UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato finale in Ingegneria del software

Confronto tra sviluppo nativo e sviluppo cross-platform (MAUI e Flutter)

Anno Accademico 2023/2024

Candidato:

Zurlo Simone

matr. N46003818

A tutti quelli che mi hanno sopportato.

Indice

Introduzione

Nel vasto panorama delle applicazioni mobili, è essenziale garantire che il proprio prodotto sia accessibile a un'ampia base di utenti, senza escludere potenziali acquirenti. Questo implica la necessità di sviluppare il software per una varietà di sistemi operativi che alimentano una gamma diversificata di hardware. Nei decenni recenti, due sistemi operativi hanno dominato il mercato delle applicazioni mobili: iOS, il sistema operativo dei dispositivi mobili sviluppati da Apple, e Android, creato da Google e basato sul kernel Linux.

Questi due sistemi operativi sono radicalmente diversi tra loro e richiedono competenze e tecnologie specifiche per la costruzione di un'applicazione. Un’applicazione nativa è progettata esclusivamente per un singolo sistema operativo, utilizzando gli strumenti e i linguaggi di programmazione specifici di quel sistema.

D'altra parte, i framework cross-platform rappresentano un'alternativa all'approccio nativo. Questi strumenti operano a un livello di astrazione superiore rispetto ai framework nativi, consentendo lo sviluppo di un'unica base di codice che può essere compilata e distribuita su più sistemi operativi. In pratica, ciò significa che gli sviluppatori possono scrivere e mantenere un'unica base di codice per creare applicazioni compatibili sia con iOS che con Android, riducendo così il tempo e le risorse necessarie per sviluppare e mantenere più versioni di un'applicazione per sistemi operativi diversi.

Contestualizzazione dello sviluppo mobile

Nel contesto dello sviluppo nativo, emergono notevoli vantaggi che ne fanno una scelta attraente per molti sviluppatori e aziende. Questo approccio è caratterizzato da una profonda integrazione con il sistema operativo sottostante ed offre prestazioni ottimizzate sfruttando appieno le risorse hardware disponibili. Inoltre gli sviluppatori nativi possono beneficiare di un accesso tempestivo alle ultime funzionalità introdotte dal sistema operativo, consentendo loro di integrare queste novità nelle proprie applicazioni in anticipo rispetto ai concorrenti che utilizzano altre metodologie di sviluppo. Un altro vantaggio significativo dello sviluppo nativo è la coerenza del design che garantisce un'esperienza utente in linea con le aspettative dell'utente finale. Tuttavia, nonostante questi notevoli vantaggi, il processo di sviluppo nativo può essere complesso e richiede risorse considerevoli, sia in termini di tempo che di denaro. Spesso le aziende si trovano di fronte alla sfida di dover sviluppare e mantenere la stessa applicazione su entrambe le piattaforme principali, iOS e Android, con risorse limitate a disposizione. È in questo contesto che i framework cross-platform emergono come una soluzione attraente. (Fojtík & Pawlas, 2023)

I framework cross-platform offrono un'alternativa efficace per lo sviluppo di applicazioni compatibili con più sistemi operativi. Questi strumenti consentono agli sviluppatori di scrivere un'unica base di codice che può essere compilata e distribuita separatamente su diversi dispositivi e sistemi operativi. Questo approccio consente di risparmiare tempo e denaro, riducendo la necessità di sviluppare e mantenere codice separato per ogni piattaforma. Tuttavia, l'utilizzo dei framework cross-platform comporta anche alcuni compromessi. L'aggiunta di un livello di astrazione superiore può portare a un calo delle prestazioni e a un accesso limitato alle funzionalità specifiche del dispositivo. Pertanto, la scelta tra sviluppo nativo e cross-platform dipende da diversi fattori, tra cui le esigenze specifiche dell'applicazione, le risorse disponibili e le preferenze dello sviluppatore. Se l'obiettivo è creare un'applicazione altamente performante con accesso completo alle risorse del dispositivo, lo sviluppo nativo potrebbe essere la scelta migliore. D'altra parte, se è necessario creare un'applicazione eseguibile su più sistemi operativi con risorse limitate, i

framework cross-platform rappresentano un'opzione valida. (Systä & Sand, 2023)

Panoramica sullo sviluppo nativo e cross-platform

Lo sviluppo nativo rappresenta un approccio fondamentale nello sviluppo di applicazioni mobile, poiché si basa sull'utilizzo delle specifiche e delle API fornite dai sistemi operativi nativi, come iOS e Android. Questo approccio consente agli sviluppatori di sfruttare appieno le funzionalità del dispositivo e di offrire un'esperienza utente ottimizzata. Tuttavia, lo sviluppo nativo può risultare vincolante in termini di portabilità delle applicazioni su diverse piattaforme, richiedendo la scrittura di codice specifico per ciascun sistema operativo. Ciò comporta un aumento dei costi e dei tempi di sviluppo, oltre a una maggiore complessità nella gestione del codice. (Mannino et Al., 2023)

Dall'altro lato, lo sviluppo cross-platform si propone di superare le limitazioni della programmazione nativa consentendo agli sviluppatori di creare applicazioni utilizzando un singolo codice base per più piattaforme. Questo approccio offre vantaggi significativi in termini di efficienza e riduzione dei costi, poiché consente di raggiungere un vasto pubblico target con un unico sforzo di sviluppo. Tuttavia, il cross-platform presenta anche sfide specifiche, come la necessità di garantire la compatibilità tra le diverse piattaforme e la gestione delle differenze nell'esperienza utente e nelle prestazioni. (Hvenfelt, 2023; Alvarez et Al., 2023)

Scopo e struttura della tesi

In questa tesi, esploreremo in dettaglio sia lo sviluppo nativo che il cross-platform, analizzando i loro rispettivi vantaggi e svantaggi. In particolare, ci concentreremo sull'analisi delle caratteristiche principali e delle sfide associate allo sviluppo cross-platform, con l'obiettivo di fornire una panoramica completa delle diverse metodologie disponibili per lo sviluppo di applicazioni mobile.

Eseguiremo quindi un'analisi approfondita e comparativa tra lo sviluppo nativo e lo sviluppo cross-platform, con particolare attenzione ai framework .NET MAUI e Flutter di Google.

L'obiettivo principale è quello di valutare le prestazioni, la portabilità e le caratteristiche di sviluppo di entrambi gli approcci, al fine di fornire agli sviluppatori un quadro chiaro e informato per la selezione della tecnologia più adatta alle proprie esigenze specifiche.

La struttura della tesi prevede una prima sezione dedicata all'introduzione e alla contestualizzazione dello sviluppo mobile, seguita da una descrizione dettagliata dei principi dello sviluppo nativo e cross-platform. Successivamente, verrà approfondito il confronto tra .NET MAUI e Flutter, con una analisi esaustiva delle loro caratteristiche, punti di forza e debolezze.

Un elemento chiave della tesi è la realizzazione di un test sperimentale, finalizzato a confrontare le performance delle applicazioni sviluppate utilizzando i tre approcci selezionati: nativo, .NET MAUI e Flutter. Questo test consiste nell'implementazione di tre applicazioni di esempio, ciascuna delle quali esegue un carico di lavoro generico simulato. Le applicazioni verranno quindi valutate su un dispositivo Android reale considerando le caratteristiche di performance secondo gli standard definiti dalla ISO 25010.

Attraverso questo test sperimentale, ci proponiamo di fornire dati empirici e confronti diretti tra le diverse modalità di sviluppo, al fine di supportare le conclusioni e le raccomandazioni presentate nella tesi. In questo modo, gli sviluppatori potranno prendere decisioni informate sulla scelta della tecnologia più adatta alle proprie esigenze specifiche, tenendo conto sia delle prestazioni che dei vincoli di sviluppo e di deployment.

Capitolo 1: Sviluppo Nativo - Principi e Caratteristiche

Lo sviluppo nativo rappresenta un approccio essenziale per la creazione di software destinati a piattaforme specifiche come Android e iOS/iPadOS, le due principali piattaforme mobile attualmente in circolazione. Con l'esplosione dell'uso degli smartphone negli ultimi anni, la scelta del metodo di sviluppo giusto è diventata cruciale per garantire un'esperienza utente ottimale su questi dispositivi.

