена програма Flutter, як правило, це MaterialApp або CupertinoApp).

Приклад такої композиції у програмі “Hello World” на Flutter:

```
import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Welcome to Flutter',
      home: Scaffold(
        appBar: AppBar(
          title: Text('Welcome to Flutter'),
        ),
        body: Center(
          child: Column(
       children: [
              Text('Hello World'),
              SizedBox(height: 20),
              ElevatedButton(
                     onPressed: () {
                            print('Click);
                     },
                     child: Text('A button),
              ),
       ],
   ),
        ),
      ),
    );
  }
}
```

У цьому прикладі, можна побачити, як у першій строчці коду спочатку у код імпортується пакет flutter/meterial.dart, який є набором віджетів виконаних у Material Design. Далі в точці входу програми (метод main) виконується запуск додатку з корневим віджетом MyApp.

Віджет MyApp наслідується від StatelessWidget (віджет без стану), та задає метод свого батьківського класу – build(BuildContext context). Саме в цьому методі виконується побудова інтерфейсу віджету.

Далі можна побачити початок композиції віджетів один в інший. Корневим віджетом є MaterialApp та Scaffold, які задають каркас для інтерфейсу додатку. За допомогою цих віджетів можна сформувати меню додатку (бокове меню, нижню навігацію, верхній тулбар тощо).

А у тіло (body) віджету MaterialApp вже можна додати інші віджети, які і стануть основним інтерфейсом користувача.

У вищезгаданому прикладі, після проголошення каркасу додатку, спочатку додається віджет Center, який буде центрувати усі інші віджети, які будуть його дітьми. Далі, в цей віджет компонується віджет Column, який задає вертикальне розташування наступних віджетів (в одну колонку).

І вже у цей віджет компонуються віджети Text, для відображення тексту, SizedBox, для відступу від тексту, та ElevatedButton – кнопка, яка обробляє клики на себе.

Побудоване дерево віджетів слід читати з верхнього до нижнього: “по центру, вертикально, розташовуємо текст, відступ та кнопку один за одним”.

Віджети, як правило, складаються з багатьох інших невеликих одноцільових віджетів, які поєднуються для отримання потужних ефектів.

Саме так виглядає побудова інтерфейсів у Flutter за допомогою віджетів. І код з прикладу є повноцінним мобільним додатком на Flutter, який можна скомпілювати під iOS, Android, web та desktop.

Фреймворк має два основних класи віджетів: віджети з формуванням стану (Stateful) та без нього (Stateless).

У багатьох віджетах немає змінних станів: вони не мають властивостей, які змінюються з часом (наприклад, іконка чи якийсь текстовий лейбл). Ці віджети будуть підходити підкласу StatelessWidget.

Однак, якщо унікальні характеристики віджета потрібно змінити на основі взаємодії користувача чи інших факторів, цей віджет повинен мати стан – бути StatefulWidget [46].

Наприклад, якщо у віджеті є лічильник, який збільшується щоразу, коли користувач натискає кнопку, тоді значення лічильника є станом для цього віджета. Коли це значення змінюється, віджет потрібно відновити, щоб оновити його частину інтерфейсу.

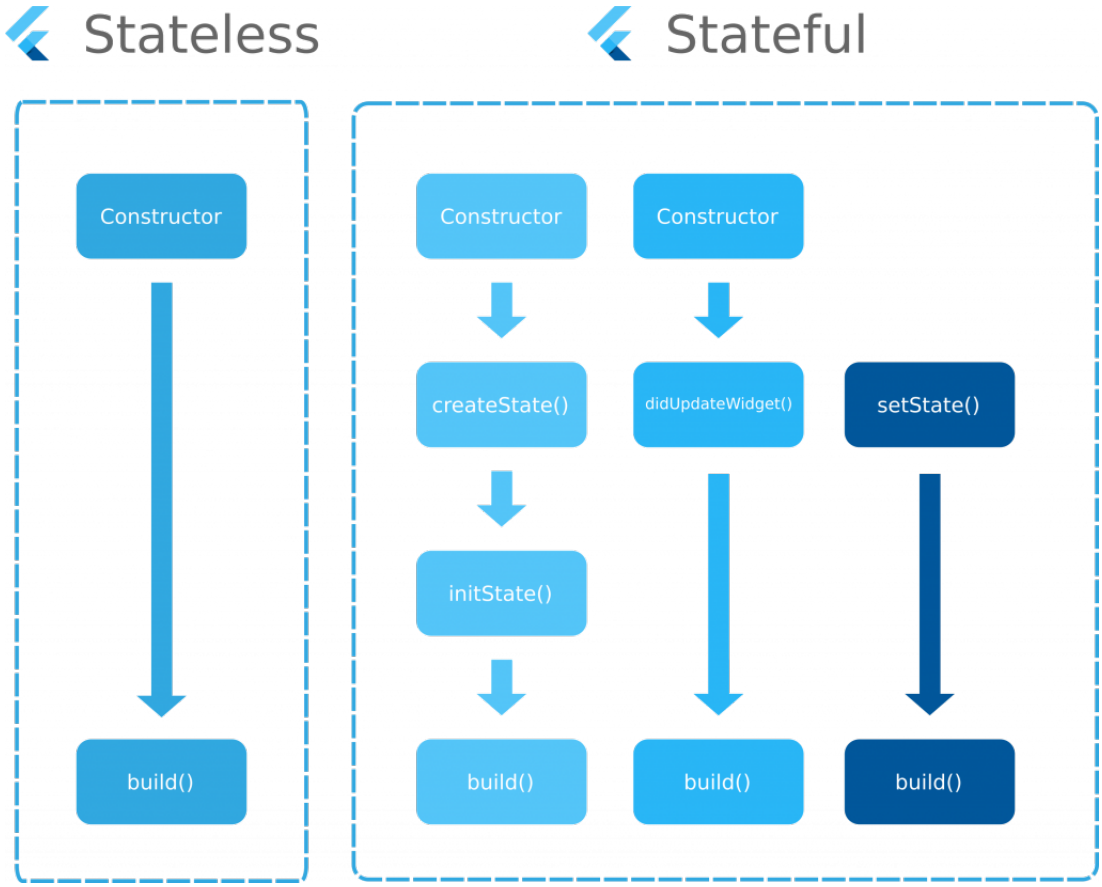

Рис. 2.5. Схема рендеру Stateless та Stateful віджетів.

Ці віджети мають підклас StatefulWidget, і (оскільки сам віджет є незмінним) вони зберігають змінний стан в окремому класі, який називається State, та є частиною StatefulWidget. StatefulWidget не має методу побудови, натомість користувальницький інтерфейс з цими віджетами будується через їхній об'єкт State (рис. 2.5.).

Кожного разу, коли змінюється об'єкт стану (наприклад, збільшується лічильник), повинен викликатися метод setState(), що сигналізує фреймворку про оновлення користувальницького інтерфейсу, знову викликаючи метод побудови стану та відтворення віджету [47].

2.1.2. Управління потоком даних у Flutter

Вбудована система управління станами у Flutter дозволяє створювати комплексні системи з віджетів, які можуть реагувати на зміни станів один одного, отримувати його від інших джерел тощо, наприклад за допомогою InheritedWidget.

Це особливий вид віджета, який визначає контекст в корені піддерева. Він може ефективно надавати цей контекст кожному віджету в своєму піддереву. Тобто, передавати якісь дані від кореня, до листя дерева. Багато з часто використовуваних контекстів Flutter, такі як Style або MediaQuery, є ні що інше, як InheritedWidget-и, які існують на рівні MaterialApp, та дають можливість працювати зі стилями та медіа-запитами в усіх віджетах [48].

На основі InheritedWidget, в стандартну бібліотеку Flutter був доданий ChangeNotifier. ChangeNotifier – це простий клас, включений до Flutter SDK, який забезпечує повідомлення про зміни своїх слухачів. Іншими словами, якщо щось є ChangeNotifier, то на нього можна підписатися для того, щоб отримувати його зміни, та змінювати за допомогою нього стан віджетів, та перебудовувати їх при змінах (наприклад, коли додаток отримує відповідь від серверу або бази даних) [49].

Це реалізація шаблон проєктування "Observer" ("Спостерігач"), який надає механізм підписки, який дає можливість одним об’єктам стежити й реагувати на події, які відбуваються в інших об’єктах [50].

Але такий підхід не є найкращим для побудови складних додатків, в яких існує безліч станів, які можуть суто змінювати дерево елементів.

Для рішення цієї задачи було створено безліч архітектурних підходів, таких як Provider, BLoC, MobX, Rx, Redux та інші. На поточний час спільнота не дійшла до єдиного стандарту або шаблону, за яким слід створювати програмне забезпечення за допомогою Flutter.

Однією з найпоширеніших є шаблон BLoC [51].

BLoC перекладається як business logic component, тобто "компонент бізнес-логіки". Це клас, що відокремлює бізнес-логіку додатка від інтерфейсу користувача. Такий компонент містить код, який можна повторно використовувати де завгодно: в іншому модулі, на іншій платформі, в іншому додатку [52].

BLoC працює як єдине джерело правди (single source of truth) для кожної незалежної функції додатку. Прикладом функції може бути автентифікація, робота з профілем користувача або іншою бізнес-логікою. BLoC реагує на події, відповідаючи на них змінами стану за допомогою асинхронних функцій (рис. 2.6.).

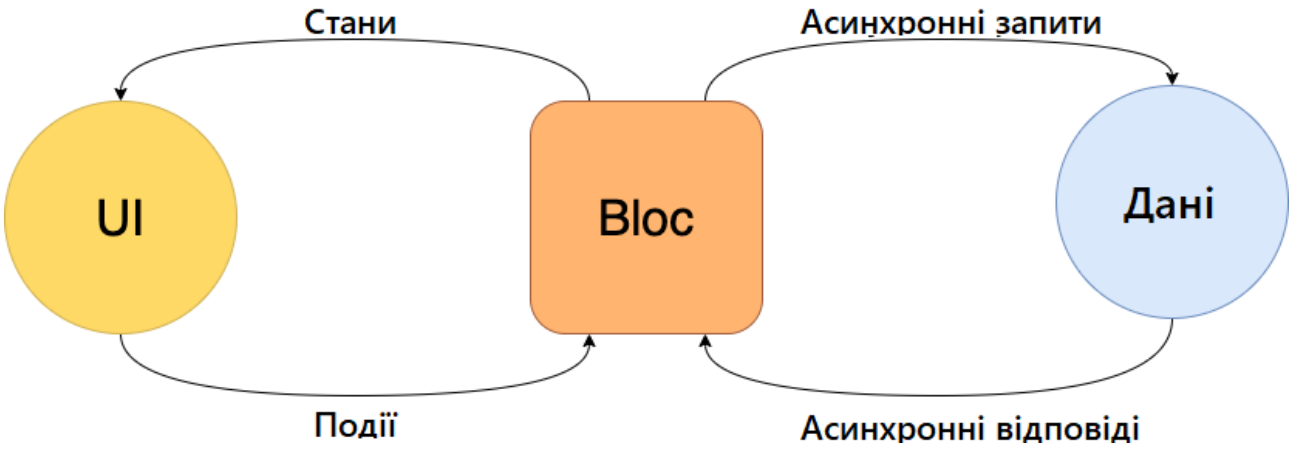

Рис. 2.6. Принцип роботи BLoC.

2.2. Аналіз методів роботи фреймворку React Native

React Native – це фреймворк від Facebook для розробки кросплатформенних мобільних додатків (рис. 2.7.). Побудований на базі бібліотеки ReactJS, яка призначена для розробки вебдодатків, але без

використання WebView та DOM API, які використовуються у браузері, або Cordova [53].

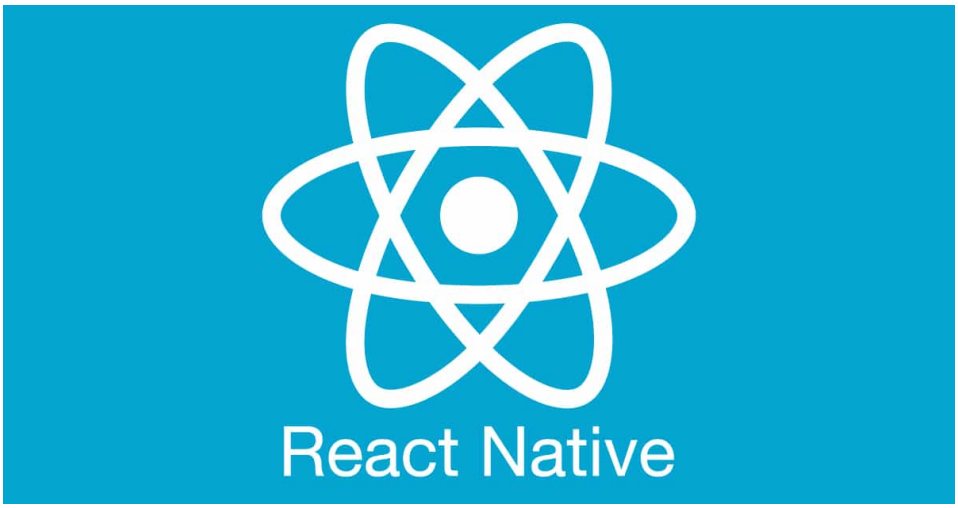

Рис. 2.7. Логотип React Native.

Фреймворк працює у фоновому процесі (який інтерпретує JavaScript) безпосередньо на кінцевому пристрої та взаємодіє з власною платформою через асинхронний міст, через який відтворюється нативний інтерфейс для кожної платформи окремо [54].

Платформа React Native складається з трьох основних частин:

- Нативний код та модулі.

Більша частина нативного коду, який є частиною фреймворку React Native, для iOS написана на Objective-C або Swift, тоді як для Android вона написана на Java або Kotlin. Але для написання програми на React Native ці мови треба використовувати доволі рідко, тільки коли не вистачає вже існуючих нативних модулей, або треба внести зміни в існуючі [55].

- Javascript Virtual Machine.

Це віртуальна машина JS, яка запускає весь код додатку на JavaScript. На емуляторах і пристроях iOS та Android React Native використовує JavaScriptCore, який є двигуном JavaScript, що працює на Safari.

JavaScriptCore – це рушій JavaScript з відкритим кодом, який на початку свого існування був вбудований у WebKit. У випадку iOS, React Native

використовує JavaScriptCore, наданий платформою iOS. Вперше він був представлений в iOS 7 разом з OS X Mavericks [56].

У випадку Android, React Native поєднує JavaScriptCore разом із додатком. Це збільшує розмір програми. Отже, додаток на кшталт “Hello World”, на RN займе від 3 до 4 мегабайт для Android, бо буде поставлятися разом з віртуальною машиною, на якій він працює [57].

У випадку режиму налагодження програми, написаної на React Native, використовується Chrome Dev Tools, і код JavaScript в цьому випадку працює в самому Chrome (замість JavaScriptCore на пристрої) і взаємодіє з нативним кодом через WebSocket. Тут він буде використовувати рушій V8, як і браузер Chrome [58].

Це дозволяє бачити розробникам багато інформації про інструменти налагодження додатку, такі як мережеві запити, журнали консолі тощо.

React Native bridge – це міст, написаний на C++ та Java, який відповідає за зв'язок між нативною частиною і потоком JavaScript, в якому відбувається обробка основного коду додатка (рис. 2.8.).

Цей міст використовує спеціальний протокол для передачі повідомлень між JavaScript та платформою [59].