Android, sviluppato da Google e basato su un modello open-source, è il sistema operativo dominante nel mercato degli smartphone, mentre iOS/iPadOS, sviluppato da Apple e basato su un modello closed-source, equipaggia i dispositivi della società di Cupertino, come iPhone e iPad.

Quando si tratta di sviluppare applicazioni per queste piattaforme, la scelta naturale è quella di adottare lo sviluppo nativo. Questo approccio implica la creazione di applicazioni specificamente ottimizzate per la piattaforma di destinazione, utilizzando linguaggi di programmazione e strumenti di sviluppo nativi propri di quella piattaforma. Ad esempio, per lo sviluppo di applicazioni iOS/iPadOS si utilizza principalmente il linguaggio di programmazione Swift, insieme alle API e agli strumenti di sviluppo forniti da Apple tramite Xcode.

1.1 Definizione e principi fondamentali dello sviluppo nativo

Lo sviluppo nativo rappresenta un approccio fondamentale e consolidato nel campo dello sviluppo software per la creazione di applicazioni specifiche per una determinata piattaforma, come iOS o Android. Questo approccio si basa sull'utilizzo di linguaggi di programmazione e strumenti di sviluppo nativi forniti dalle case madri delle rispettive

piattaforme, consentendo agli sviluppatori di sfruttare appieno le caratteristiche e le funzionalità del sistema operativo di destinazione.

Il principio fondamentale dello sviluppo nativo prevede l'utilizzo di linguaggi di programmazione specifici della piattaforma. Per lo sviluppo di applicazioni native per iOS, vengono utilizzati principalmente i linguaggi Swift e Objective-C, mentre per Android si utilizzano Java o Kotlin. Questi linguaggi sono ottimizzati per le rispettive piattaforme e consentono agli sviluppatori di scrivere codice altamente performante e efficiente. (Darji et Al., 2021)

Le applicazioni native sono progettate per un particolare sistema operativo (OS), come ad esempio iOS o Android, il che significa che non vi è alcuna portabilità tra piattaforme. Comunemente, queste app sono disponibili nei negozi digitali (ad esempio App Store per iOS e Google Play Store per Android).

Le app native tendono a seguire l'interfaccia (anche detto look and feel) e i pattern tecnici supportati dalla piattaforma specifica. Questo tipo di app ha accesso diretto alle risorse del dispositivo come sensori, fotocamere, GPS (Global Positioning System). Questo approccio offre sia elevate prestazioni che un'interfaccia utente (UI/UX) specializzata per gli utenti. Tuttavia, richiede un framework e un SDK specifico. In sintesi, le applicazioni native sono ottimizzate per una piattaforma specifica, offrendo vantaggi come prestazioni elevate e un'esperienza utente personalizzata, ma richiedono l'uso di strumenti e tecnologie dedicate a quella piattaforma. (Zupancic, 2021)

1.2. Esempi di sviluppo nativo su piattaforme iOS e Android

Esempi di sviluppo nativo su piattaforme iOS e Android illustrano l'utilizzo di linguaggi specifici e strumenti di sviluppo dedicati per la creazione di applicazioni ottimizzate per ciascuna piattaforma.

Nel caso di dispositivi Android, è comune utilizzare il linguaggio Java o Kotlin insieme al framework di sviluppo Android SDK e, preferibilmente, l'IDE Android Studio. Questo ambiente fornisce agli sviluppatori tutte le risorse necessarie per creare e testare applicazioni

Android in modo efficiente e completo. Inoltre, gli sviluppatori sono tenuti a gestire la compatibilità con le diverse versioni della piattaforma, specificando parametri come Target SDK Version e Min SDK Version.

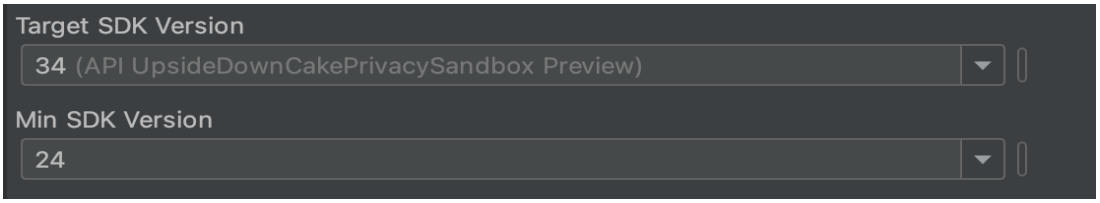

Figura 1 - Parametri che determinano la versione di Android utilizzata come riferimento per l'esecuzione dell'applicazione (Target SDK Version) e la versione minima richiesta per il corretto funzionamento dell'applicazione su dispositivi più datati (Min SDK Version)

È importante selezionare attentamente questi parametri per garantire la massima compatibilità con il maggior numero possibile di dispositivi, mantenendo al contempo le funzionalità e le prestazioni desiderate.

Per quanto riguarda dispositivi iOS/iPadOS, l'utilizzo del linguaggio Swift è diventato lo standard, insieme al framework UIKit per la creazione di interfacce grafiche. L'IDE Xcode è lo strumento principale utilizzato dagli sviluppatori iOS/iPadOS per la creazione, il debug e il testing delle applicazioni. Tuttavia, è importante notare che Xcode è disponibile solo su macOS, il che significa che gli sviluppatori devono necessariamente utilizzare un computer Apple per sviluppare applicazioni iOS/iPadOS.

Anche se lo sviluppo nativo offre prestazioni elevate, un accesso completo alle funzionalità del dispositivo ed una ricca documentazione fornita dalle case madri, presenta anche alcune limitazioni. Una delle principali limitazioni è la necessità di creare e mantenere due codebase separate per ciascuna piattaforma, il che richiede risorse e tempo aggiuntivi.

Questo rende lo sviluppo nativo meno pratico per alcuni progetti, spingendo gli sviluppatori a considerare alternative come lo sviluppo cross-platform, che consente di utilizzare un'unica codebase per supportare più piattaforme. Tuttavia, questa soluzione può comportare compromessi in termini di caratteristiche e prestazioni. (Aurelius, 2020)

1.3. Vantaggi e limitazioni dello sviluppo nativo

Lo sviluppo nativo offre numerosi vantaggi, tra cui prestazioni ottimizzate, accesso completo alle funzionalità del dispositivo e integrazione nativa con l'ecosistema della piattaforma. Questo approccio consente agli sviluppatori di sfruttare appieno le caratteristiche specifiche della piattaforma di destinazione, garantendo un'esperienza utente fluida e coerente. (Biørn-Hansen, 2020)

I **vantaggi** dello sviluppo nativo sono numerosi e significativi, tra cui:

- Prestazioni: Uno dei vantaggi più evidenti delle applicazioni native è rappresentato dalle loro prestazioni ottimizzate. Poiché sono sviluppate specificamente per una piattaforma particolare utilizzando linguaggi di programmazione nativi come Swift o Objective-C per iOS e Java o Kotlin per Android, le app native possono sfruttare appieno la potenza e le capacità della piattaforma. Questo si traduce in applicazioni più veloci, efficienti e reattive, in grado di offrire un'esperienza utente fluida e senza interruzioni.
- Accesso alle funzionalità platform-specific: Un altro vantaggio importante è la possibilità di accedere a tutte le funzionalità e API specifiche della piattaforma. Questo consente agli sviluppatori di sfruttare al meglio gli ultimi aggiornamenti e le funzionalità offerte dal sistema operativo sottostante. Ad esempio, le applicazioni native possono integrare facilmente funzionalità come la voce di Siri su iOS o i servizi Google su Android, garantendo un'integrazione perfetta con il sistema operativo e un'esperienza utente più ricca e coinvolgente.
- Migliore User eXperience: Grazie alle prestazioni ottimizzate e all'accesso alle funzionalità specifiche del dispositivo, le applicazioni native offrono un'esperienza utente superiore. Possono essere progettate e ottimizzate per adattarsi perfettamente alla piattaforma e integrarsi in modo trasparente con l'interfaccia utente del sistema operativo. Ciò si traduce in app più fluide, intuitive e piacevoli da utilizzare, che possono contribuire a migliorare l'engagement degli utenti e la soddisfazione complessiva dell'utente.

- Maggiore sicurezza: Le applicazioni native possono integrarsi più efficacemente con i meccanismi di sicurezza della piattaforma, offrendo una maggiore protezione e crittografia dei dati. Gli sviluppatori possono sfruttare gli strumenti e le tecnologie di sicurezza forniti dalla piattaforma per garantire la sicurezza dei dati sensibili degli utenti. Inoltre, la possibilità di ricevere aggiornamenti specifici della piattaforma consente di affrontare rapidamente e efficacemente eventuali problemi di sicurezza che possono emergere.
- Maturità della community: La community che sviluppa applicazioni native per iOS e Android è estremamente vasta e attiva. Sono disponibili una vasta gamma di risorse, documentazione ufficiale e supporto da parte degli sviluppatori tramite forum, blog, articoli tecnici e newsletter. Questo fornisce agli sviluppatori un prezioso supporto e una fonte di informazioni e conoscenze, facilitando il processo di sviluppo e aiutandoli a risolvere eventuali problemi o sfide che possono incontrare lungo il percorso.

Tuttavia, lo sviluppo nativo può comportare anche alcune sfide, come la necessità di scrivere e mantenere codice separato per ciascuna piattaforma e la curva di apprendimento associata agli strumenti di sviluppo nativi.

Gli **svantaggi** dello sviluppo nativo possono essere meglio delineati di seguito:

- Duplicazione del lavoro di sviluppo: Un altro svantaggio significativo dello sviluppo nativo è la necessità di creare e mantenere due codebase separate per ciascuna piattaforma. Questo richiede sforzi e risorse aggiuntive da parte degli sviluppatori, in quanto devono scrivere e mantenere codice specifico per ciascuna piattaforma. Questo può aumentare i costi e i tempi di sviluppo complessivi del progetto e può essere inefficace per progetti con risorse limitate o tempi di sviluppo stretti.
- Complessità della gestione: La gestione di due codebase separate può portare a complicazioni aggiuntive nel processo di sviluppo e manutenzione dell'applicazione. Gli sviluppatori devono tenere traccia delle differenze tra le due codebase e assicurarsi che le modifiche apportate a una piattaforma non influenzino negativamente l'altra.

Questo può aumentare il rischio di errori e comportare una maggiore complessità nella gestione del progetto.

In sintesi, sebbene lo sviluppo nativo offra prestazioni elevate, un completo accesso alle funzionalità del dispositivo e una ricca documentazione fornita dalle case madri, gli svantaggi come la duplicazione del lavoro e la complessità della gestione possono rendere questo approccio meno pratico per alcuni progetti. Pertanto, è importante valutare attentamente le esigenze specifiche del progetto e considerare alternative come lo sviluppo cross-platform, che possono offrire un compromesso tra prestazioni e costi.

Lo sviluppo nativo rimane una scelta popolare e affidabile per la creazione di applicazioni mobile di alta qualità, offrendo un equilibrio tra prestazioni ottimali e accesso completo alle funzionalità del dispositivo.

Capitolo 2: Concetto di sviluppo cross-platform e vantaggi

Con il crescere della complessità delle applicazioni e la diversificazione delle piattaforme, è emerso un nuovo approccio nello sviluppo software: lo sviluppo cross-platform.

I framework cross-platform rappresentano una soluzione efficace per lo sviluppo di applicazioni compatibili con più sistemi operativi. Questi strumenti consentono agli sviluppatori di scrivere un'unica base di codice che può essere compilata e distribuita separatamente su diverse piattaforme e dispositivi. In questo modo, gli sviluppatori possono risparmiare tempo e risorse, evitando inoltre la necessità di sviluppare e mantenere codice separato per ciascuna piattaforma.

Questo approccio offre numerosi vantaggi, tra cui una maggiore efficienza nello sviluppo, una maggiore flessibilità e la possibilità di raggiungere un pubblico più ampio con un singolo sforzo di sviluppo.

Tuttavia, è importante considerare che lo sviluppo cross-platform potrebbe comportare compromessi in termini di prestazioni o caratteristiche, poiché le applicazioni devono essere progettate per funzionare su una varietà di piattaforme e dispositivi con diversi requisiti e limitazioni.

In questo capitolo esploreremo in dettaglio i principali framework cross-platform, tra cui .NET MAUI e Flutter, analizzando le loro caratteristiche, vantaggi e limitazioni. Inoltre, discuteremo delle situazioni in cui lo sviluppo cross-platform potrebbe essere preferibile rispetto allo sviluppo nativo e forniremo esempi pratici di applicazioni sviluppate utilizzando questi framework.

2.1. Approcci dello sviluppo cross-platform

Nel passato, i ricercatori hanno compiuto ampi sforzi per categorizzare gli approcci cross-platform in diverse categorie, guidati dalla proliferazione di framework associati a ciascuna categoria. Queste categorizzazioni non sono state consistenti in tutto il settore, con diverse classificazioni dei framework da parte di autori diversi che contribuiscono alla complessità nella comprensione del panorama dello sviluppo di applicazioni mobili cross-platform. Ad esempio, lo studio di Choi Yang & Jeong nel 2009 si è concentrato su un "Application Framework" che definiva regole per la trasformazione dell'interfaccia utente specifiche per ogni piattaforma, mettendo in luce l'importanza di adattare le interfacce ai diversi ambienti dei sistemi operativi mobili. Ricerche successive di Allen et al. e Holzinger Treitler & Slany hanno approfondito analisi comparative di framework popolari come Rhodes, RhoSync, Titanium Mobile e PhoneGap, esplorando fattori come principi di design dell'interfaccia utente, metriche di prestazioni e compatibilità tra piattaforme. Inoltre, il lavoro di autori come Raj & Tolety nel 2012 ha ampliato questo ambito categorizzando gli approcci cross-platform in gruppi distinti come Web, Hybrid, Interpreted e Cross-compiled, fornendo agli sviluppatori un framework strutturato per valutare i compromessi e i vantaggi di ciascun approccio.

Nonostante questo progresso, l'evoluzione rapida degli strumenti e delle tecnologie per lo sviluppo di applicazioni mobili richiede ricerca continua per tenere il passo con gli ultimi sviluppi e garantire che gli sviluppatori abbiano accesso a informazioni aggiornate e complete per selezionare il framework cross-platform più adatto per i loro progetti.

Nonostante siano state proposte diverse classificazioni degli approcci di sviluppo cross-platform da parte di vari autori, alcune categorie emergono come comuni e universalmente riconosciute. Tra queste, quattro sono le categorie che spiccano maggiormente in tutte le classificazioni: Web, Hybrid, Interpreted e Cross-compiled.

La categoria Web si riferisce agli approcci che sfruttano le tecnologie web standard come HTML, CSS e JavaScript per creare applicazioni che possono essere eseguite su diverse piattaforme attraverso un browser web.

Le applicazioni ibride combinano elementi di sviluppo web e nativo, consentendo agli sviluppatori di scrivere un'unica base di codice utilizzando tecnologie web standard e quindi "imballarla" in un contenitore nativo per distribuirla su più piattaforme.

Gli approcci interpretati coinvolgono l'utilizzo di un interprete che esegue il codice sorgente su più piattaforme senza la necessità di compilazione. Questo consente agli sviluppatori di scrivere un'unica base di codice che può essere eseguita su diverse piattaforme senza la necessità di adattamenti specifici.

Infine, gli approcci cross-compiled coinvolgono la traduzione del codice sorgente in un linguaggio di basso livello o intermedio che può essere eseguito su diverse piattaforme. Questo metodo consente agli sviluppatori di scrivere il codice una volta e distribuirlo su più piattaforme, garantendo prestazioni ottimizzate per ciascuna piattaforma di destinazione.