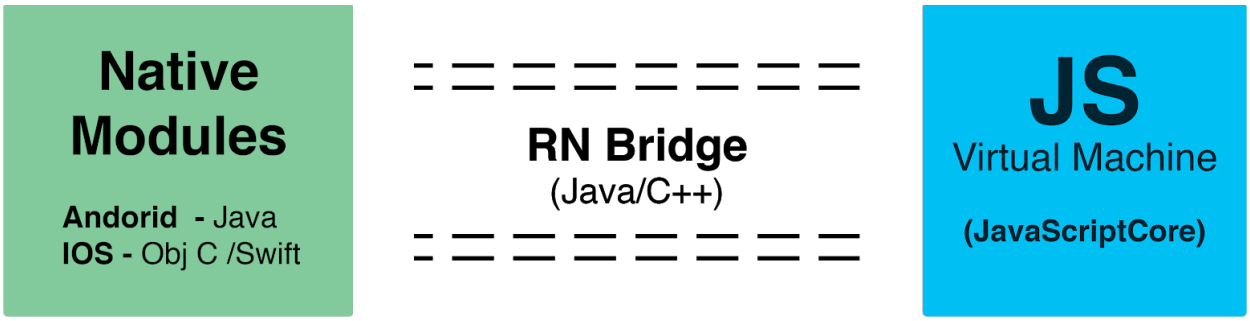

Рис. 2.8. Три основні модулі React Native.

Через використання спеціального мосту страждає швидкодія роботи такого програмного забезпечення, особливо на слабких пристроях, бо увесь код додатку спочатку обробляється у віртуальній машині, а потім через міст

передається до нативної частини мобільної платформи, яка вже є відповідальна за відтворення інтерфейсу.

В цьому випадку виграє Flutter, який призводить AOT-компіляцію коду додатку у нативний код, та самостійно відтворює кожен піксель на екрані, без використання мостів.

Міст у React Native має три важливі характеристики.

1. Асинхронний. Це забезпечує асинхронний зв'язок між потоками. Це гарантує, що вони ніколи не блокують одне одного.
2. Серійний. Він передає повідомлення з одного потоку в інший в оптимізованому вигляді.
3. Серіалізується. Два потоки ніколи не обмінюються одними даними та не працюють з ними. Натомість вони обмінюються серіалізованими повідомленнями.

У більшості випадків розробник пише всю програму React Native у Javascript. Для запуску програми через CLI видається одна з наведених команд – *react-native run-ios* або *react-native run-android*.

На цьому етапі React Native CLI створює пакувальник вузлів, який об’єднає код JS в один файл main.bundle.js.

Коли запускається програма React Native, першим завантажувальним елементом є рідна точка входу. Нативний потік створює потік JS VM, який запускає в комплекті код JS.

Код JS включає всю бізнес-логіку програми. Нативний потік надсилає повідомлення через RN Bridge для запуску програми JS.

Далі породжений потік Javascript починає видавати інструкції для нативного потоку через RN Bridge.

Інструкції включають, які екрани та елементи відтворювати, яку інформацію слід отримати з апаратного забезпечення тощо.

Наприклад, якщо потік JS хоче створити якийсь візуальний лемент, наприклад, View та Text, він буде групувати запит в одне повідомлення та надсилати його до нативного потоку для їх візуалізації [60].

Наприклад, в такому вигляді:

[[2,3,[2,'Text',{...}]] [2,3,[3,'View',{...}]]]

Коли запущено додаток React Native, він створює наступні черги потоків:

1. Основний потік (нативна черга, main thread).

Це основний потік, який виникає відразу після запуску програми. Він завантажує додаток і запускає потік JS для виконання коду Javascript.

Основний потік також прослуховує події користувацького інтерфейсу, такі як тапи, скролли тощо.

Потім ці події передаються в потік JS через RN Bridge. Після завантаження Javascript потік JS надсилає інформацію про те, що потрібно відтворити на екран.

Ця інформація використовується потоком тіньового вузла для обчислення UI. Потік тіні в основному схожий на математичний механізм, який остаточно вирішує, як обчислити позиції UI. Потім ці вказівки передаються назад до основного потоку, щоб відтворити UI.

Щоб підтримувати хорошу продуктивність, потік JS повинен мати можливість надсилати пакетні оновлення в потік інтерфейсу користувача до закінчення терміну наступного кадру [61].

2. Потік Javascript (черга JS).

Черга Javascript – це черга потоків, де працює основний пакетний потік JS. Потік JS запускає всю бізнес-логіку, тобто код, який розробник пише в React Native.

3. Спеціальні нативні модулі.

Окрім потоків, породжених React Native, розробники також можуть створювати потоки на власних модулях. Наприклад, анімації обробляються в React Native окремим власним потоком, щоб розвантажити роботу потоку JS.

React, основа React Native, – це JavaScript-бібліотека з відкритим вихідним кодом для розробки інтерфейсів користувача. Вона має декларативну основу, як і Flutter, і підходить для створення користувацьких інтерфейсів. Вона дозволяє збирати складний UI з маленьких ізольованих шматочків коду, званих

«компонентами», так само як і Flutter, який перейняв від React декларативний підхід.

Наприклад, iOS відображає 60 кадрів на секунду, і це призводить до нового кадру кожні 16,67 мс. Якщо виконується якась складна обробка у циклі подій JavaCcript, яка призводить до змін користувальницького інтерфейсу, і це займає більше 16,67 мс, тоді інтерфейс буде виглядати млявим та не швидким.

З використанням мосту, цієї цілі досягти може бути важче, ніж у нативній розробці або у Flutter, хоча команда React Native намагається вносити нові оптимізації для забезпечення швидкої роботи.

Інтерфейс у React Native будується за допомогою множини компонентів, які виступають в ролі цеглинок в UI.

Компонент у React – це ділянка коду, який представляє частину сторінки. Це може бути кнопка, текстове поле, або більш комплексний компонент, який включає в себе купу інших компонентів [62].

Створені компоненти описують, яким чином необхідно їх відтворити, за допомогою методу render() (аналог методу build() у віджетах Flutter).

React буде ефективно оновлювати компоненти і малювати їх, коли дані зміняться. Такий самий підхід має і Flutter для оновлення даних у віджетах.

React створює кеш-структуру в пам'яті, що дозволяє обчислювати різницю між попереднім і поточним станами інтерфейсу для оптимального оновлення компонентів.

Таким чином програміст може працювати зі сторінкою, вважаючи, що вона оновлюється вся, але бібліотека самостійно вирішує, які компоненти сторінки необхідно оновити, так само як у Flutter.

Але з автоматичним оновленням треба бути обачним через те, що неправильні дії програміста можуть призвести до зайвого перемальовування інтерфейсів, що в свою чергу може призвести до зменшення швидкодії.

React спостерігає за зміною стану. Якщо нічого не змінилося, бібліотека нічого не перемальовує на екрані. Якщо якісь дані змінилися, бібліотека знає, що потрібно оновити частину компонентів.

Компоненти у методі render() описуються за допомогою JSX.

JavaScript XML (JSX) – це розширення синтаксису JavaScript, яке дозволяє використовувати HTML-подібний синтаксис для опису структури інтерфейсу.

Фундаментально, JSX є синтаксичним цукром для функції React.createElement (component, props, ... children) [63].

Тобто, наступний JSX код:

```
<Button color="blue" shadowSize={2}>
  Click Me
</Button>
```

Компілюється в:

```
React.createElement(
  Button,
  {color: 'blue', shadowSize: 2},
  'Click Me'
)
```

Цей код описує компонент “Кнопку” синього кольору, з тінню розмірів в 2 пікселі, та текстом “Click Me”.

У кожного компонента є свій життєвий цикл. Методи життєвого циклу дозволяють розробнику запускати код на різних стадіях життєвого циклу компонента [64].

Наприклад:

- shouldComponentUpdate – дозволяє запобігти перерисовку компонента за допомогою повернення false, якщо перерисовка не потрібна.
- componentDidMount – викликається після першого відтворення компонента. Часто використовується для запуску отримання даних з віддаленого джерела через API.
- render – найважливіший метод життєвого циклу. Кожен компонент повинен мати цей метод. Зазвичай викликається при зміні даних компонента для перемальовування даних в інтерфейсі.

В процесі роботи компонент проходить через ряд етапів життєвого циклу. На кожному з етапів викликається певна функція, в якій можна визначити будь-які дії (рис. 2.9.):

- constructor(props): конструктор, в якому відбувається початкова ініціалізація компонента.
- componentWillMount(): викликається безпосередньо перед рендерингом компонента.
- render(): рендеринг компонента.
- componentDidMount(): викликається після рендеринга компонента. Тут можна виконувати запити до віддалених ресурсів.
- componentWillUnmount(): викликається перед видаленням компонента.

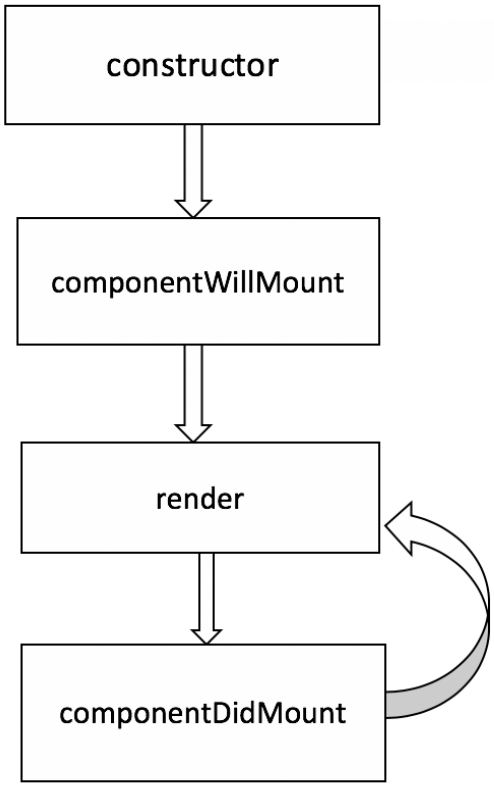

Рис. 2.9. Життєвий цикл компонентів React Native.

Одним з плюсів React Native порівняно з Flutter є те, що деякі компоненти та механізми програми можна використовувати з фреймворку ReactJS, тобто перевикористовувати частину вже написаного коду.

Але на практиці це виходить не дуже часто через те, що код з ReactJS треба налаштувати для роботи у React Native: змінити стилі, налагодити роботу на усіх платформах тощо.

Також існують версії React для роботи з віртуальною реальністю, та сторонні реалізації на базі фреймворку Electron для побудови desktop-програм для настільних операційних систем.

React Native адаптує Javascript під розробку для мобільних пристроїв. Це досягається тим, що для складання проєктів він використовує кілька збирачів – Metro Bundler, який інтерпретує JS-код і представляє ресурси і збирач цільової системи [65].

React Native підтримує також і використання сторонніх бібліотек на базі NPM, де також знаходиться велика кількість різних бібліотек для React Native, що розширюють функціонал і спрощують розробку [71].

Але через те, що всі бібліотеки пишуться в різний час, вони використовують різні версії SDK і вимагають різного підходу.

Іноді буває так, що бібліотеки несумісні одна з одною, або остання версія бібліотеки виявляється експериментальної, і самі розробники радять знизити версію до передостанньої. Проблема несумісності різних версій React Native є доволі розповсюдженою.

Фреймворк React Native використовує Facebook, UberEats, Bloomberg.

Але є і компанії, які використовували React Native, але через ряд проблем, з якими вони зіткнулися під час експлуатації та розробки на React Native, вирішили відмовитися від нього, та повернулися до написання мобільних додатків іншими методами. Серед таких компаній є, наприклад, AirBnb [66] та Udacity [67].

2.2.1. Управління потоком даних у React Native

React використовує односпрямовану передачу даних між компонентами та екранами. Властивості передаються від батьківських компонентів дочірнім.

Компоненти отримують властивості як множину незмінних (immutable) значень, тому компонент не може безпосередньо змінювати властивості, але може викликати зміни через callback-функції. Такий механізм називають «властивості вниз, події наверх».

Props представляє колекцію значень, які асоційовані з компонентом. Ці значення дозволяють створювати динамічні компоненти, що не залежать від жорстко закодованих статичних даних. Параметр props, який передається в функцію компонента, інкапсулює властивості об'єкта [69].

Об'єкт state описує внутрішній стан компонента, він схожий на props за тим винятком, що стан визначається всередині компонента і є доступен тільки з компонента.

Якщо props представляє вхідні дані, які передаються в компонент ззовні, то стан зберігає такі об'єкти, які створюються в компоненті і повністю залежать від компонента.

Також на відміну від props значення в state можна змінювати.

І ще важливий момент – значення з state повинні використовуватися під час рендерингу. Якщо якийсь об'єкт не використовується під час рендерингу компонента, то немає сенсу зберігати його в state.

Нерідко state описує якісь візуальні властивості елемента, які можуть змінюватися при взаємодія з користувачем.

Наприклад, коли користувач натиснув кнопку, і відповідно можна змінити її стан – надати їй якийсь інший колір, тінь і так далі.

Єдине місце, де можна встановити об'єкт state – це конструктор класу.

Механізм локального сховища компонента, який поставляється разом з базовою бібліотекою (React) незручний тим, що таке сховище ізольовано.

Наприклад, якщо треба зробити так, щоб різні незалежні компоненти реагували на будь-яку подію, доведеться або передавати локальний стан у вигляді props дочірнім компонентам, або піднімати його вгору до найближчого батьківського компонента.

В обох випадках робити це не зручно. Код стає брудніше, складним для розуміння, а компоненти залежними від їх вкладеності.

Така бібліотека, як Redux знімає цю проблему так як увесь стан доступен усім компонентом, і є одним з найпопулярніших рішень для роботи зі станом у середовищі React-розробників.

Але Redux є універсальним засобом розробки і може бути використаний в зв'язці з різними бібліотеками і фреймворками [70].

У Redux загальний стан додатку представлено одним об'єктом JavaScript – state (стан) або state tree (дерево станів).

Сховище (store) – це об'єкт, який:

- Містить стан додатка,
- відображає стан через getState();
- може постійно оновлювався через dispatch();
- дозволяє реєструватися в якості слухача зміни стану через subscribe().

Сховище в додатку завжди унікально.

Незмінне дерево станів є доступне тільки для читання, але змінити нічого на пряму не є можливим. Зміни можливі тільки при відправці action (дії).

Дія (action) – це JavaScript-об'єкт, який лаконічно описує суть зміни, та дає команду.

Під час запуску дії обов'язково щось відбувається і стан додатка змінюється. Далі робота передається до редукторів.

Редуктор (reducer) – це чиста функція, яка обчислює наступний стан дерева на підставі його попереднього стану і застосовуваної дії (action).

Функція повинна задовольняти двом умовам, щоб вважатися «чистої»:

- Кожен раз функція повертає однаковий результат, коли вона викликається з тим же набором аргументів.
- Немає побічних ефектів.

Чиста функція працює незалежно від стану програми і видає вихідне значення, приймаючи вхідне і не змінюючи нічого в ньому.

Тобто, редуктор повертає абсолютно новий об'єкт дерева станів, яким замінюється попередній об’єкт стану.

Редуктор – це завжди чиста функція, тому він не повинен мутувати (змінювати) аргументи або мутувати стан. Замість цього створюється новий стан. Також він не має мати побічні ефекти (ніяких API-викликів з якими-небудь змінами), викликати нечисті функції [71].

Нечисті функції – це функції, результат яких залежить від чогось крім їх аргументів (наприклад, функція, яка повертає актуальний час).

Протилежність чистій функції, де результат завжди залежить від аргументів, які цій функції передаються розробником.

Оскільки стан в складних додатках може сильно розростатися, до кожної дії застосовується не один, а відразу кілька редукторів.

Потік даних в Redux завжди однонаправлений.

Передача дій з потоками даних відбувається через виклик методу dispatch() в сховищі. Саме сховище передає дії редуктора і генерує наступний стан, а потім оновлює стан і повідомляє про це всіх слухачів (рис. 2.10.).