2.2. NET Multi-platform App UI (MAUI)

Il rilascio di .NET 6 (supporto a lungo termine), nel novembre 2021, ha reso Microsoft un vero concorrente nell'industria cross-platform. .NET Multi-platform App UI (MAUI) è ora incluso nel pacchetto, e qualsiasi sviluppatore può ora iniziare a scrivere app native, web o ibride multi-piattaforma immediatamente. Si può inoltre considerare che MAUI è stato sviluppato come un'evoluzione di Xamarim.Forms, che aveva già sei anni.

Questo framework rivoluziona lo sviluppo di applicazioni mobili cross-platform, consentendo agli sviluppatori di creare applicazioni native per Android, iOS e Windows utilizzando una singola base di codice. Le radici di .NET MAUI possono essere rintracciate nell'evoluzione delle piattaforme di sviluppo di Microsoft, a partire dall'introduzione di Xamarin.Forms, seguita da Xamarin, e infine il passaggio a .NET MAUI. Xamarin.Forms ha fornito un modo per sviluppare app mobili cross-platform utilizzando una base di codice condivisa, ma esistevano limitazioni in termini di funzionalità specifiche della piattaforma e prestazioni. (Ye, 2023)

Con il rilascio di .NET MAUI, Microsoft ha affrontato queste limitazioni offrendo un framework più potente e versatile che sfrutta gli ultimi progressi nelle tecnologie .NET e

Xamarin.

Rispetto a Xamarin.Forms, .NET MAUI incorpora miglioramenti come una struttura del progetto più flessibile, componenti UI cross-platform migliorate, capacità di ricarica rapida migliorate e supporto per nuove piattaforme rispetto a quelle già esistenti in precedenza.

2.2.1 Architettura di .NET MAUI

L'architettura a strati di .NET MAUI è progettata per fornire un framework modulare e flessibile per la costruzione efficiente di applicazioni cross-platform. (Ponomarev, 2023)

L'architettura è composta da diversi strati, ognuno dei quali serve uno scopo specifico nel processo di sviluppo dell'applicazione. Al suo nucleo, .NET MAUI segue un approccio a strati che separa le responsabilità e favorisce la riutilizzabilità e la manutenibilità del codice. Partendo dal basso, lo strato fondamentale include le implementazioni specifiche della piattaforma per Android, iOS e Windows, consentendo agli sviluppatori di accedere alle API e alle funzionalità native in modo trasparente.

Sopra questo strato si trova lo strato cross-platform, dove il codice condiviso e i componenti UI sono definiti per astrarre le differenze specifiche della piattaforma e facilitare la condivisione del codice tra diversi sistemi operativi. Lo strato di presentazione comprende gli elementi dell'interfaccia utente, i layout e lo stile, utilizzando XAML per definizioni di UI dichiarative e garantendo un aspetto coerente tra le piattaforme. Lo strato della logica dell'applicazione contiene la logica di business, i modelli di dati e i servizi necessari per guidare la funzionalità dell'applicazione, promuovendo la separazione delle responsabilità e la testabilità.

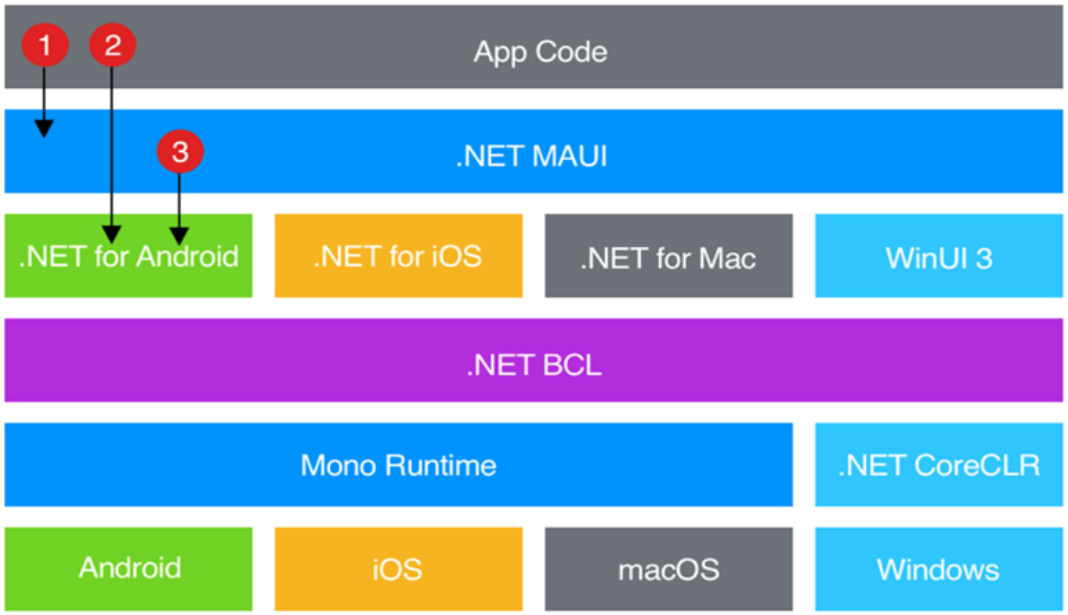

Figura 2 - L'applicazione del pattern architetturale a strati nel framework MAUI

Inoltre, lo strato dei servizi dell'applicazione include funzionalità come l'iniezione delle dipendenze, i servizi di navigazione e le estensioni specifiche della piattaforma, consentendo l'estensibilità e la personalizzazione in base ai requisiti della piattaforma.

Strutturando il framework in questi distinti strati, .NET MAUI fornisce agli sviluppatori un'architettura coesa e organizzata che semplifica lo sviluppo, promuove l'efficienza del codice e semplifica la manutenzione delle applicazioni cross-platform.

In .NET MAUI, la modalità di sviluppo nota come WORA (Write Once, Run Anywhere) incarna il principio fondamentale del framework della riutilizzabilità del codice e della compatibilità cross-platform.

Con questo approccio, gli sviluppatori possono scrivere una sola base di codice per le proprie applicazioni utilizzando .NET MAUI, che può quindi essere distribuita ed eseguita senza soluzione di continuità su piattaforme multiple, incluse Android, iOS e Windows. Questa modalità di sviluppo consente agli sviluppatori di concentrarsi sulla creazione di un'esperienza utente e un set di funzionalità unificato, senza la necessità di duplicare gli sforzi per ciascuna piattaforma di destinazione. Sfruttando il potere delle astrazioni e dei

componenti condivisi di .NET MAUI, gli sviluppatori possono costruire efficacemente applicazioni che si adattano alle linee guida di design e alle funzionalità distintive di ogni piattaforma, massimizzando al contempo il riutilizzo del codice.

La modalità di sviluppo WORA in .NET MAUI non solo accelera il processo di sviluppo, ma semplifica anche la manutenzione e gli aggiornamenti, garantendo prestazioni e esperienza utente coerenti su tutte le piattaforme supportate. (Palmqvist, 2023)

Nel contesto di .NET MAUI, dove l'obiettivo è raggiungere la compatibilità cross-platform e la riutilizzabilità del codice, possono verificarsi scenari in cui una specifica funzionalità unica a una particolare piattaforma non può essere implementata direttamente utilizzando una singola base di codice. Questa limitazione può derivare dai diversi approcci di implementazione e funzionalità proprietarie offerte dalla casa madre per ciascuna piattaforma.

Per affrontare questa sfida senza ricorrere alla gestione di codebase separate per ogni piattaforma, gli sviluppatori possono sfruttare le funzionalità di compilazione condizionale del codice presenti in .NET MAUI. Utilizzando direttive di compilazione condizionale, gli sviluppatori possono efficacemente segregare segmenti di codice specifici della piattaforma all'interno della base di codice condivisa, consentendo loro di implementare la funzionalità in modo personalizzato tra le piattaforme di destinazione. Questo approccio consente agli sviluppatori di incorporare funzionalità specifiche della piattaforma all'interno della base di codice unificata di .NET MAUI, preservando i vantaggi della riutilizzabilità del codice e al contempo adattandosi alle differenze e ai requisiti intrinseci delle singole piattaforme.

2.2.2 Esempio di implementazione in C# con .NET MAUI

Pensiamo, ad esempio, al caso in cui si desidera connettersi ad una rete Wi-Fi specifica su piattaforme iOS e Android.

In questo caso, iOS e Android possono avere meccanismi e requisiti diversi per la gestione delle connessioni Wi-Fi, rappresentando una sfida per un approccio unificato cross-platform.

Per affrontare questo problema, gli sviluppatori possono utilizzare la compilazione

condizionale del codice in .NET MAUI per adattare l'implementazione della connessione Wi-Fi per ciascuna piattaforma.

Questo approccio consente agli sviluppatori di implementare le funzionalità di connettività Wi-Fi specifiche della piattaforma nella base di codice unificata di .NET MAUI mantenendo al contempo la riutilizzabilità del codice e riducendo al minimo la duplicazione.

Segue un esempio di codice implementato in C# con .NET MAUI:

```
#if ANDROID
    if(OperatingSystem.IsAndroidVersionAtLeast(29)){
        // If android >= 10 i need to the new api to connect to the AP
        WifiNetworkSpecifier.Builder specifierBuilder = new WifiNetworkSpecifier.Builder();
        specifierBuilder.SetSsid("My Network");
        specifierBuilder.SetWpa2Passphrase("mypsw123");
        WifiNetworkSpecifier = specifierBuilder.Build();
        NetworkRequest.Builder networkRequestBuilder = new NetworkRequest.Builder();
        networkRequestBuilder.AddTransportType(TransportType.Wifi);
        networkRequestBuilder.SetNetworkSpecifier(wifiNetworkSpecifier);
        NetworkRequest = networkRequestBuilder.Build();
        ConnectivityManager cm = (ConnectivityManager)Android.App.Application.Context.GetSystemService(Context.ConnectivityService);
        if (cm != null)
        {
            cm.RequestNetwork(networkRequest, new NetworkCallbackCustom());
        }
    }else{
        //If android < 10 i need to use the old API to connect to the AP
        var ssid = "\"My Network\"";
        var password = "\"mypsw123\"";
        var wifiConfig = new WifiConfiguration();
        wifiConfig.Ssid = ssid;
        wifiConfig.PreSharedKey = password;
        wifiConfig.AllowedKeyManagement.Set((int)KeyManagementType.WpaPsk);
        var wifiManager = (WifiManager)Android.App.Application.Context.GetSystemService(Context.WifiService);
        if (!wifiManager.IsWifiEnabled){
            wifiManager.SetWifiEnabled(true);
            Thread.Sleep(5000);
        }
        //Testing the network
        int netId = wifiManager.AddNetwork(wifiConfig);
        wifiManager.RemoveNetwork(netId);
        //Disabling the currently configured network
        var netId1=wifiManager.ConnectionInfo.NetworkId;
        wifiManager.DisableNetwork(netId1);
        //Adding the network
```

```
        int finalNetId=wifiManager.AddNetwork(wifiConfig);
        //Connecting
        wifiManager.Disconnect();
        wifiManager.EnableNetwork(finalNetId, true);
        wifiManager.Reconnect();
        CancellationTokenSource = new CancellationTokenSource();
        Toast.Make("Connecting to the network", ToastDuration.Long, 16).Show(cancellationTokenSource.Token);
    }
#elif IOS
    //iOS code
    var wifiManager = new NEHotspotConfigurationManager();
    var wifiConfig = new NEHotspotConfiguration("My Network","mypsw123", false) { JoinOnce = true }; //Eseguo una connessione alla
rete e mi assicuro che il dispositivo si disconnetta quando va in sleep o l'app si chiude (JoinOnce)
    wifiManager.ApplyConfiguration(wifiConfig, (error) =>
    {
        if (error != null)
        {
            Console.WriteLine(error.Description);
        }
    });
#endif
```

Dall'esempio riportato si nota l'uso della compilazione condizionale mediante l'istruzione "#if" per delimitare l'inizio di una "zona Android" mentre successivamente con l'istruzione "#elif" viene realizzato il codice per iOS.

2.2.3 Peculiarità di .NET MAUI

In .NET MAUI, dove l'interfaccia utente e la grafica dell'applicazione sono definite utilizzando XAML (eXtensible Application Markup Language), la natura multi-piattaforma del framework introduce la sfida di garantire una rappresentazione visiva coerente su diverse piattaforme di destinazione. Senza l'uso strategico di compilazioni condizionali, gli elementi grafici specificati in XAML vengono resi come previsto su ciascuna piattaforma, riflettendo il design e il layout definiti nella base di codice condivisa. (MAUI Docs, 2023)

Per impostazione predefinita, .NET MAUI sfrutta la natura agnostica della piattaforma di XAML per consentire agli sviluppatori di creare un design dell'interfaccia utente unificato che può adattarsi alle caratteristiche e ai requisiti unici di Android, iOS e Windows. Tuttavia, in scenari in cui sono necessarie variazioni specifiche della piattaforma nella

grafica o negli elementi visivi per conformarsi alle linee guida della piattaforma o sfruttare funzionalità proprietarie, le compilazioni condizionali diventano un'attrezzatura essenziale. L'uso di compilazioni condizionali in .NET MAUI permette agli sviluppatori di regolare finemente la presentazione grafica delle loro applicazioni su diverse piattaforme, trovando un equilibrio tra coerenza della piattaforma e personalizzazione per fornire interfacce utente coinvolgenti e specifiche della piattaforma.

Nell'ambito di .NET MAUI, il processo di creazione dell'interfaccia grafica si distingue per l'utilizzo di controlli UI nativi specifici per ogni piattaforma di destinazione. Questo approccio garantisce che la grafica dell'applicazione assomigli strettamente a quella delle applicazioni native su dispositivi Android, iOS e Windows, migliorando l'esperienza complessiva dell'utente fornendo un aspetto e una sensazione familiari e coerenti. Sfruttando i controlli UI nativi della piattaforma di destinazione, .NET MAUI consente agli sviluppatori di progettare interfacce che si integrano perfettamente con il linguaggio visivo e i modelli di interazione dell'utente di ogni sistema operativo, promuovendo un'estetica di design coesa e appropriata alla piattaforma.

Inoltre, .NET MAUI si distingue per il suo supporto alla tecnologia "Hot Reload", una funzionalità che ottimizza il flusso di lavoro di sviluppo consentendo agli sviluppatori di visualizzare e testare modifiche grafiche in tempo reale senza richiedere una ricompilazione completa del codice sorgente. Questa capacità aumenta significativamente la produttività e l'efficienza degli sviluppatori, poiché consente iterazioni rapide sul design grafico dell'applicazione, facilitando la prototipazione rapida, sperimentazione e perfezionamento dell'interfaccia utente.

La combinazione dell'utilizzo di controlli UI nativi e dell'incorporazione della tecnologia Hot Reload in .NET MAUI sottolinea l'impegno del framework nel fornire interfacce utente visivamente accattivanti e native della piattaforma, migliorando l'esperienza di sviluppo per gli sviluppatori.

Nel campo di .NET MAUI, il processo di sviluppo offre un alto grado di flessibilità consentendo alle applicazioni di essere create su vari sistemi operativi come Windows,

Linux o macOS. Una volta completato lo sviluppo dell'applicazione, il progetto .NET MAUI può essere compilato in app native personalizzate per le piattaforme di destinazione. Tuttavia, il processo di compilazione varia a seconda della piattaforma. Per le app Android, la compilazione comporta la traduzione del codice C# in Linguaggio Intermedio (IL), che viene successivamente compilato in codice assembly nativo durante l'avvio dell'app tramite la compilazione Just-In-Time (JIT). Questo approccio garantisce un'esecuzione efficiente e un'ottimizzazione sui dispositivi Android. Viceversa le app iOS sono pre-compilate interamente in codice assembly ARM. Questo processo di pre-compilazione semplifica l'esecuzione del codice sui dispositivi iOS, garantendo prestazioni e reattività migliorate.