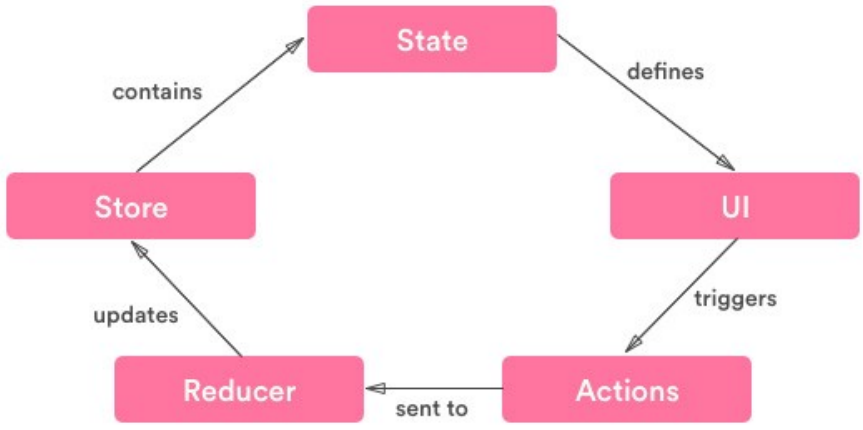

Рис. 2.10. Схема роботи Redux.

2.3. Висновки до другого розділу

В другому розділі були проаналізовані методи роботи засобів React Native та Flutter, їхня архітектура та внутрішня будова, які дозволяють проводити розробку мобільних додатків під декілька платформ одночасно.

Було розібрано, як саме Flutter та React Native відображають графічний інтерфейс користувача, через що і як проходять їхні команди, щоб виконувати їх однаково на різних операційних системах.

Flutter використовує свій власний рушій для відтворення інтерфейсу, який працює на рівні нативних бібліотек, коли як React Native з фонового потоку через віртуальну машину JavaScript надає команди операційній системі щодо відтворення інтерфейсу через спеціальний міст.

Також були розібрані підходи до управління станами всередині мобільних додатків, та те, як це відбувається. Стандартне управління станами між віджетами та компонентами, також управління станом за допомогою BLoC та Redux підходів, які дозволяють маніпулювати великими потоками даних між компонентами мобільного додатку.

РОЗДІЛ 3
ПОРІВНЯННЯ ТА ДОСЛІДЖЕННЯ ЗАСОБІВ FLUTTER ТА REACT NATIVE

3.1. Аналіз, порівняння та дослідження засобів Flutter та React Native між собою.

React Native був випущений на конференції F8 від Facebook у березні 2015 року. Flutter був представлений на 3 роки пізніше – у 2017 році, а перший стабільний реліз – в грудні 2018 року.

Вони обидва є OpenSource. Тобто, проєктами з відкритим сирцевим кодом, який можуть копіювати, змінювати та відтворювати усі охочі.

Основною мовою React Native є JavaScript, яка використовується для розробки вебдодатків, які працюють на стороні клієнта.

Мовою Flutter – Dart, мова, яку створила компанія Google задля вирішення основних вад JavaScript.

З одного боку, мова JavaScript є більш поширеною, і нею можуть розробляти більша кількість розробників. З іншого боку, команда React Native не може якось впливати на розвиток JavaScript в великому обсягу, і фреймворк React Native прив’язаний до недоліків JavaScript (динамічні типи, ООП на основі прототипів, підтримка різних стандартів реалізації мови тощо).

У випадку з Flutter, треба використовувати менш поширену мову, яка є доволі схожа на мову Java. Але і мову, і сам фреймворк контролює компанія Google, і вона пристосовує їх одне до одного, наприклад, додаючи такі функції в мову Dart, які є важливими саме для розробки на Flutter, а релізи мови та фреймворку пов’язані між собою. Завдяки цьому створюється доволі ефективний симбіоз мови та фреймворку для розробки.

А схожість синтаксису на інші C-подібні мови (такі, як Java, C# та інші), дозволяє швидко почати розробляти додатки на Dart (рис. 3.1.) [72].

Рис. 3.1. Логотип мови програмування Dart.

Встановлення Flutter відбувається з офіційного сайту flutter.dev, шляхом завантаження архіву з фреймворком, або завантаження з репозиторії за допомогою Git:

git clone https://github.com/flutter/flutter.git -b stable

Після завантаження та розархівування архіву фреймворк Flutter та середа виконання Dart готова до роботи та взаємодії. За допомогою команди flutter doctor можна запустити перевірку готовності усіх системних компонентів. Для роботи з Android також треба завантажити Android SDK з головного сайту Android, а ось для розробки під iOS треба мати комп'ютер з macOS та встановленим Xcode.

Для розробки з використанням Flutter можна завантажити різноманітні IDE: IntelliJ IDEA, Visual Studio Code або Android Studio зі спеціальним плагіном для роботи з Flutter.

Створити новий проєкт на Flutter можна за допомогою IDE або за допомогою команди:

flutter create myapp

Початок роботи з React Native є дещо складніший. Для початку, розробник має встановити NodeJS – середу виконання мови JavaScript за межами браузеру. Після встановлення середи виконання NodeJS, за допомогою менеджера JavaScript пакетів треба встановити пакет React Native, за допомогою наступної команди:

npm install -g react-native-cli

Після цього розробник може створювати нові проєкти для React Native за допомогою команди:

react-native init myapp

Розробляти додатки за допомогою React Native можливо у Visual Studio Code, WebStorm або іншому IDE з підтримкою потрібних плагінів.

Також, як і з Flutter, для роботи з Android треба завантажити Android SDK з головного сайту Android, а ось для розробки під iOS треба мати комп'ютер з macOS та встановленим Xcode.

Процес встановлення та початку роботи з обома фреймворками є доволі схожий, хоча Flutter потребує менше команд для початку роботи, встановлення середи виконання та фреймворку відбувається разом за допомогою однієї команди.

Офіційну документацію щодо роботи з цими фреймворками можна знайти на сайтах flutter.dev та reactnative.dev, але документація Flutter є більш дружньою для розробників, ніж документація React Native.

Сайт Flutter має інформацію щодо встановлення фреймворку, налаштування IDE або редактору, приклад Hello World, інструкції щодо міграції на основну інформацію про фреймворк для розробників, які раніше працювали в якості нативних розробників Android або iOS додатків, розробників React Native, Xamarin та розробників вебсайтів.

Крім цього, є введення у декларативну розробку інтерфейсів, маленький курс в мову Dart, приклади додатків на Flutter, уроки з розробки, тестування та випуску додатків, а також документації по усім стандартним віджетам.

React Native має набагато меншу документацію. В ній описані принципи роботи фреймворку, уроки з розробки та тестування, налаштування оточення розробки та інструкції щодо компонентів.

Для більшого уявлення про структуру фреймворків, розглянемо імплементації програми “Hello World”.

Програма “Hello World” написана на React Native:

```
import React from 'react';
import { Text, View } from 'react-native';
import ToolbarAndroid from '@react-native-community/toolbar-android';

const HelloWorldApp = () => {
  return (
    <ToolbarAndroid
      title="AwesomeApp">
      <View
        style={{
            flex: 1,
            justifyContent: "center",
            alignItems: "center"
        }}>
        <Text>Hello, world!</Text>
      </View>
</ToolbarAndroid>
  )
}
export default HelloWorldApp;
```

В цьому прикладі можна побачити, як імпортуються ключові компоненти React Native, а також сторонній компонент Toolbar для Android, який відповідає за відображення верхнього заголовку додатку.

React Native не має рідних компонентів, які б могли відобразити тулбар додатку, на кшталт того, як він виглядає у Material Design. Для цього треба завантажувати сторонні рішення.

Далі проходить створення стрілкової функції JavaScript “HelloWorldApp”, яка повертає JSX об’єкти, які будуть відтворюватися на екрані користувача.

Спочатку йде сторонній компонент ToolbarAndroid, який відповідає за відтворення тулбару додатку, в середині нього компонент View, який відіграє роль контейнеру для набору інших компонентів.

І цьому компоненту задаються flexbox стилі, схожі на CSS: flex, justifyContent, alignItems для вирівнювання контенту по центру екрана. Тобто, для того, щоб розмістити контент контейнеру по центру, розробник повинен задіяти одразу три параметри стилів.

В середині цього контейнеру вже знаходиться компонент Text, який є відповідальним за відображення тексту всередині нього.

І в останній строчці додатку дозволяється експорт функції “HelloWorldApp” (стандарт ES6 в JavaScript) (рис. 3.2.).

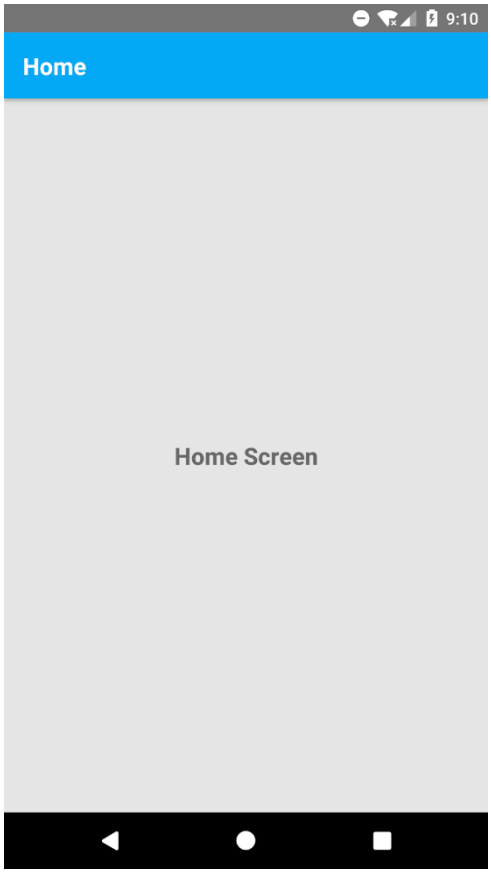

Рис. 3.2. Приклад “Hello World” на React Native.

Тепер можна подивитися на подібний “Hello World” у Flutter (рис. 3.3.):

```
import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Welcome to Flutter',
      home: Scaffold(
        appBar: AppBar(
```

```
          title: Text('Welcome to Flutter'),
        ),
        body: Center(
          child: Text('Hello World'),
        ),
      ),
    );
  }
}
```

В прикладі на Dart за допомогою фреймворку Flutter можна побачити один імпорт основних віджетів фреймворку (material.dart).

Запуск програми у Dart починається від функції main(), як в мовах на кшталт C/C++, де фреймворку дається команда для запуску додатку.

Додаток представлений як клас, який наслідується від віджету без стану (StatelessWidget), та повертає у методі build() об’єкт віджета MaterialApp, який відтворює на екрані основу додатку та має різні методи для стилей (назва додатку, стиль фону, колір фону та інше). В якості аргументу home йому надається інший об’єкт – Scaffold.

Scaffold – це універсальний віджет у Flutter, який надає розробнику можливість стандартними методами в кілька строк коду додати на екрані тулбар (заголовок з можливістю додати кнопки), бокове та/або нижнє меню та деякі інші функції (такі як відображення діалогів-повідомлень та іншого).

Далі у MaterialApp присутній аргумент body, в якому знаходиться головний контент додатку. Всередині нього створюється віджет-контейнер Center, який відцентровує дочірні віджети. Аргумент конструктору віджета child, який присутній в усіх віджетах-контейнерах, надає можливість задачи дочірній віджет (віджет в середині контейнеру або інших віджетів, схожа система, як в React Native).

В цьому прикладі це віджет Text, який відображає текст, який був написаний в аргументів конструктору.

Можна побачити, що React Native та Flutter використовують доволі схожі принципи побудови інтерфейсу – за допомогою декларативного стилю, замість

імперативного, який існує у нативній розробці мобільних додатків. Хоча нативна розробка мобільних додатків вже має такі засоби для написання інтерфейсу в декоративному стилі, як JetPack Compose для Android та SwiftUI для iOS.

React Native використовує JSX, який дає можливість у JavaScript писати декларативний UI в стилі схожим на HTML та CSS (стилі у React Native не використовують CSS, але є тільки схожими на неї).

У Flutter, в свою чергу, віджети створюються за допомогою створення стандартних екземплярів класів з іменованими аргументами.

Обидва засоби відтворення декларативного інтерфейсу є зручними та швидкими. Але стандартна бібліотека Flutter має набагато більшу кількість компонентів, порівняно з React Native, де частіше треба шукати сторонні рішення.

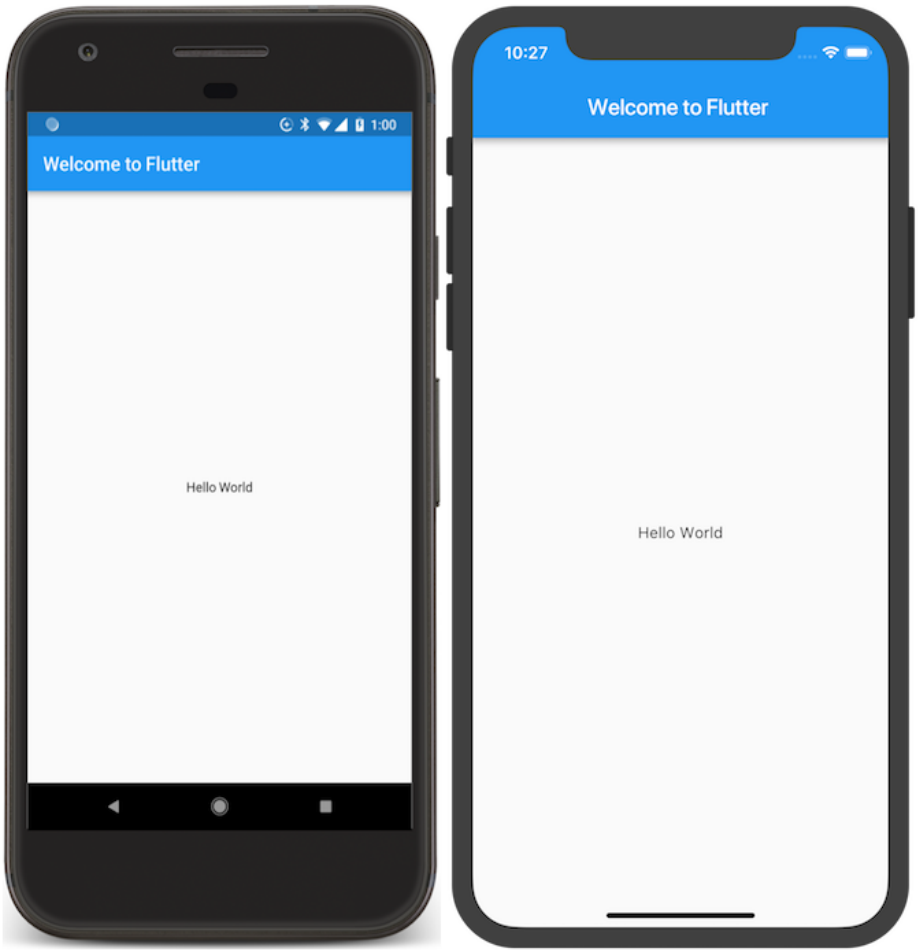

Рис. 3.3. “Hello World” на Flutter у Android та iOS.

Тепер також можна подивитися на більш просунутий приклад додатку на Flutter, який буде інкрементувати число по тапу на кнопку та буде відображати його у текстовому віджеті (рис. 3.4.):

```
class MyHomePage extends StatefulWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;
  @override
  _MyHomePageState createState() => _MyHomePageState();
}


class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;


  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }


  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
          children: <Widget>[
            Text('You have pushed the button this many times:'),
            Text('$_counter'),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        child: Icon(Icons.add),
      ),
    );
```

```
    }
}
```

Тут можна побачити приклад використання віджету, що має стан (StatefulWidget), про який йшлося у розділі 2.1.2. “Управління потоком даних у Flutter”.

“MyHomePage” є StatefulWidget, тобто таким, який підтримує стан, та після зміни цього стану викликає повторне відтворення себе з новим станом.

В якості стана виступає об’єкт “_MyHomePageState”, що є дочірнім від класу “State<MyHomePage>”.

Зміна стану відбувається у методі “_incrementCounter()”, в якому викликається метод батьківського класу “State” – “setState()”, що повідомляє віджету про зміну стану (в цьому прикладі – змінна-лічильник “_counter”).

Віджет отримує повідомлення про зміну стан, та розпочинає перевідтворення (рендерингу) себе та усього свого піддерева з новим станом.