Inoltre, la flessibilità e l'estensibilità di .NET MAUI sono potenziate dalla possibilità di sfruttare funzionalità aggiuntive tramite l'installazione di pacchetti utilizzando NuGet. Questo sistema di gestione dei pacchetti consente agli sviluppatori di integrare facilmente librerie, strumenti e componenti di terze parti nei loro progetti .NET MAUI, espandendo le capacità e le funzionalità delle loro applicazioni senza reinventare la ruota.

Infine, nonostante l'approccio di sviluppo unificato di .NET MAUI, gli sviluppatori mantengono l'autonomia nella selezione delle piattaforme di destinazione per la compilazione del software. Ciò significa che gli sviluppatori hanno la libertà di scegliere le piattaforme specifiche - che siano Android, iOS, Windows o qualsiasi altra piattaforma supportata - per le quali desiderano generare le applicazioni native, adattando il deployment per soddisfare le esigenze e i requisiti specifici della base utenti di destinazione. Questa flessibilità nella selezione della piattaforma consente agli sviluppatori di ottimizzare le loro applicazioni per diversi ecosistemi di dispositivi, garantendo ampia compatibilità e prestazioni su una gamma di dispositivi e sistemi operativi.

2.3. Flutter Framework

Flutter, un popolare framework open-source multi-piattaforma sviluppato da Google, mira a semplificare lo sviluppo di applicazioni mobili consentendo agli sviluppatori di creare app di alta qualità per le piattaforme iOS e Android utilizzando una singola base di codice.

Al centro della funzionalità di Flutter c'è il suo approccio allo sviluppo dell'interfaccia utente basato su widget. Flutter utilizza i widget come blocchi per la creazione di interfacce utente, consentendo agli sviluppatori di comporre elementi UI personalizzati con facilità. Questi widget vengono resi direttamente sulla canvas, bypassando i componenti UI nativi utilizzati nello sviluppo tradizionale delle app. Questo approccio garantisce un look and feel coerente sui dispositivi iOS e Android, consentendo agli sviluppatori di creare interfacce visualmente ricche e reattive.

Il motore di rendering di Flutter, Skia, svolge un ruolo cruciale nel fornire animazioni e grafica fluide su entrambe le piattaforme iOS e Android. Skia, una libreria grafica 2D open-source, alimenta le capacità di rendering ad alte prestazioni di Flutter, garantendo che le app costruite con Flutter offrano un'esperienza utente simile a quella nativa su dispositivi iOS e Android.

Per lo sviluppo su iOS con Flutter, il framework viene compilato in codice nativo ARM, consentendo alle app Flutter di essere eseguite direttamente sui dispositivi iOS con prestazioni ed efficienza elevate. Le app Flutter su iOS possono accedere senza problemi a funzionalità e API specifiche della piattaforma utilizzando plugin, consentendo agli sviluppatori di integrare funzionalità specifiche di iOS nelle loro applicazioni. (Haider, 2021)

Allo stesso modo, per lo sviluppo su Android, Flutter viene compilato anche in codice nativo ARM, garantendo prestazioni ottimali su dispositivi Android. Le app Flutter su Android possono sfruttare funzionalità specifiche della piattaforma tramite plugin, consentendo agli sviluppatori di accedere all'hardware del dispositivo, ai sensori, alle autorizzazioni e ad altre funzionalità specifiche di Android.

Inoltre, l'ampio set di plugin e pacchetti di Flutter fornisce agli sviluppatori accesso a una vasta gamma di soluzioni predefinite e integrazioni per compiti e funzionalità comuni. Questi plugin consentono l'integrazione di funzionalità native, come ad esempio l'accesso alla fotocamera, i servizi di geolocalizzazione e le notifiche push, per entrambe le piattaforme iOS e Android.

Queste caratteristiche rendono Flutter uno strumento potente per gli sviluppatori per creare app mobili robuste, visivamente accattivanti e performanti che offrono un'esperienza utente coerente su dispositivi iOS e Android.

2.3.1 Architettura di Flutter

L'architettura di Flutter è progettata attorno a diversi principi e concetti fondamentali che sostengono il suo efficiente e versatile framework per lo sviluppo di applicazioni mobili multi-piattaforma. A un livello elevato, l'architettura di Flutter ruota attorno ai seguenti componenti chiave e principi di progettazione:

L'architettura è incentrata sui widget, che sono i mattoni di base per la costruzione delle interfacce utente. Tutto in Flutter è un widget, consentendo agli sviluppatori di comporre elementi UI complessi combinando piccoli widget riutilizzabili. Questo approccio basato su widget semplifica lo sviluppo dell'interfaccia utente e supporta la creazione di interfacce utente personalizzate e interattive.

Flutter segue un'architettura stratificata che separa diversi componenti di un'applicazione, inclusi il motore di rendering, il framework e i widget. Questo approccio stratificato assicura una chiara separazione delle responsabilità e facilita la modularità, rendendo più facile gestire e mantenere applicazioni complesse. (Bhagat et Al., 2022)

Inoltre, questo framework consente la comunicazione con il codice specifico della piattaforma attraverso canali delle piattaforme e plugin. I canali delle piattaforme consentono alle app Flutter di interagire con il codice nativo scritto in linguaggi come Java (per Android) e Objective-C/Swift (per iOS), mentre i plugin forniscono accesso a funzionalità e API specifiche del dispositivo per ciascuna piattaforma.

Integrando questi principi e concetti fondamentali nella sua architettura, Flutter permette agli sviluppatori di costruire applicazioni multi-piattaforma di alta qualità, visivamente accattivanti e performanti con un'esperienza utente coerente su dispositivi iOS e Android.

Flutter è progettato come un sistema stratificato ed estensibile. Stratificato perché è un sistema strutturato in modo che diversi servizi del framework siano divisi in vari strati di componenti, dove ogni componente ha un compito specifico ben definito da svolgere.

Questo design favorisce la modularità. Estensibile perché ogni funzionalità di alto livello esiste come una serie di librerie indipendenti - note come pacchetti - dove ognuna di esse dipende dal nucleo sottostante chiamato Flutter Engine. (Carius et Al., 2022)
Per questo motivo il framework Flutter è relativamente piccolo. Gli sviluppatori scelgono cosa aggiungere e questo aiuta nel lungo termine a mantenere le dimensioni dell'applicazione ridotte. Flutter è composto da tre strati principali:

1. Flutter Engine: Espone le primitive necessarie per supportare tutte le applicazioni Flutter ed è principalmente scritto in C++. Ogni volta che è necessario dipingere un nuovo frame, il motore è responsabile della rasterizzazione - la conversione di un'immagine da un formato grafico vettoriale a un'immagine bitmap - della scena composta dall'albero dei widget. Fornisce l'implementazione a basso livello dell'API di base di Flutter, inclusa la grafica (tramite Skia), la disposizione del testo, l'I/O file e di rete, il supporto per l'accessibilità, l'architettura dei plugin e un runtime Dart e un toolchain di compilazione. Il motore è esposto al framework Flutter attraverso dart:ui, che avvolge il codice C++ sottostante in classi Dart. Questa libreria espone le primitive di livello più basso, come classi per guidare l'input, la grafica e i sottosistemi di rendering del testo.
2. Framework Flutter: Gli sviluppatori di alto livello interagiscono con Flutter attraverso il framework Flutter, che fornisce un framework moderno e reattivo scritto nel linguaggio Dart. Include un ricco insieme di librerie di piattaforma, layout e fondamentali, composte da una serie di strati. Lavorando dal basso verso l'alto, abbiamo:
    a. Classi di base fondamentali e servizi di costruzione come animazioni, disegno e gesti che offrono astrazioni comunemente utilizzate sulla base di fondamenta sottostanti.
    b. Lo strato di rendering, che fornisce un'astrazione per gestire il layout. Viene utilizzato nella costruzione di un albero di oggetti renderizzabili che possono

essere cambiati dinamicamente e auto-aggiornati riflettendo le modifiche nel layout.

c. Lo strato dei widget, che renderizza gli oggetti nello strato di rendering ha una classe corrispondente nello strato dei widget. Inoltre, lo strato dei widget consente agli sviluppatori di definire classi personalizzate riutilizzabili. Questo è lo strato in cui viene introdotto il modello di programmazione reattiva - la programmazione reattiva è un modello di sviluppo costruito attorno a flussi di dati asincroni e propagazione delle modifiche.
d. Le librerie Material e Cupertino, che implementano il design e i comportamenti di Material e iOS - attraverso un set completo di componenti predefiniti basati su primitive dello strato dei widget.

3. Incorporatore. C'è un incorporatore della piattaforma per ogni piattaforma supportata e fornisce un punto di ingresso per le funzioni principali del motore; L'incorporatore lavora in coordinamento con il sistema operativo sottostante per utilizzare superfici di rendering e accedere all'input dati dell'utente, nonché gestire il ciclo di eventi del messaggio. L'incorporatore è scritto in un linguaggio appropriato per la piattaforma: Java e C++ per Android, Objective-C/Objective-C++ per iOS e macOS e C++ per Windows e Linux. Grazie allo strato dell'incorporatore, il codice Flutter può essere integrato in un'applicazione esistente e distribuito come app autonomo.

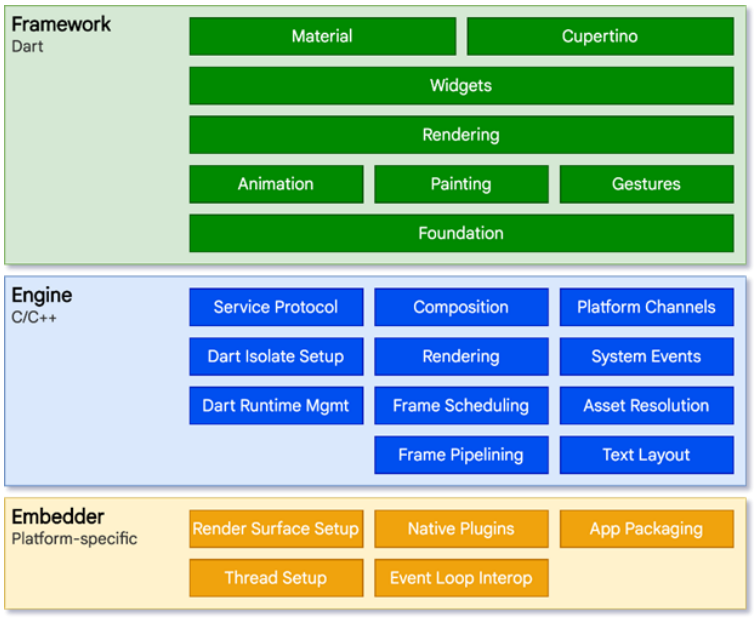

Figura 3 - I tre principali layers che compongono l'architettura interna di Flutter

2.3.2 Caratteristiche di Flutter

Al centro delle capacità di rendering di Flutter si trova il motore grafico Skia, una potente libreria grafica 2D che gestisce il rendering, le animazioni e la composizione grafica. Skia consente a Flutter di offrire visivi fluidi e performanti su diverse piattaforme, garantendo un'esperienza utente coerente.

Anche in Flutter ritroviamo il ricaricamento rapido (Hot Reload), che consente agli sviluppatori di visualizzare istantaneamente le modifiche apportate al codice riflesse nell'interfaccia utente dell'app in tempo reale. Questa funzionalità consente un'iterazione rapida, una risoluzione rapida dei problemi e uno sviluppo dell'interfaccia utente più rapido, senza la necessità di ricompilazioni dispendiose in termini di tempo. (Bhagat et Al., 2022)

Flutter consente la comunicazione con il codice specifico della piattaforma tramite canali di piattaforma e plugin. I canali di piattaforma consentono alle app Flutter di interagire con il codice nativo scritto in linguaggi come Java (per Android) e Objective-C/Swift (per iOS), mentre i plugin forniscono accesso alle funzionalità e alle API specifiche del dispositivo per ciascuna piattaforma.

Inoltre, Flutter fornisce due set di widget specifici per il design: i widget Material Design per Android e i widget Cupertino per iOS, per creare componenti dell'interfaccia utente

coerenti con la piattaforma. Ciò consente agli sviluppatori di attenersi alle linee guida dell'interfaccia utente di ciascuna piattaforma mantenendo una codebase unificata.

Lo sviluppatore che sceglie di usare Flutter si interfaccia con quest'ultimo mediante un apposito framework denominato "Flutter framework" impiegando il linguaggio Dart, un linguaggio orientato agli oggetti e con garbage collection sviluppato da Google, come linguaggio principale per lo sviluppo dell'applicazione.

La semplicità, le prestazioni e la facilità di apprendimento di Dart contribuiscono all'efficienza e alla facilità d'uso di Flutter.

Capitolo 3: Confronto Sperimentale tra .NET MAUI e Flutter

Nel campo dello sviluppo di applicazioni mobili, la scelta della tecnologia giusta può influenzare significativamente le prestazioni, l'efficienza e la fruibilità del prodotto finale. In questo contesto, emerge la necessità di valutare attentamente le diverse opzioni disponibili per garantire il successo del progetto. Questo studio si propone di affrontare questa sfida attraverso l'analisi comparativa di tre approcci di sviluppo di applicazioni mobili: lo sviluppo nativo, .NET MAUI e Flutter. Attraverso l'implementazione di tre applicazioni di esempio, ci concentreremo sul confronto diretto delle prestazioni tra queste tre modalità di sviluppo.

L'obiettivo principale di questo studio è quello di fornire una valutazione empirica delle prestazioni di ciascuna tecnologia, consentendo agli sviluppatori e alle aziende di prendere decisioni informate riguardo alla scelta della piattaforma più adatta alle proprie esigenze. Per raggiungere questo obiettivo, adotteremo uno scenario di carico di lavoro che simula operazioni matematiche intensive ed esegue un’operazione di rendering, consentendoci di valutare in modo accurato l'efficacia di ciascuna modalità di implementazione.

È bene ricordare che la velocità di esecuzione è fortemente influenzata dalle pratiche di codifica, dalle scelte algoritmiche e dalle tecniche di ottimizzazione implementate dagli sviluppatori. Al contrario, caratteristiche come la compatibilità e la portabilità dipendono maggiormente dalle piattaforme di destinazione selezionate dagli sviluppatori. nelle prestazioni all'approccio di sviluppo scelto senza l'interferenza di altri fattori.

Analizzare specificamente la velocità di esecuzione aiuta nella valutazione dell'impatto delle ottimizzazioni a livello di codice implementate dagli sviluppatori. Questo può

includere aspetti come la selezione degli algoritmi, le strutture dati e le pratiche di codifica che influenzano direttamente la velocità delle operazioni.

3.1 Lo standard ISO/IEC 25010

Per garantire una valutazione completa e accurata, ci baseremo sui criteri definiti nello standard ISO 25010, che fornisce una guida approfondita per la valutazione della qualità del software.

Lo standard ISO/IEC 25010:2011, incluso nella più ampia serie di standard ISO/IEC 25000 conosciuta come SQuaRE (Software Quality Requirements and Evaluation), svolge un ruolo cruciale nel facilitare la valutazione della qualità del software in varie fasi del ciclo di vita dello sviluppo del software. Questo standard costituisce un quadro completo progettato per definire e valutare la qualità dei sistemi e dei prodotti software.

ISO/IEC 25010 è fondamentale nella valutazione dei sistemi software esistenti. Le organizzazioni possono sfruttare questo standard per condurre valutazioni approfondite della qualità dei sistemi legacy, identificando aree di miglioramento e ottimizzazione. Allineandosi ai modelli di qualità specificati nello standard, le aziende possono analizzare e migliorare sistematicamente le prestazioni, la sicurezza e la qualità complessiva dei loro asset software esistenti.