Тому також важливо оновлювати стан тільки в тих віджетах, в яких потрібне оновлення даних: щоб не перемальовувати ті частини інтерфейсу, які не змінилися, що може вплинути на швидкодію. Працювати зі станом у React Native необхідно так само.

Зміна стану в цьому прикладі відбувається під час натискання плаваючої кнопки “+”, що прив’язана до Scaffold.

За схожим сценарієм працюють і складніші додатки, які спілкуються з сервером тощо, але в колі розробників прийнято використовувати інші, більш складні, системи управління станом (BLoC, MobX, Redux тощо).

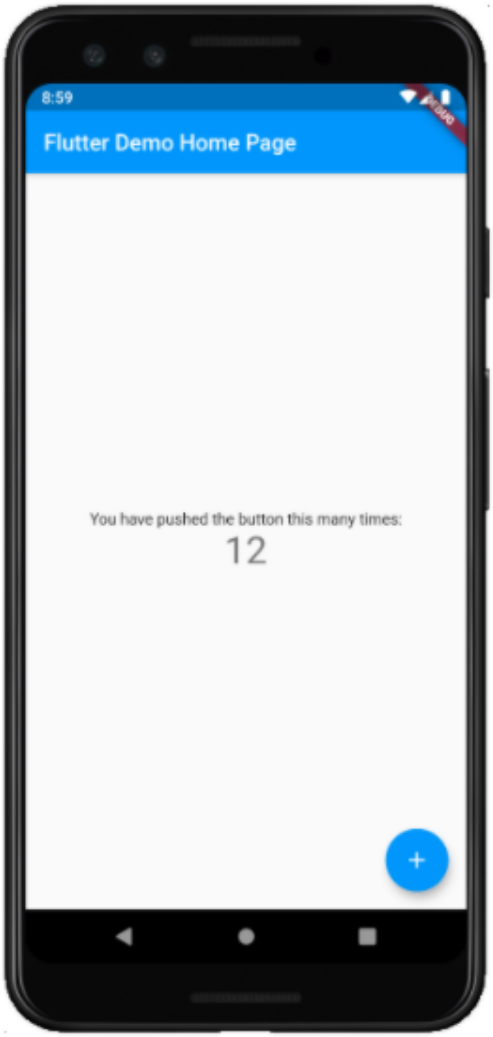

Рис. 3.4. Приклад програми-лічильника на Flutter.

Схожі системи управління станом реалізовані і в інших декларативних фреймворках.

Так, у React Native реалізована схожа система управління станом у компонентах:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }


  render() {
    return (
      <div>
```

```
        <h1>Привіт, світе!</h1>
        <h2>Зараз {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

У React Native стан задається через об’єкт “state” у самого компонента, і при зміні стану відбувається повторний рендер компоненту, так само як у Flutter.

Обидва фреймворки підтримують сторонні бібліотеки для використання. У React Native вони виконані через NPM, менеджер бібліотек та пакувань у мові JavaScript. У Flutter використовується менеджер бібліотек та пакунків Dart – Pub.

Обидва менеджери пакетів є зручними, мають багату кількість готових рішень та мають інтерфейс управління, за допомогою якого можна перевіряти та оновлювати бібліотеки.

Додатки Flutter виглядають на сучасних операційних системах так само добре, як і на старих версіях ОС.

Оскільки у них лише одна кодова база, програми виглядають і поводяться однаково в iOS та Android – але завдяки віджетам Material Design та Cupertino вони також можуть імітувати сам дизайн платформи.

Flutter містить два набори віджетів, які відповідають певним мовам дизайну: віджети Material Design реалізують однойменну мову дизайну Google; віджети Купертіно імітують дизайн Apple від iOS.

Додаток Flutter буде виглядати та виглядати природно на кожній платформі, імітуючи їхні рідні компоненти.

У React Native компоненти програми виглядають точно так само, як нативні (наприклад, кнопка на пристрої iOS виглядає так само, як кнопка у нативних додатках iOS, і така ж на Android).

Той факт, що React Native використовує власні компоненти під капотом, надає впевненості, що після будь-якого оновлення інтерфейсу ОС компоненти додатка також будуть миттєво оновлені.

Тим не менш, це може порушити користувальницький інтерфейс програми. Так, деякі оновлення систем можуть спровокувати помилки, а іноді треба додатково прописувати код для сумісності зі старими версіями ОС.

Написання додатку на Flutter є швидшим, ніж написання такого ж самого додатку у React Native.

React Native використовує міст та нативні елементи інтерфейсу, тому через це може знадобитися окрема оптимізація для кожної платформи. Це проблема, з якою Flutter, створений на основі власних віджетів, не стикається.

Необхідність додаткової отладки та вирішення проблем на різних врсіях операційних систем може зробити розробку додатків за допомогою React Native довшою, ніж на Flutter.

Також існують задачі, з якими фреймворки справляються погано.

Flutter не буде найкращим вибором, якщо у додатку треба імплементувати роботу з 3D об'єктами, або якщо додаток вимагає якісь специфічні задачі зав'язані на внутрішні механізми операційних систем (наприклад, робота з правами адміністратора, або щільна робота з компонентами на кшталт Wi-Fi).

React Native не буде найкращим варіантом також для додатків, які мають великі обчислення у фоновому потоці, потребують багато специфічної роботи з Bluetooth або іншим специфічним механізмом операційних систем.

Що стосується продуктивності, Flutter має перевагу, оскільки він компілюється у нативні бібліотеки ARM або x86, що робить його дійсно швидким. React Native не компілюється до нативного коду, і все ще має рівень JavaScript, який виконується в іншому потоці, що робить його менш продуктивним, ніж Flutter.

React Native працює як обгортка над нативними методами, тому йому потрібен міст для перекладу цих викликів у власний API. Це стає вузьким місцем, коли відбувається багато нативних викликів. Є способи обійти це, але з Flutter ніколи не треба турбуватися про цю проблему.

Виправлення помилок у фреймворку React Native також займає набагато більше часу. Більшість компаній, що використовують React Native у

виробництві, запускають власну гілку з форком (копією) фреймворку, щоб виправити помилки, які не виправляються розробниками з Facebook.

Розробники Flutter є більш активними, і можна очікувати виправлення швидше, ніж у Flutter.

Вихідний сирцевий код обох фреймворків представлений на GitHub. Статистика цього сайту свідчить про те, що станом на листопад 2020 року, репозиторій фреймворку React Native має 91 тисячу зірочок від інших користувачів сервісу. Flutter, в той самий час, 107 тисячу зірок.

Також обидва фреймворка мають приблизно однакову кількість комітів до репозиторію, з не великою перевагою Flutter. Це 21415 комітів у React Native, та 21537 до Flutter.

Але треба взяти до уваги, що Flutter є на кілька років молодшим за React Native. Також, репозиторій React Native має 19 тисяч зачинених проблем та запитань (issues), а Flutter – 38 тисяч зачинених проблем та запитань.

З цього можна зробити висновок, що Flutter розвивається більш швидкими темпами, ніж React Native.

StackOverflow – це найпопулярніший у світі портал для питань та відповідей серед програмістів. Кожного року цей сервіс проводить опитування серед своїх користувачів, щоб досліджувати тренди технологій серед сучасних розробників.

Згідно з опитуванням StackOverflow 2020 серед 65000 розробників, в категорії “Найулюбленіші технології, з якими ви працювали і хоті ли бі працювати і далі” Flutter зайняв 3 місце, з 68.8% відсотками розробників. React Native займає 10 сходинку у рейтингу з 57.9% розробників.

Найближчі конкуренти обох фреймворків, Cordova та Xamarin, займають 16 місце (45.4% розробників) та передостаннє, 18 місце, відповідно (28.7% розробників). Усього в рейтингу представлено 19 технологій [73].

Перші два місця зайняли .Net Core та Torch/pyTorch, які не мають відношення до розробки мобільних додатків.

Тобто, працювати з фреймворком Flutter сподобалося на 10.7% розробників більше, ніж тим розробникам, які працювали з фреймворком React Native (рис. 3.5.).

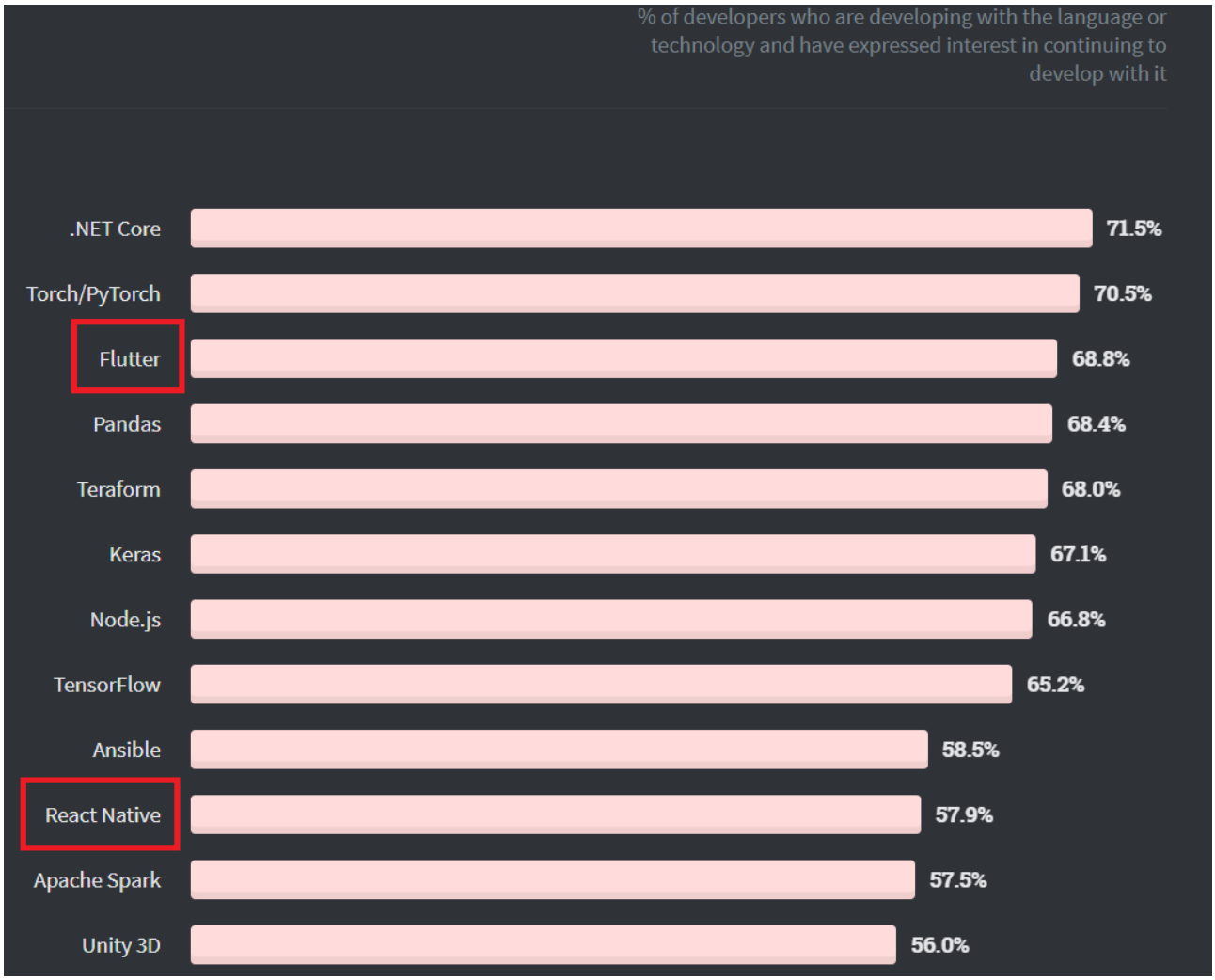

Рис. 3.5. Опитування StackOverflow 2020. Найулюбленіші технології.

І також згідно з даних цього сервісу, станом на жовтень 2020 року, кількість запитань серед користувачів з тегом фреймворку React Native становить 88 тисяч запитань. З тегом фреймворку Flutter – 66 тисяч запитань. Тобто, запитань о React Native існує більше.

Але різниця в виникненні цих фреймворків становить 3 роки, і Flutter поступово наздоганяє React Native за зацікавленістю серед користувачів.

Так, перші 1000 питань о React Native були поставлені у 2015 році. Перша тисяча питань стосовно Flutter – у 2018.

Станом на початок 2020 року, кількість запитань щодо React Native становила 63480 запитання, а щодо Flutter – 31058.

Тобто, за 10 місяців 2020 року, розробники поставили 24844 питання стосовно фреймворку React Native, та 35593 питання щодо фреймворку Flutter (рис. 3.6.).

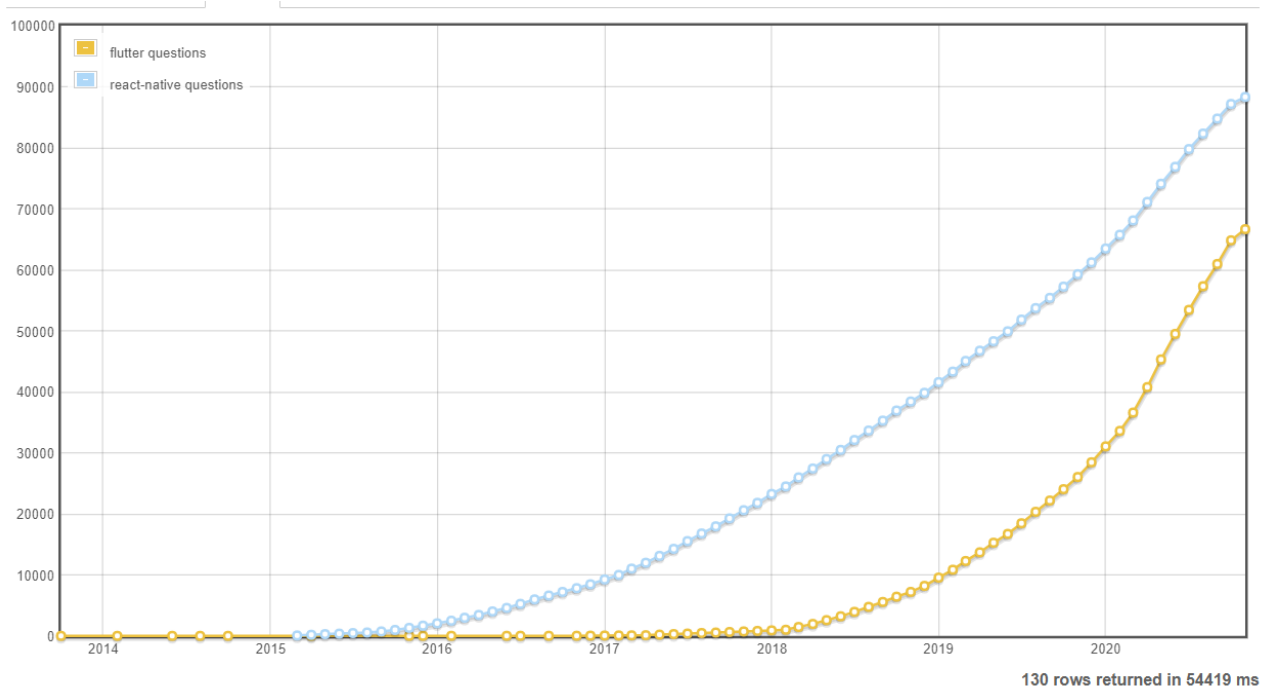

Рис. 3.6. Статистика запитань щодо Flutter та React Native на StackOverflow

Стосовно Flutter за перші 10 місяців 2020 року було поставлено на 10749 запитань більше (або на 43.26% більше), ніж запитань щодо React Native. Тобто, фреймворк Flutter є популярнішим, ніж React Native серед користувачів сервісу StackOverflow [74].

Хоча ще в минулому році, 2019, React Native випереджав Flutter за кількістю нових питань.

Можна зробити прогноз, що кількість запитань щодо Flutter буде тільки збільшуватись, та наздоганяти спільне загальну кількість питань на сервісі StackOverflow (рис. 3.7.).

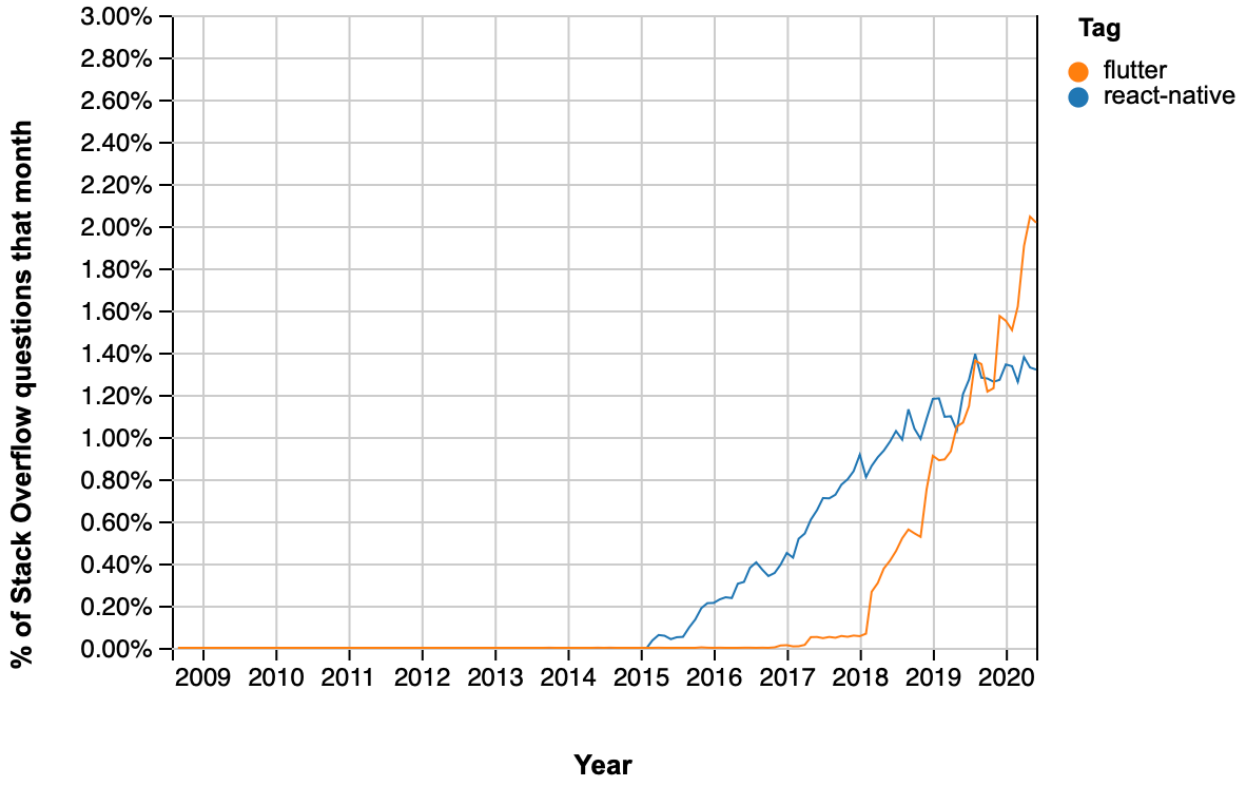

Рис. 3.7. Кількість питань у відсотках протягом місяця на StackOverflow.

Google також має можливість відслідкувати популярність пошукових засобів у своїй пошуковій системі. Для цього Google має свій сервіс Google Trends.

Google Trends має можливість показати поточні тренди та те, як ті або інші пошукові запити набирають або втрачають популярність з часом у різних країнах, порівняно з іншими пошуковими запитами.

В такий спосіб можна досліджувати, як часто шукають ті або інші речі, технології, словосполучення тощо в різних країнах світу.

Так, згідно з даних Google Trends, з листопада 2019 року, користувачі почали шукати Flutter частіше за React Native, і цей тренд стає все сильнішим.

Flutter шукають все частіше, тоді як популярність React Native серед пошукових запитів залишається приблизно незмінною вже декілька років (рис. 3.8.).

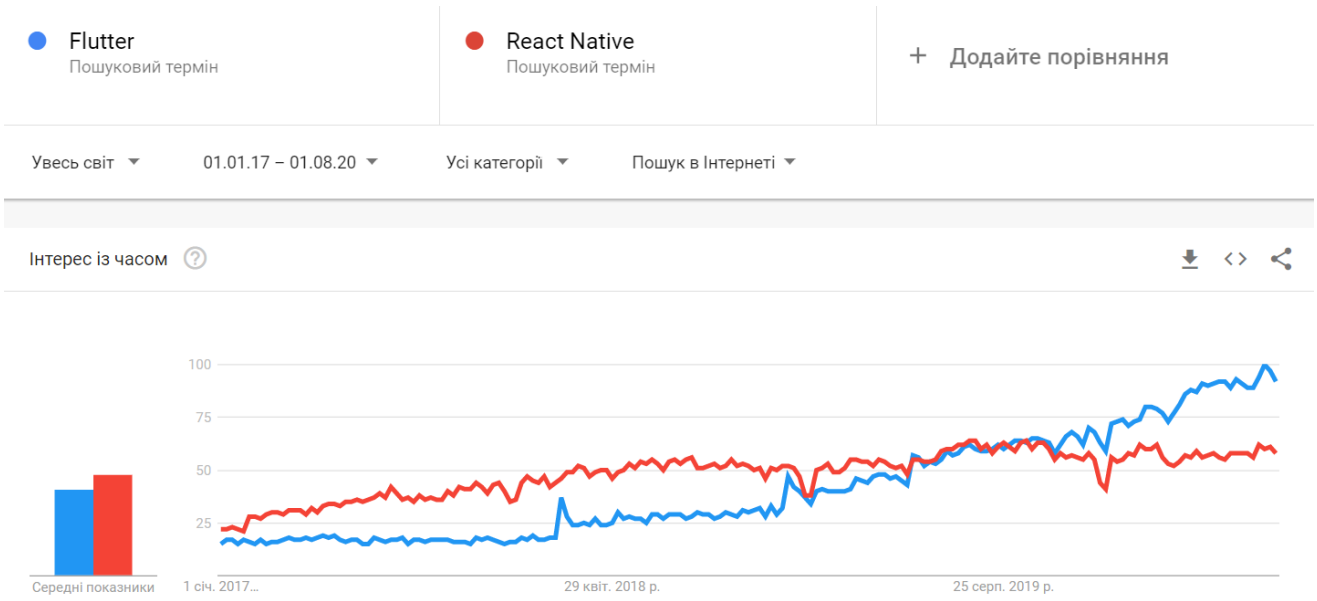

Рис. 3.8. Тренди популярності пошукових запитів «Flutter» та «React Native» за часом.

З цих усіх даних можна зробити висновок, що Flutter став популярнішим за React Native за критеріями трендів пошуку на Google Trends, по кількості запитань на головному сервісі питань та відповідей для розробників StackOverflow, і також має більшу частку задоволених з використання технології розробників згідно різних опитувань.

Але React Native все ще випереджає Flutter по кількості вакансій на ринку.

Так, на сайті Developers of Ukraine (DOU.UA) станом на листопад 2020 року було розміщено 88 вакансій на позицію React Native та 29 вакансій на позицію Flutter по всій Україні.

Тобто, вакансій на позицію розробника React Native у 3 рази більше, ніж вакансій на позицію розробника Flutter.

Це можна пояснити тим, що технологія React Native є старіша на декілька років за Flutter, також одним зі стоп-факторів є потреба вести розробку на більш рідкісною мовою програмування, Dart, коли як React Native використовує JavaScript – одну з найпопулярніших мов програмування у світі (рис. 3.9.).

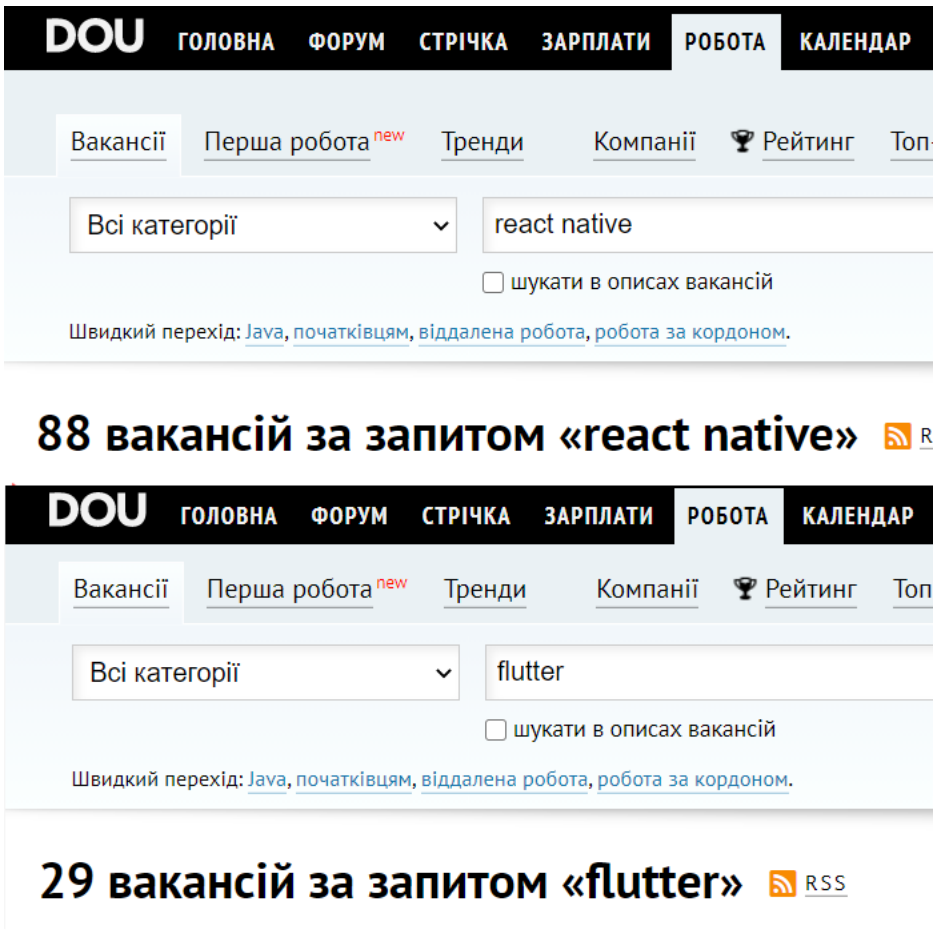

Рис. 3.9. Кількість вакансій на React Native та Flutter на DOU.UA.

Великі компанії використовують як Flutter, так і React Native для своїх продуктів.

Так, React Native використовується компанією Facebook у своїх продуктах. Також, такі популярні додатки як Skype, Pinterest, Soundcloud, Discord (для iOS) використовують React Native.

Але також є компанії, які використовували React Native раніше, але потім переписали свої продукти на Flutter, або повернулися на нативну розробку, як це зробили такі компанії, як AirBnb та Udacity.

Flutter у свою чергу використовується в деяких продуктах Google, таких як Google Ads, Google Stadia та Google Pay (версія для США). Також Flutter використовує Alibaba, Reflectly, Baidu, ByteDance, ebay для розробки своїх додатків, та Sony у embedded-продуктах (рис. 3.10.).

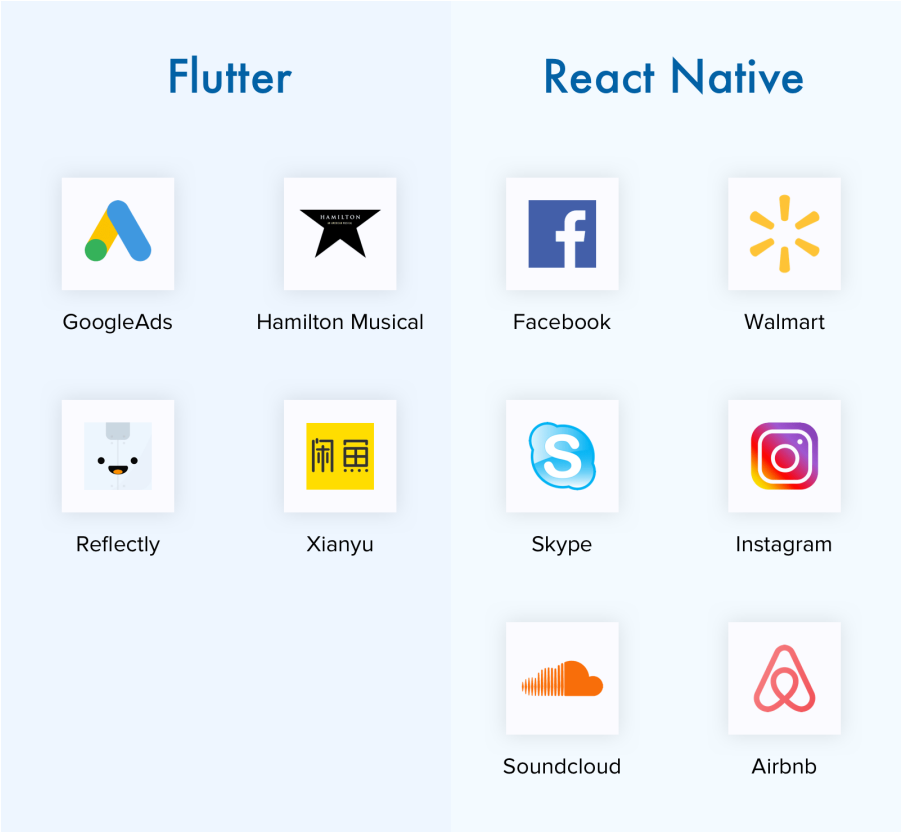

Рис. 3.10. Приклади додатків, написаними за допомогою фреймворків React Native та Flutter.

Також згідно з даними GitHub за 2019 рік (GitHub Octoverse 2019), Dart став мовою програмування, яка набрала найбільшу популярність в використанні відносно себе рок до року у відсотковому відношенні – на 532% збільшилась кількість репозиторіїв на Dart у GitHub (рис. 3.11.) [75].

Статистика та дані на 2020 сервісом GitHub ще не опубліковані (вони публікуються у кінці року).

Частка Flutter поступово збільшується на ринку, бо це молода технологія. Але в першу чергу інтерес до неї приділяють ентузіасти, які готові пробувати нові підходи на власних полігонах та розробники, які зацікавлені в нових технологіях.

Вже зараз можна бачити, що зацікавлення у Flutter вже більше, ніж в усіх інших його конкурентах, але він все ще не має переваги в реальному бізнесі через відносно молоду історію розробки.

Але більш широке використання цього фреймворку на сучасному ринку розробки мобільних додатків більше питання часу.

Рис. 3.11. Топ за ростом кількості репозиторіїв обраною мовою програмування на GitHub.

Все ще більш популярною і прийнятною залишається і нативна розробка, на Kotlin для Android та на Swift для iOS, хоча і кросплатформна розробка зайняла свою бранжу на ринку.

3.2. Розробка плагіну для визначення ефективності засобу Flutter

Так, в рамках кваліфікаційної роботи, був розроблений спеціальний плагін для засобу Flutter, за допомогою якого можна проводити заміри швидкодії мобільних додатків, написаних на Flutter.

Розроблене програмне забезпечення може робити заміри кількості кадрів на секунду (FPS) в режимі реального часу без залежності, чи то debug-збірка мобільного додатку, або release-збірка, яка проходить етапи оптимізації компілятором та має працювати на кінцевих девайсах користувачів.

Також, цей плагін має змогу записувати дані щодо швидкодії рендерингу Flutter-додатку у файл з часовими відрізками.

Дані щодо швидкодії рендерингу програмного забезпечення можна виводити на екран девайсу, або збирати ці дані у фоні та зберігати до файлу або відправляти через мережу.

Для зручного користування цим плагіном іншими розробниками, було спроєктовано зручний API для отримання цих даних в будь-якій точці додатку з допомогою шаблону проєктування Observer.

Observer ("Спостерігач") – патерн, або шаблон проєктування, який надає механізм підписки, тобто дає можливість одним об’єктам стежити й реагувати на події, які відбуваються в інших об’єктах.

Користувач підписується на зміни швидкодії та починає отримувати їх у спеціальну callback-функцію, тобто функцію зворотного виклику в наступний спосіб:

```
FpsPlugin.instance.registerForUpdates((fps, dropCount) {
      // нові значення швидкодії fps та її зменшення propCount
});
```

Цей плагін здатен працювати на обох найпопулярніших мобільних операційних системах – Android та iOS.

Для реалізації цією сумісності з обома ОС, крім основного коду плагіну на Dart, також було реалізовано інтерфейси, які здатні надавати Flutter-коду інформацію щодо системи. Ці інтерфейси мають бути реалізовані нативною мовою програмування під кожну платформу окремо, щоб далі ці дані можна було використовувати у Flutter в уніфікованому вигляді під всі платформи.

Код для Android:

```
        @Override
        public void onMethodCall(@NonNull MethodCall call, @NonNull Result
result) {
          if (call.method.equals("getRefreshRate")) {
            try {
              WindowManager windowManager = (WindowManager)
context.getSystemService(Context.WINDOW_SERVICE);

              float refreshRate =
windowManager.getDefaultDisplay().getRefreshRate();
              result.success(refreshRate);
            } catch (Exception e) {
              result.success(null);
            }
          } else {
            result.notImplemented();
          }
        }
```

Для iOS:

```
- (double)displayRefreshRate:(CADisplayLink *)link {
        NSInteger preferredFPS = link.preferredFramesPerSecond;
        // maximumFramesPerSecond property.
        if (preferredFPS != 0) {
            return @(preferredFPS).doubleValue;
        }
        return @([UIScreen
mainScreen].maximumFramesPerSecond).doubleValue;
}
```

І вже у Flutter у плагіні за допомогою мови Dart можна робити запит на отримання даних з нативних платформ, і використовувати їх мультиплатформенно:

```
static Future<double> get getRefreshRate async {
    final double fpsHz = await _channel.invokeMethod('getRefreshRate');
    return fpsHz;
```

```
}
```

Також, крім плагіну, був написаний тестовий додаток за допомогою Flutter, який реалізує комунікацію з сервером відкритої бази даних фільмів.

Цей додаток реалізує найбільш популярні задачі під час розробки мобільних додатків: спілкування з сервером за допомогою HTTP-запитів, відображення відповідей з серверу у вигляді списку, відтворення більш детальних даних по пунктам списку та інше.

Для додатку на Flutter, який буде працювати на Android та iOS, для реалізації цієї задачі знадобилося близько 350 строк коду (Рис. 3.12.). Для створення подібного нативного подібного додатку, який буде працювати тільки на Android – близько 1050 строк коду.

Проєкт на React Native має приблизно таку ж саму кількість строк програмного коду, як і проєкт на Flutter, бо вони використовують приблизно однакові принципи при побудові інтерфейсів, і також задіє стандартну бібліотеку фреймворку.

Також, вихідний APK файл написаний на нативному Андроїд важить 3,4 мегабайти. Тоді як APK з додатком на Flutter – 8 мегабайт.

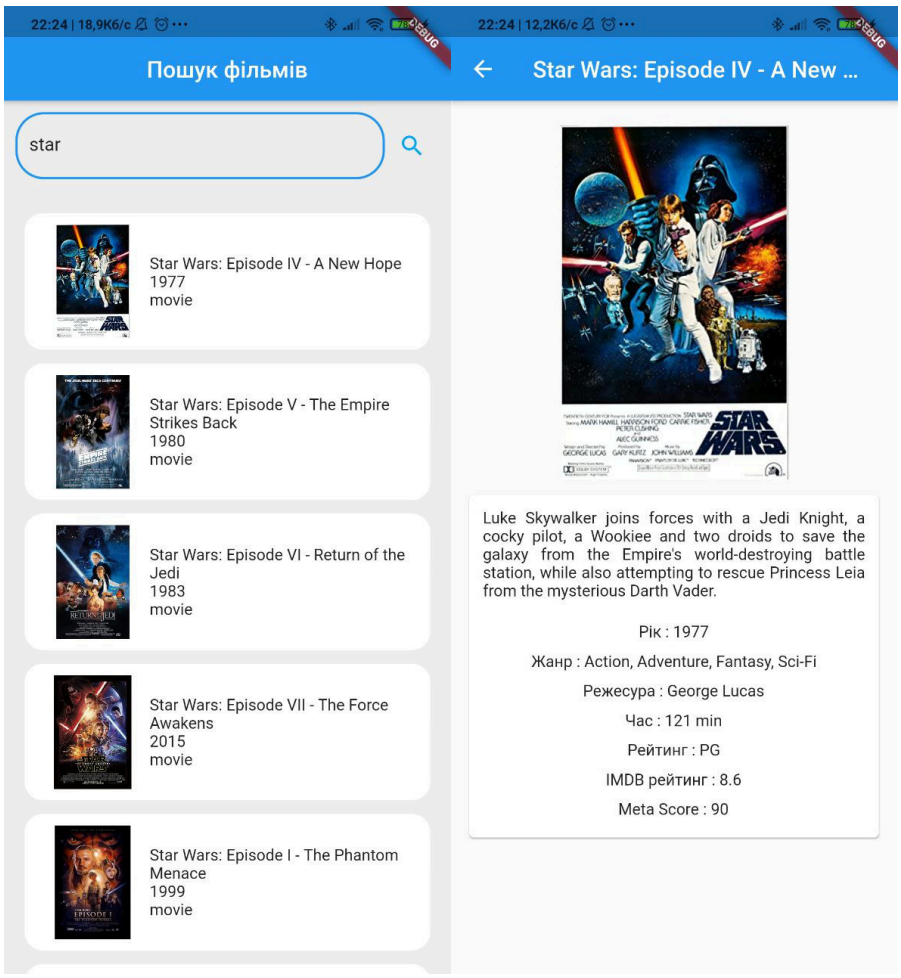

Рис. 3.12. Скриншот розробленого додатку на Flutter.

Цей додаток використовує тільки стандартну бібліотеку Flutter та пакет http для взаємодії з сервером через спеціальне API. Зміни станів віджетів відбуваються через стандартний механізм станів у StatefulWidget та реалізацію callback-функцій, які викликаються при отримані результату з серверу, за допомогою FutureBuilder.

Подібний приклад, який був написаний мовою програмування Java, використовує сторонню бібліотеку Retrofit для зв’язку з серверною частиною разом зі спеціальною бібліотекою для роботи з форматом json – бібліотекою Gson.

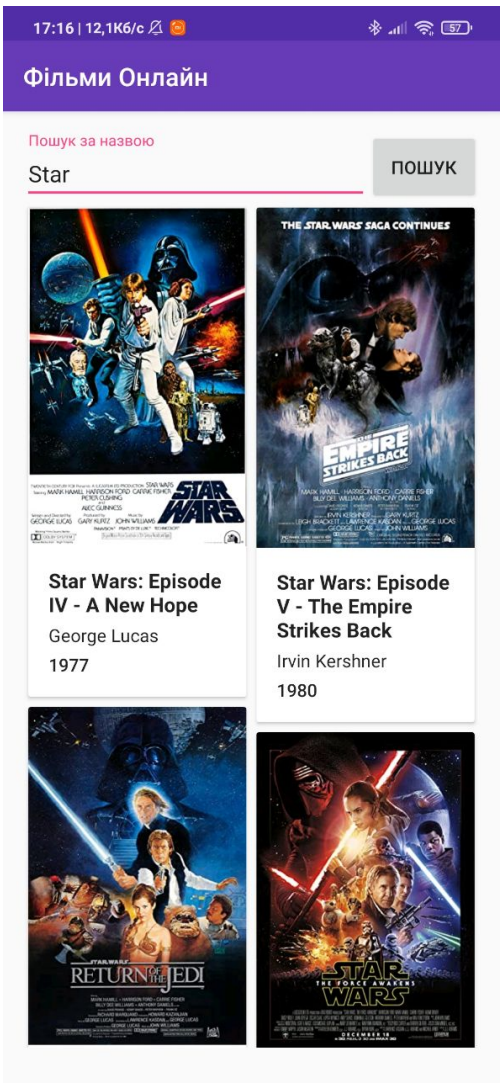

Рис. 3.13. Скриншот додатку написаного на нативному Android.

У Flutter та React Native присутній подібний функціонал у стандартній бібліотеці, тому вони не потребують зайвих імпортів.

Загальна кількість класів та файлі у нативному проєкті на Андроїд є більшою, ніж для проєктів на Flutter та React Native, бо нативний проєкт включає в себе також багату кількість XML-файлів з верстокою візуального інтерфейсу користувача.

Так, використовуючи плагін для визначення швидкодії додатків на Flutter, який був розроблений в цьому розділі, на тестовому додатку було отримано стабільні 60 кадрів на секунду без падіння швидкодії при таких складних кейсах, як скролінг великого списку результатів пошуку з зображеннями (рис. 3.14.).

Ці результати відповідають нативній швидкодії додатків під мобільні операційні системи Android та iOS, тобто відтворення кожного кадру раз на 16,6 мілісекунд на стандартних дисплеях з частотою оновлення 60 Гц.

```
[log] FPS=60.000003814697266, dropCount=0.0
[log] FPS=60.000003814697266, dropCount=0.0
[log] FPS=60.000003814697266, dropCount=0.0
[log] FPS=60.000003814697266, dropCount=0.0
[log] FPS=60.000003814697266, dropCount=0.0
```

Рис. 3.14. Результати тестів швидкодії з плагіном у Flutter-додатку.

3.3. Висновки до третього розділу

В цьому розділі були досліджені засоби розробки мобільних додатків React Native та Flutter та їх порівняння між собою.

Були досліджені різниці в підходах, процесі розробки та у підтримці існуючих додатків. Також Flutter був порівняний з нативною розробкою мобільних додатків та різницю в підходах між ними. Також була досліджена різниця в принципах побудови графічного інтерфейсу користувача, та розібран декларативний підхід в розробці інтерфейсів.

Крім того, був розроблений плагін для Flutter, за допомогою якого можливо досліджувати швидкодію та ефективність роботи мобільних додатків, написаними за допомогою засоба Flutter.

Цей плагін здатен реєструвати швидкодію рендерингу мобільного додатку на кінцевому девайсі користувача в debug та release збірках, виводити дані на екран або у логи девайсу і також записувати швидкодію з плином часу.

Цей плагін може використовуватися в будь-якому проєкті на Flutter іншими розробниками для вимірювання швидкодії та працює як на Android, так і на iOS.

Також, для прикладу були розроблені тестові мобільні додатки, які взаємодіють з сервером та виводять списки та інформацію на екран користувача.

Крім цього, були проведені дослідження щодо популярності та трендів підходів Flutter та React Native серед співтовариства розробників у GitHub, StackOverflow, DOU тощо.

З чого можна зробити висновки, що технологія Flutter вже випереджає React Native за популярністю серед розробників, але в бізнесі все ще частіше використовують React Native через відносно молодий вік Flutter.

РОЗДІЛ 4
ЕКОНОМІЧНА ЧАСТИНА

При розробці програмного забезпечення важливими етапами є визначення трудомісткості розробки ПЗ і розрахунок витрат на створення програмного продукту.

4.1. Визначення трудомісткості розробки програмного забезпечення

Задані дані:

1. передбачуване число операторів – 310;
2. коефіцієнт складності програми – 1,3;
3. коефіцієнт корекції програми в ході її розробки – 0,3;
4. годинна заробітна плата програміста, грн/год – 90;
5. коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі – 1,1;
6. коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності – 1,5;
7. вартість машино-години ЕОМ, грн/год – 15 грн.

Нормування праці в процесі створення ПЗ істотно ускладнено в силу творчого характеру праці програміста. Тому трудомісткість розробки ПЗ може бути розрахована на основі системи моделей з різною точністю оцінки.

Трудомісткість розробки ПЗ можна розрахувати за формулою:

$$t=t_o+t_и+t_a+t_п+t_{отл}+t_д \text{ , людино-годин,} \quad (4.1)$$

де t_o - витрати праці на підготовку й опис поставленої задачі (приймається 50);

$t_и$ - витрати праці на дослідження алгоритму рішення задачі;

t_a - витрати праці на розробку блок-схеми алгоритму;

$t_п$ - витрати праці на програмування по готовій блок-схемі;

$t_{отл}$ - витрати праці на налагодження програми на ЕОМ;

$t_д$ - витрати праці на підготовку документації.

Складові витрати праці визначаються через умовне число операторів у ПЗ, яке розробляється.

Умовне число операторів (підпрограм):

$$Q = q * C * (1 + p), \tag{4.2}$$

де q - передбачуване число операторів;

C - коефіцієнт складності програми;

p - коефіцієнт корекції програми в ході її розробки.

$$Q=310*1,3*(1+0,3)= 523,9, \text{ людино-годин} \tag{4.3}$$

Витрати праці на вивчення опису задачі $t_и$ визначається з урахуванням уточнення опису і кваліфікації програміста:

$$t_u=\frac{Q * B}{(75 \dots 85) * k}, \text{ людино-годин,} \tag{4.4}$$

де B - коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі;

k - коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності.

$$t_u=\frac{523,9*1,1}{75*1,5}=\frac{576,29}{112,5}=4, \text{ людино-годин} \tag{4.5}$$

Витрати праці на розробку алгоритму рішення задачі:

$$t_a = \frac{Q}{(20 \ldots 25) * k}, \text{ людино-годин,} \quad (4.6)$$

$$t_a = \frac{523,9}{23*1,5} = 15, \text{ людино-годин} \quad (4.7)$$

Витрати на складання програми по готовій блок-схемі:

$$t_n = \frac{Q}{(20 \ldots 25)*k}, \text{ людино-годин} \quad (4.8)$$

$$t_n = \frac{523,9}{20 * 1,5} = 17, \text{ людино-годин} \quad (4.9)$$

Витрати праці на налагодження програми на ЕОМ:

- за умови автономного налагодження одного завдання:

$$t_{отл} = \frac{Q}{(4 \ldots 5)*k}, \text{ людино-годин,} \quad (4.10)$$

$$t_{отл} = \frac{523,9}{4 * 1,5} = 87, \text{ людино-годин} \quad (4.11)$$

- за умови комплексного налагодження завдання:

$$t_{отл}^{k} = 1,5 * t_{отл}, \text{ людино-годин.} \quad (4.12)$$

$$t_{отл}^{k} = 1,5*87,3 = 130, \text{ людино-годин} \quad (4.13)$$

Витрати праці на підготовку документації:

$$t_д = t_{др} + t_{до}, \text{ людино-годин,} \quad (4.14)$$

де $t_{др}$ - трудомісткість підготовки матеріалів і рукопису.

$$t_{др} = \frac{Q}{15..20*k}, \text{ людино-годин.} \quad (4.15)$$

$$t_{др} = \frac{523,9}{17*1,5} = 20, \text{ людино-годин} \quad (4.16)$$

$t_{до}$ - трудомісткість редагування, печатки й оформлення документації

$$t_{до} = 0,75 * t_{др}, \text{ людино-годин} \quad (4.17)$$

$$t_{до} = 0,75*20 = 15, \text{ людино-годин} \quad (4.18)$$

$$t_д = 20+15 = 35, \text{ людино-годин} \quad (4.19)$$

Тепер розрахуємо трудомісткість ПЗ:

$$t = 4+15+17+87+35 = 158, \text{ людино-годин.} \quad (4.20)$$

4.2. Витрати на створення програмного забезпечення

Витрати на створення ПЗ включають витрати на заробітну плату виконавця програми з/п і витрат машинного часу, необхідного на налагодження програми на ЕОМ.

$$К_{по}=З_{зп}+З_{мв}, грн. \quad (4.21)$$

Заробітна плата виконавців визначається за формулою:

$$З_{зп}=t*С_{пр}, грн, \quad (4.22)$$

де: t - загальна трудомісткість, людино-годин;

$С_{пр}$ - середня годинна заробітна плата програміста, грн/година

$$З_{зп}=158*70=11060, грн. \quad (4.23)$$

Вартість машинного часу, необхідного для налагодження програми на ЕОМ:

$$З_{мв}=t_{отл}*С_{мч}, грн, \quad (4.24)$$

де $t_{отл}$ - трудомісткість налагодження програми на ЕОМ, год.

$С_{мч}$ - вартість машино-години ЕОМ, грн/год.

$$З_{мв}=87*15=1305, грн. \quad (4.25)$$

Визначені в такий спосіб витрати на створення програмного забезпечення є частиною одноразових капітальних витрат на створення АСУП.

$$К_{по}=11060+1305=12365, грн. \quad (4.26)$$

Очікуваний період створення ПЗ:

$$T=\frac{t}{B_k*F_p}, \text{ міс}, \quad (4.27)$$

де B_k - число виконавців;

F_p - місячний фонд робочого часу (при 40 годинному робочому тижні F_p=176 годин).

$B_k = 1$

$$T=\frac{158}{1*176}=0,89, \quad \text{міс} \quad (4.28)$$

4.3. Маркетингові дослідження

Сучасному бізнесу важливо мати постійний зв’язок зі своїми клієнтами та мати зручний інтерфейс, через який клієнти та бізнес можуть взаємодіяти у найбільш зручний спосіб. Сьогодні одним з таких засобів є мобільні додатки для смартфонів та планшетів.

Смартфон – це та річ, яка є у кишені більшості людей: і це та сама річ, через яку бізнес та клієнти можуть взаємодіяти один з одним. І сьогодні для збільшення кількості клієнтів та продажів, для поліпшення якості взаємодії з клієнтами все більш популярними, крім сторінок у соціальних мережах, стають і мобільні додатки, які можуть грати роль того інструменту, через який клієнти здатні робити замовлення, отримувати зворотній зв’язок, знайомитися з послугами та/або асортиментом бізнесу, отримувати промо-матеріали, знижки тощо. І при цьому не залежати від стороннього сервісу (наприклад, тільки від стороннього маркетплейсу).

Але малий та середній бізнес доволі часто стискається з проблемою, коли він не має достатніх коштів на розробку свого власного мобільного додатку одразу під декілька платформ одночасно.

Щоб розробити функціонально однакові мобільні додатки під і Android, і iOS, бізнесу, який хоче мати власний додаток, треба найняти або оплачувати роботу двох окремих команд розробників, які будуть паралельно вести розробку додатків кожна під свою цільову платформу.

Після основної розробки додатку та виходу на ринок, постане питання періодичних оновлень функціоналу додатків, і для цього знову ж таки треба залучати дві окремі команди, що загрожує зростом витрат та зайвими тратами на подальшу підтримку мобільних додатків.

Ці проблеми роблять наявність власних мобільних додатків доволі дорогим задоволенням для більшості малого та навіть середнього бізнесу.

В рамках кваліфікаційної роботи було розглянуто метод, який покликаний вирішити цю проблему – фреймворк для кросплатформної розробки мобільних додатків Flutter.

Він дозволяє використовувати одну кодову базу для розробки додатків для iOS та Android, а також для створення вебсайтів та комп'ютерних програм одночасно.

Це означає, що замість 2-3 команд для створення мобільних додатків та сайту, тепер достатньо однієї команди (а іноді і однієї людини-розробника) для того, щоб створити мобільний додаток для двох основних операційних систем, та навіть вебсайт.

І при цьому, цей фреймворк надає справжню однаковість кодової бази між платформами та швидкодію додатків на рівні рідних додатків платформи, які писали б різні команди розробників під кожну платформу окремо.

Переваги використання Flutter для розробки мобільних додатків наступні:

- Справжня однаковість коду під різні платформи.
- Дружнє та чуйне співтовариство розробників, які допомагають новачкам та швидко вирішують проблеми з фреймворком.
- Швидкодія на рівні нативних додатків, які розроблялися спеціально під обрану платформу нативно (рідним шляхом, як це задумали розробники операційних систем).

- Спромога розробляти додатки силами однієї команди замість двох або трьох окремих.

Це все вигідно відрізняє фреймворк Flutter серед своїх найближчих конкурентів: таких як React Native, Xamarin, Cordova.

Також в сумі розробка на Flutter дозволяє писати програми швидше, ніж за допомогою нативних методів для кожної платформи окремо.

І додаток, написаний на Flutter для обох платформ, потребує менше коду, ніж один нативний додаток, написаний для однієї платформи (в нашому прикладі – на Android).

Більш детально про цей йшлося у попередніх розділах даної роботи.

4.4. Оцінка економічної ефективності

Оскільки в даній кваліфікаційній роботі була розглянута тільки можливість використання фреймворка Flutter та плагіну для вимірювання ефективності, розрахунок точного економічного ефекту, який складається з витрат до впровадження ПЗ і після нього, неможливий. Ці розрахунки залежать від кожного окремого проєкту, в якому використовується ці методи та підходи до розробки мобільних додатків.

Тому розглядається тільки соціальний ефект.

Було проведено дослідження, які показали ефективність використання фреймворка Flutter для розробки мобільних додатків під декілька мобільних операційних систем одночасно, а також зріст популярності та зацікавленості в цій технології: незважаючи на її молодий вік, порівняно з іншими існуючими рішеннями на ринку.

В зв’язку з цим, ця технологія дозволяє витрачати менше часу та людських ресурсів на створення та подальшу підтримку мобільних додатків, за допомогою того, що використовується єдина кодова база, а не 2-3 незалежних одна від іншої кодових баз, які виконують однакові функції, але тільки на різних платформах.

Також це відбувається без значних втрат у швидкодії вихідного програмного забезпечення, порівняно з іншими подібними рішеннями на ринку (такими, як React Native, Xamarin, Cordova та інші).

Також, додаток, написаний на Flutter для обох платформ, потребує менше коду, ніж один нативний додаток, написаний для однієї платформи.

А подальша підтримка такого програмного забезпечення є простішою, що було досліджено в попередніх розділах цієї кваліфікаційної роботи.

Це дозволяє надавати клієнтам, які потребують розробку мобільного додатку для своїх потреб, більше можливостей для розробки швидшими темпами та за участю уніфікованої інфраструктури для кожної мобільної платформи.

ВИСНОВКИ

За проведеним аналізом можна зробити висновки, що технології та принципи, які використовує Flutter, є більш перспективними, надійними, зручними та швидкими до використання при розробці кросплатформенних мобільних додатків, ніж ті, що використовує його найближчий та найпопулярніший аналог – React Native.

Фреймворк Flutter випереджає React Native у розробці мобільний додатків за наступними параметрами:

- Продуктивність та швидкодія створеного програмного забезпечення.
- Простота проєктування та розробки користувацького інтерфейсу (UI).
- Час, потрібний для розробки продукту.
- Надійність та краща підтримка від розробників та товариством розробників.
- Поточний тренд зацікавленності у технології стрімко зростає та вже випередив React Native.
- Фреймворк має можливість компіляції під десктопні операційні системи (Windows, macOS, Linux) та збірку під вебдодаток, який відтворюється у браузері.

В свою чергу, React Native продовжує випереджати Flutter за наступними параметрами:

- Поточна кількість розробників, які здатні розробляти програму цією технологією більша, ніж на Flutter.
- Використання більш поширеної мови програмування (JavaScript замість Dart).
- Деякі частини логіки можна перевикористовувати у вебдодатках, які написані за допомогою бібліотеки React.js.
- Все ще більше вакансій на ринку, ніж у Flutter.

Також Flutter є доброю альтернативною більш дорожчої та довшої нативної розробки під кожну платформу окремо, якщо розроблюваний

мобільний додаток не використовує 3D графіку, та не взаємодіє на низькому рівні з hardware-частиною мобільних пристроїв. Тобто, якщо додаток більше зав’язаний на взаємодію з віддаленим сервером.

Також, в результаті роботи, було створено прототип плагіну для Flutter, який дозволяє визначати швидкодію розроблених кросплатформенних мобільних додатків засобом Flutter під операційними системами iOS та Android. Використані методи та підходи можуть застосовуватися при розробці програмного забезпечення під мобільні пристрої на замовлення бізнесу, щоб розробка мобільних додатків проходила швидшими та ефективнішими методами. Бібліотеку можна використовувати в будь-яких проєктах, що використовують Flutter.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Shaun Lewis, Mike Dunn. Native Mobile Development: A Cross-Reference for iOS and Android. – Chapter 1 – 2019 – P. 1-3.

2. Shaun Lewis, Mike Dunn. Native Mobile Development: A Cross-Reference for iOS and Android. – Chapter 1 – 2019 – P. 4-5.

3. Niranjan Kumar. Learn Android. – Chapter 1 – 2020 – P. 4-5.

4. Nate Ebel. Mastering Kotlin. – Chapter 13 – 2019 – P. 267.

5. Peter Späth. Pro Android with Kotlin: Developing Modern Mobile Apps. – Chapter 6 – 2018 – P. 73.

6. Ole Henry Halvorsen, Douglas Clarke. OS X and iOS Kernel Programming. –Vol. 1 – 2012 – P. 16-17.

7. Carlos Oliveira. Objective-C Programmer's Reference. – Chapter 3 – 2014 – P. 30.

8. Steve Derico. Introducing IOS 8: Swift Programming from Idea to App Store. – Vol. 3 – 2014 – P. 55-56.

9. Sean Liao. Migrating to Swift from Android. – Chapter 1 – 2014 – P. 3-5.

10. Mahesh Panhale. Beginning Hybrid Mobile Application Development. – Chapter 12 – 2015 – P. 199-203.

11. John M. Wargo. Apache Cordova API Cookbook. – Vol. 1 – 2014 – P. 1-3.

12. Patricia Pesado, Claudio Aciti. Computer Science – CACIC 2018: 24th Argentine Congress. – 2018 – P. 174.

13. PJeng-Shyang Pan, Jerry Chun-Wei Lin, Bixia Sui. Genetic and Evolutionary Computing. – Chapter 1 – 2019 – P. 148.

14. John M. Wargo. Learning Progressive Web Apps. – Chapter 8 – 2020 – P. 150.

15. Gaurav Saini. Hybrid Mobile Development with Ionic. – Chapter 8 – 2017 – P. 199.

16. Eric Masiello, Jacob Friedmann. Mastering React Native. – Chapter 10 – 2017 – P. 384.

17. Eric Masiello, Jacob Friedmann. Mastering React Native. – Chapter 2 – 2017 – P. 45.

18. Eric Masiello, Jacob Friedmann. Mastering React Native. – Chapter 2 – 2017 – P. 47-50.

19. Eric Masiello, Jacob Friedmann. Mastering React Native. – Chapter 11 – 2017 – P. 441-443.

20. Dan Hermes. Xamarin Mobile Application Development. – Vol. 1 – 2015 – P. 1-5.

21. Jan Rabe. Everything that is wrong with Xamarin and why it is bad for you. – Medium – 2018.

22. Prajyot Mainkar, Salvatore Giordano. Google Flutter Mobile Development Quick Start Guide. – Chapter 1 – 2019 – P. 5-10.

23. Marco L. Napoli. Beginning Flutter: A Hands On Guide to App Development. – Chapter 2 – 2019 – P. 44-45.

24. Barry Burd. Flutter For Dummies. – Part 1 – 2020 – P. 22-23.

25. Google Trends "Flutter vs React Native" – [Електронний ресурс] – Режим доступу: https://trends.google.com/trends/explore?date=2017-01-01%202020-08-01&q=Flutter,React%20Native.

26. Marco L. Napoli. Beginning Flutter: A Hands On Guide to App Development. – Chapter 1 – 2019 – P. 4.

27. Frank Zammetti. Practical Flutter: Improve your Mobile Development. – Chapter 1 – 2019 – P. 17.

28. Alessandro Biessek. Flutter for Beginners: An introductory guide. – Chapter 3 – 2019 – P. 99.

29. Gilad Bracha. The Dart Programming Language. – Chapter 1 – 2015 – 9-15.

30. Marco L. Napoli. Beginning Flutter: A Hands On Guide to App Development. – Chapter 1 – 2019 – P. 5-6.

31. Marco L. Napoli. Beginning Flutter: A Hands On Guide to App Development. – Chapter 1 – 2019 – P. 44.

32. Frank Zammetti. Practical Flutter: Improve your Mobile Development. – Chapter 1 – 2019 – P. 4-5.

33. Alessandro Biessek. Flutter for Beginners: An introductory guide. – Chapter 1 – 2019 – P. 9.

34. Flutter Документація – [Електронний ресурс] – Режим доступу: https://flutter.dev/docs/resources/architectural-overview.

35. Frank Zammetti. Practical Flutter: Improve your Mobile Development. – Chapter 1 – 2019 – P. 16.

36. Marco L. Napoli. Beginning Flutter: A Hands On Guide to App Development. – Chapter 1 – 2019 – P. 22.

37. Chris Sells. Canonical enables Linux desktop app support with Flutter. – Medium Flutter – 2020.

38. Marco L. Napoli. Beginning Flutter: A Hands On Guide to App Development. – Chapter 14 – 2019 – P. 376.

39. Hidenori Matsubayashi. Graphical User Interface Using Flutter in Embedded Systems. – Embedded Linux Conference Europe. – 2020.

40. Flutter Showcase – [Електронний ресурс] – Режим доступу: https://flutter.dev/showcase.

41. Fu Cheng. Flutter Recipes: Mobile Development Solutions for iOS and Android. – Chapter. 4 – Part 4.2 – 2019 – P. 106.

42. Marco L. Napoli. Beginning Flutter: A Hands On Guide to App Development. – Chapter 1 – 2019 – P. 10-12.

43. Fu Cheng. Flutter Recipes: Mobile Development Solutions for iOS and Android. – Chapter. 4 – Part 4.3 – 2019 – P. 110-113.

44. Fu Cheng. Flutter Recipes: Mobile Development Solutions for iOS and Android. – Chapter. 5 – Part 5.1 – 2019 – P. 137.

45. Fu Cheng. Flutter Recipes: Mobile Development Solutions for iOS and Android. – Chapter. 5 – Part 5.2 – 2019 – P. 139-140.

46. Barry Burd. Flutter For Dummies. – Part 2 – Chapter 3 – 2020 – P. 80-81.

47. Simone Alessandria. Flutter Projects: A practical, project-based guide. – Chapter 2 – 2020 – P. 54-55.

48. Alessandro Biessek. Flutter for Beginners: An introductory guide. – Chapter 6 – 2019 – P. 200.

49. Rap Payne. Beginning App Development with Flutter. – Chapter 9 – 2019 – P. 200-201.

50. Christopher G. Lasater. Design Patterns. – Chapter 3 – 2006 – P. 145.

51. Simone Alessandria. Flutter Projects: A practical, project-based guide. – Chapter 10 – 2020 – P. 379.

52. Marco L. Napoli. Beginning Flutter: A Hands On Guide to App Development. – Chapter 16 – 2019 – P. 432.

53. Akshat Paul, Abhishek Nalwaya. React Native for iOS Development. – Chapter 1 – 2015 – P. 20-21.

54. Дмитрий Беспалов, Наталия Коробейникова. Операционные системы реального времени и технологии разработки кроссплатформенного программного обеспечения. – Часть 2 – 2019 – С. 163.

55. Eric Masiello, Jacob Friedmann. Mastering React Native. – Chapter 10 – 2017 – P. 352.

56. Akshat Paul, Abhishek Nalwaya. React Native for Mobile Development. – Chapter 1 – 2019 – P. 37.

57. Frank Zammetti. Practical React Native: Build Two Full Projects. – Chapter 1 – 2018 – P. 21.

58. Vladimir Novick. React Native – Building Mobile Apps with JavaScript. – Chapter 4 – 2017 – P. 114.

59. Vladimir Novick. React Native – Building Mobile Apps with JavaScript. – Chapter 1 – 2017 – P. 15.

60. Frank Zammetti. Practical React Native: Build Two Full Projects. – Chapter 1 – 2018 – P. 22-23.

61. Vladimir Novick. React Native – Building Mobile Apps with JavaScript. – Chapter 2 – 2017 – P. 46-47.

62. Vladimir Novick. React Native – Building Mobile Apps with JavaScript. – Chapter 3 – 2017 – P. 63-65.

63. Vladimir Novick. React Native – Building Mobile Apps with JavaScript. – Chapter 2 – 2017 – P. 45-49.

64. Vladimir Novick. React Native – Building Mobile Apps with JavaScript. – Chapter 2 – 2017 – P. 54-55.

65. Roy Derks. React Projects: Build 12 real-world applications. – Chapter 12 – 2019 – P. 416.

66. Vladimir Novick. React Native – Building Mobile Apps with JavaScript. – Chapter 9 – 2017 – P. 291.

67. Gabriel Peal. Sunsetting React Native. – Medium – 2018.

68. Nate Ebel. React Native: A retrospective from the mobile-engineering team at Udacity. – Udacity Engineering – 2018.

69. Frank Zammetti. Practical React Native: Build Two Full Projects. – Chapter 1 – 2018 – P. 27.

70. Vladimir Novick. React Native – Building Mobile Apps with JavaScript. – Chapter 8 – 2017 – P. 225-227.

71. Vladimir Novick. React Native – Building Mobile Apps with JavaScript. – Chapter 8 – 2017 – P. 228-232.

72. Alessandro Biessek. Flutter for Beginners: An introductory guide. – Chapter 1 – 2019 – P. 23.

73. StackOverflow Survey 2020 – [Електронний ресурс] – Режим доступу: https://insights.stackoverflow.com/survey/2020.

74. StackOverflow Data by Query – [Електронний ресурс] – Режим доступу: https://data.stackexchange.com/stackoverflow/query/201360/question-count-or-score-growth-over-time-by-tag-comparison?ShowScore=1&Tag1=flutter&Tag2=react%2BnativeTag3=NA&Tag4=NA#graph.

75. GitHub Octoverse 2019 – [Електронний ресурс] – Режим доступу: https://octoverse.github.com/#top-languages

ДОДАТОК А

ЛІСТИНГ ПРОГРАМ

Приклад додатку на Flutter мовою Dart, який виводить текст та кнопку:

```
import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Welcome to Flutter',
      home: Scaffold(
        appBar: AppBar(
          title: Text('Welcome to Flutter'),
        ),
        body: Center(
          child: Column(
               children: [
                      Text('Hello World'),
                      SizedBox(height: 20),
                      ElevatedButton(
                              onPressed: () {
                                     print('Click);
                              },
                              child: Text('A button),
                      ),
               ],
           ),
        ),
      ),
    );
  }
}
```

Приклад JSX-коду, який використовує React Native:

```
<Button color="blue" shadowSize={2}>
  Click Me
</Button>
```

Приклад в що транслюється JSX код у React Native:

```
React.createElement(
```

```
Button,
{color: 'blue', shadowSize: 2},
'Click Me'
);
```

Приклад додатку «Hello World» у React Native:

```
import React from 'react';
import { Text, View } from 'react-native';
import ToolbarAndroid from '@react-native-community/toolbar-android';

const HelloWorldApp = () => {
  return (
    <ToolbarAndroid
      title="AwesomeApp">
               <View
                 style={{
                      flex: 1,
                      justifyContent: "center",
                      alignItems: "center"
                 }}>
                 <Text>Hello, world!</Text>
               </View>
       </ToolbarAndroid>
  )
}
export default HelloWorldApp;
```

Приклад додатку «Hello World» у Flutter:

```
import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Welcome to Flutter',
      home: Scaffold(
        appBar: AppBar(
          title: Text('Welcome to Flutter'),
        ),
        body: Center(
          child: Text('Hello World'),
        ),
      ),
    );
  }
}
```

Приклад роботи зі станом у Flutter за використанням StatefulWidget та State у віджетів:

```
MyHomePage({Key key, this.title}) : super(key: key);

 final String title;

 @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
          children: <Widget>[
               Text('You have pushed the button this many times:'),
               Text('$_counter'),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        child: Icon(Icons.add),
      ),
    );
  }
}
```

Приклад роботи зі станом у React Native за використанням State компонента:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Привіт, світе!</h1>
        <h2>Зараз {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Код розробленого додатку на Flutter для пошуку фільмів.

```
class MoviesApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
```

```
    return MaterialApp(
      title: 'Пошук фільмів',
      home: MoviesAppHome(),
    );
  }
      }
```

Код головного екрану зі списком:

```
class MoviesAppHome extends StatefulWidget {
  @override
  MoviesAppHomeState createState() => MoviesAppHomeState();
}

class MoviesAppHomeState extends State<MoviesAppHome> {
  final searchTextController = new TextEditingController();
  String searchText = "";

  @override
  void dispose() {
    searchTextController.dispose();
    super.dispose();
  }

  void itemClick(Movie item) {
    Navigator.push(
      context,
      MaterialPageRoute(
          builder: (context) => MovieDetail(
                movieName: item.title,
                imdbId: item.imdbID,
              )),
    );
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
          title: Center(child: Text('Пошук фільмів')),
        ),
        body: Column(
          children: <Widget>[
            Container(
              child: Row(children: <Widget>[
                Flexible(
                  child: TextField(
                    controller: searchTextController,
                    decoration: InputDecoration(
                      hintText: "Фраза для пошуку",
                      fillColor: Colors.white,
                      border: new OutlineInputBorder(
                        borderRadius: new BorderRadius.circular(25.0),
                        borderSide: new BorderSide(
                        ),
                      ),
                      //fillColor: Colors.green
                    ),
                  ),
                ),
                IconButton(
                  icon: Icon(Icons.search),
                  tooltip: 'Пошук фільмів',
                  color: Colors.lightBlue,
                  onPressed: () {
                    setState(() {
                      searchText = searchTextController.text;
                      SystemChannels.textInput.invokeMethod('TextInput.hide');
                    });
                  },
                ),
              ]),
              padding: EdgeInsets.all(10),
              color: Colors.grey[200],
            ),
            if (searchText.length > 0)
```

```
            FutureBuilder<List<Movie>>(
                future: searchMovies(searchText),
                builder: (context, snapshot) {
                  if (snapshot.hasData) {
                    return Expanded(
                        child: MovieList(
                            movies: snapshot.data,
                            itemClick: this.itemClick)
                    );
                  } else if (snapshot.hasError) {
                    return Text("${snapshot.error}");
                  }
                  return CircularProgressIndicator();
                }),
          ],
        ));
  }
      }
```

Код екрану деталей фільму:

```
class MovieDetail extends StatelessWidget {
  final String movieName;
  final String imdbId;

  MovieDetail({this.movieName, this.imdbId});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(this.movieName),
      ),
      body: FutureBuilder<MovieInfo>(
          future: getMovie(this.imdbId),
          builder: (context, snapshot) {
            if (snapshot.hasData) {
              return Container(
                  padding: EdgeInsets.all(12),
                  child: Column(
                      crossAxisAlignment: CrossAxisAlignment.start,
                      children: <Widget>[
                        Container(
                          padding: EdgeInsets.fromLTRB(0, 10, 0, 10),
                          alignment: Alignment.center,
                          child: Image.network(
                            snapshot.data.poster,
                            width: 200,
                          ),
                        ),
                        Card(
                          child: Container(
                            padding: EdgeInsets.all(12),
                            child: Column(children: [
                                  Text(snapshot.data.plot, textAlign: TextAlign.justify),
                                  SizedBox(height: 15),
                                  PaddedText("Рік : " + snapshot.data.year),
                                  PaddedText("Жанр : " + snapshot.data.genre),
                                  PaddedText("Режесура : " + snapshot.data.director),
                                  PaddedText("Час : " + snapshot.data.runtime),
                                  PaddedText("Рейтинг : " + snapshot.data.rating),
                                  PaddedText("IMDB рейтинг : " + snapshot.data.imdbRating),
                                  PaddedText("Meta Score : " + snapshot.data.metaScore),
                            ],
                            ),
                          ),
                        )
                      ]));
            } else if (snapshot.hasError) {
              return Text("${snapshot.error}");
            }
            return Center(child: CircularProgressIndicator());
          }),
    );
```

```
  }
}

class MovieItem extends StatelessWidget {
  final Movie movie;

  MovieItem({this.movie});

  @override
  Widget build(BuildContext context) {
    return Container(
      child: Row(
        mainAxisAlignment: MainAxisAlignment.start,
        children: <Widget>[
          Column(children: <Widget>[
            if (this.movie.poster != "N/A")
              Image.network(this.movie.poster, height: 100, width: 100)
          ]),
          Expanded(child: Column(
              crossAxisAlignment: CrossAxisAlignment.start,
              mainAxisAlignment: MainAxisAlignment.start,
              children: <Widget>[
                Text(
                  this.movie.title,
                  overflow: TextOverflow.ellipsis,
                  maxLines: 2,
                ),
                Text(this.movie.year),
                Text(this.movie.type)
              ]),)
        ],
      ),
      padding: EdgeInsets.all(10),
      margin: EdgeInsets.only(top: 12, left: 10, right: 10),
      decoration:
          new BoxDecoration(borderRadius: BorderRadius.all(Radius.circular(16)),color:
Colors.white),
    );
  }
           }
```

Код моделей та сервісу для комунікації з сервером:

```
class Movie {
  final String title;
  final String year;
  final String type;
  final String poster;
  final String imdbID;

  Movie({this.title, this.year, this.type, this.poster, this.imdbID});

  factory Movie.fromJson(Map<String, dynamic> json) {
    return Movie(
        title: json['Title'],
        year: json['Year'],
        type: json['Type'],
        poster: json['Poster'],
        imdbID: json['imdbID']);
  }
}

class MovieInfo {
  final String title;
  final String year;
  final String rating;
  final String runtime;
  final String genre;
  final String director;
  final String plot;
  final String poster;
  final String imdbRating;
  final String metaScore;

  MovieInfo(
      {this.title,
      this.year,
      this.rating,
```

```
            this.runtime,
            this.genre,
            this.director,
            this.plot,
            this.poster,
            this.imdbRating,
            this.metaScore});

      factory MovieInfo.fromJSON(Map<String, dynamic> json) {
        return MovieInfo(
            title: json['Title'],
            year: json['Year'],
            rating: json['Rated'],
            runtime: json['Runtime'],
            genre: json['Genre'],
            director: json['Director'],
            plot: json['Plot'],
            poster: json['Poster'],
            imdbRating: json['imdbRating'],
            metaScore: json['Metascore']);
      }
    }


    Future<List<Movie>> searchMovies(keyword) async {
      final response = await http.get('$API_URL$API_KEY&s=$keyword');

      log(response.body);

      if (response.statusCode == 200) {
        Map data = json.decode(response.body);

        if (data['Response'] == "True") {
          var list = (data['Search'] as List)
              .map((item) => new Movie.fromJson(item))
              .toList();
          return list;
        } else {
          throw Exception(data['Error']);
        }
      } else {
        throw Exception('Something went wrong !');
      }
    }

    Future<MovieInfo> getMovie(movieId) async {
      final response = await http.get('$API_URL$API_KEY&i=$movieId');

      log(response.body);

      if (response.statusCode == 200) {
        Map data = json.decode(response.body);

        if (data['Response'] == "True") {
          return MovieInfo.fromJSON(data);
        } else {
          throw Exception(data['Error']);
        }
      } else {
        throw Exception('Something went wrong !');
      }
    }
```

Код плагіну для перевірки швидкодії у Flutter, код для Android:

```
       @Override
      public void onMethodCall(@NonNull MethodCall call, @NonNull Result result) {
        if (call.method.equals("getRefreshRate")) {
          try {
            WindowManager windowManager = (WindowManager)
context.getSystemService(Context.WINDOW_SERVICE);

            float refreshRate = windowManager.getDefaultDisplay().getRefreshRate();
            result.success(refreshRate);
          } catch (Exception e) {
            result.success(null);
          }
```

```
    } else {
      result.notImplemented();
    }
  }
```

Для iOS:

```
- (double)displayRefreshRate:(CADisplayLink *)link {
        NSInteger preferredFPS = link.preferredFramesPerSecond;
        // maximumFramesPerSecond property.
        if (preferredFPS != 0) {
            return @(preferredFPS).doubleValue;
        }
        return @([UIScreen mainScreen].maximumFramesPerSecond).doubleValue;
}
```

Flutter-код для отримання даних з нативних платформ:

```
static Future<double> get getRefreshRate async {
    final double fpsHz = await _channel.invokeMethod('getRefreshRate');
    return fpsHz;
 }
```

Код плагіну:

```
class FpsPlugin {
  static FpsPlugin get instance {
    if (_instance == null) {
      _instance = FpsPlugin.();
    }
    return _instance;
  }

  static Fps _instance;

  static const _maxFrames = 120;
  final lastTime = ListQueue<FrameTiming>(_maxFrames);
  TimingsCallback _timingsCallback;
  List<FpsCallback> _callBackList = [];

  FpsPlugin._() {
    _timingsCallback = (List<FrameTime> timings) {
      _computeFps(timings);
    };
    SchedulerBinding.instance.addTimingsCallback(_timingsCallback);
  }

  registerCallBack(FpsCallback back) {
    _callBackList?.add(back);
  }

  unregisterCallBack(FpsCallback back) {
    _callBackList?.remove(back);
  }

  cancel() {
    if (_timingsCallback == null) {
      return;
    }
    SchedulerBinding.instance.removeTimingsCallback(_timingsCallback);
  }


  double _fpsHz;
  Duration _frameInterval;

  Future<void> _computeFps(List<FrameTiming> lastFrames) async {
    for (FrameTime frames in lastFrames) {
      lastTime.addFirst(frames);
    }
```

```
    while (lastTime.length > _maxFrames) {
      lastTime.removeLast();
    }

    var lastFramesSet = <FrameTime>[];

    if (_fpsHz == null) {
      _fpsHz = await FpsPlugin.getRefreshRate;
    }

    if (_frameInterval == null) {
      _frameInterval =
          Duration(microseconds: Duration.microsecondsPerSecond ~/ _fpsHz);
    }


    var drawFramesCount = lastFramesSet.length;


    // FPS / 60 = drawCount / (drawFramesCount + droppedCount)
    // costCount = (drawFramesCount + droppedCount)
    // FPS ≈  60 * drawFramesCount / costCount
    int droppedCount = 0;
    var costCount = lastFramesSet.map((frame) {
      // 15ms ~/ 16ms = 0
      // 16ms ~/ 16ms = 0
      // 17ms ~/ 16ms = 1
      int droppedCount =
          (frame.totalSpan.inMicroseconds ~/ _frameInterval.inMicroseconds);
      return droppedCount +
          1;
    }).fold(0, /(a, b) => a + b);


    droppedCount = costCount - drawFramesCount;
    double fps = drawFramesCount * _fpsHz / costCount;
    lastFrames.clear();
    _callBackList?.forEach((callBack) {
      callBack(fps, droppedCount.toDouble());
    });
  }
}
```

ДОДАТОК Б

ВІДГУК

керівника економічного розділу

на кваліфікаційну роботу магістра

на тему:

«Розробка програмного забезпечення для дослідження ефективності засобу Flutter при розробці мобільних додатків»

студента групи 121м-19-1 Ситника Романа Сергійовича

Керівник економічного розділу
доцент каф. ПЕП та ПУ, к.е.н. **Л. В. Касьяненко**

ДОДАТОК В

ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ

Ім’я файла	**Опис**
Пояснювальні документи	
Робота_Ситник.doc	Пояснювальна записка до магістерської роботи. Документ Word.
Робота_Ситник.pdf	Пояснювальна записка до до магістерської роботи в форматі PDF
Програма	
Program.zip	Архів. Містить коди прикладів програм.
Презентація	
Презентація_Ситник.ppt	Презентація до магістерської роботи