Nel contesto della valutazione della qualità del software secondo lo standard ISO/IEC 25010, è fondamentale comprendere le diverse caratteristiche e sotto-caratteristiche coinvolte nel processo, di seguito descritte:

- Adeguamento funzionale: valuta il grado in cui il software soddisfa i requisiti funzionali specificati e gli obiettivi. È importante garantire la completezza delle funzioni implementate, la correttezza delle operazioni e l'appropriatezza delle funzioni rispetto alle esigenze degli utenti. (Falco & Robiolo, 2021)
- Compatibilità: valuta la capacità del software di operare efficacemente con altri sistemi e software. Questo include la coesistenza con altri software nell'ambiente, l'interoperabilità e la compatibilità delle interfacce con diverse piattaforme.
- Usabilità: valuta il grado di user-friendlyness del software, la facilità d'uso e la

capacità di migliorare la soddisfazione dell'utente. Elementi come la comprensibilità, l'apprendibilità e l'operabilità sono fondamentali per garantire un'esperienza utente positiva.

- Affidabilità: è relativa alla capacità del software di operare in modo coerente e prevedibile in varie condizioni. Questo include la maturità del software, la tolleranza ai guasti e la capacità di recuperare rapidamente da eventuali problemi.
- Sicurezza: garantisce la protezione delle informazioni del software. Questo include la confidenzialità dei dati, l'integrità delle informazioni e la prevenzione del rifiuto delle azioni.
- Manutenibilità: valuta quanto sia facile mantenere e supportare il software nel suo ciclo di vita. Elementi come l'analizzabilità, la modificabilità e la testabilità sono fondamentali per garantire un processo di manutenzione efficiente.
- Portabilità: valuta la facilità con cui il software può essere trasferito da un ambiente all'altro. Questo include l'adattabilità a diversi ambienti, l'installabilità e la sostituibilità dei componenti software.

Lo standard ISO/IEC 25010 rappresenta un'evoluzione diretta dello standard ISO/IEC 9126:2001 nel campo dell'ingegneria del software, arricchito dall'introduzione di nuovi parametri e criteri di valutazione. Questa trasformazione riflette l'evoluzione delle esigenze nel settore e l'espansione delle dimensioni considerate per valutare la qualità dei prodotti software. (Estdale & Georgiadou, 2018)

Recentemente, lo standard ISO/IEC 25010 è stato soggetto a revisione insieme ad altri standard appartenenti alla famiglia SQuaRE. Tra questi, vi sono l'ISO/IEC 25002:2024 che fornisce una panoramica e linee guida sull'utilizzo del modello di qualità, l'ISO/IEC 25010:2023 che dettaglia il modello di qualità del prodotto, e l'ISO/IEC 25019:2023 che presenta il modello di qualità in uso.

Questi aggiornamenti riflettono l'impegno continuo nel migliorare la valutazione della qualità del software e nell'adattare gli standard alle esigenze emergenti del settore.

Figura 4 – Evoluzione degli standard ISO/IEC per la valutazione del software

3.1.1. La valutazione delle performance nello standard ISO/IEC 25010

La valutazione delle prestazioni nello standard ISO/IEC 25010 si concentra sull'analisi della velocità con cui un sistema software esegue le sue operazioni. La velocità, come aspetto critico delle prestazioni, influenza direttamente l'efficienza e la reattività delle applicazioni software.

L'ottimizzazione degli algoritmi e delle strutture dati è essenziale per migliorare la velocità. Algoritmi efficienti riducono i calcoli superflui e migliorano le prestazioni complessive. Una gestione efficace delle risorse, come l'allocazione e l'utilizzo della memoria, può migliorare la velocità minimizzando i costi indiretti e migliorando i tempi di accesso ai dati. Utilizzare tecniche di elaborazione parallela può accelerare le operazioni distribuendo i compiti su più core o processori, aumentando così le prestazioni complessive del sistema.

I test di prestazione sono un approccio comune utilizzato per valutare la velocità. Questo include la conduzione di test di carico, test di stress e test di scalabilità per misurare le prestazioni del software in condizioni e carichi di lavoro diversi.

Gli strumenti di profilatura vengono utilizzati per analizzare le prestazioni dei sistemi software identificando colli di bottiglia, operazioni intensiva di risorse e aree da ottimizzare per migliorare la velocità.

3.2. Implementazione delle Applicazioni Sperimentali

Nel corso di questo esperimento, ci concentreremo sull'analisi delle performance di tre diverse modalità di implementazione attraverso lo sviluppo e l'esecuzione di tre applicazioni distinte. Ogni applicazione sarà progettata per eseguire un carico di lavoro generico ed un'operazione di rendering.

Il carico di lavoro generico verrà ottenuto simulando operazioni matematiche intensive mediante l'addizione di funzioni sinusoidali, cosinusoidali e di radice cubica eseguite su un contatore variabile per un totale di 2 milioni di iterazioni mentre l'operazione di rendering mediante l'aggiunta di una semplice immagine.

Le tre modalità di implementazione prescelte includono un'applicazione nativa Android, un'applicazione basata su .NET MAUI e un'applicazione sviluppata utilizzando il framework Flutter.

Questo esperimento mira a fornire una valutazione comparativa delle performance delle diverse modalità di sviluppo, consentendo di identificare eventuali differenze significative e di trarre conclusioni sulla loro efficacia e adattabilità in contesti di utilizzo reali.

3.2.1. Implementazione dell'Applicazione con .NET MAUI

Per realizzare un'applicazione su .NET MAUI, il processo inizia con lo sviluppo del codice sorgente su un ambiente di sviluppo integrato (IDE) come Visual Studio o Visual Studio for Mac. L'applicazione viene scritta utilizzando il linguaggio di programmazione C#, che offre una sintassi chiara e intuitiva per la creazione di applicazioni multipiattaforma. Una volta completata la fase di sviluppo, il codice viene compilato in app native per le piattaforme di destinazione selezionate.

Un aspetto importante di .NET MAUI è la flessibilità nella scelta delle piattaforme di destinazione. Gli sviluppatori hanno la possibilità di selezionare le piattaforme desiderate per le quali desiderano compilare il software. All'interno dell'IDE, come Visual Studio for Mac, è possibile configurare facilmente le piattaforme di destinazione abilitando le opzioni desiderate. Ad esempio, è possibile abilitare Android o iOS come piattaforma di destinazione selezionando le opzioni appropriate.

Successivamente, è stato necessario specificare la versione di .NET da utilizzare e la versione delle piattaforme di destinazione. Nel caso specifico, è stata selezionata la versione .NET 7 insieme ad Android 13 come riferimento, corrispondente all'API Level 33. Inoltre, è stato definito il requisito minimo per l'applicazione, che è stato impostato su Android 5 (API Level 21). Questi parametri sono essenziali per garantire la compatibilità dell'applicazione con le diverse versioni di Android e per definire il livello di supporto offerto per le funzionalità più recenti e avanzate.

Di seguito sono riportati i codici sorgente XAML delle pagine MainPage e GuiPage.

MainPage.xaml crea una semplice pagina contenente un Label (etichetta) e due Button (pulsanti), tutti allineati verticalmente all'interno di un StackLayout. Il Label è inizialmente vuoto, mentre i Button sono pronti per gestire il click dell'utente attraverso l'evento Clicked. Il primo Button sarà gestito da un metodo chiamato ExecuteMAUILoad definito nel codice sorgente associato mentre il secondo Button verrà gestito dal metodo ExecuteGUILoad.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MauiPerformance.MainPage"
             Title="MauiPerformance">

    <StackLayout BackgroundColor="White">
        <Label x:Name="mainLabel" VerticalOptions="CenterAndExpand" HorizontalOptions="CenterAndExpand" FontSize="25"
TextColor="#000000" ></Label>
        <StackLayout HorizontalOptions="CenterAndExpand" VerticalOptions="CenterAndExpand">
            <Button x:Name="mainBtn" HorizontalOptions="Center" VerticalOptions="Center" Clicked="ExecuteMAUILoad"></Button>
            <Button x:Name="guiBtn" HorizontalOptions="Center" VerticalOptions="Center" TranslationY="10"
Clicked="ExecuteGUILoad"></Button>
        </StackLayout>
    </StackLayout>
</ContentPage>
```

GuiPage.xaml crea una semplice pagina contenente un’immagine ed un Button centrati orizzontalmente all’interno della pagina. L’immagine non viene ancora caricata mentre il pulsante gestirà il metodo ExecuteGUILoad in seguito alla pressione.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MauiPerformance.GuiPage"
             Title="GuiPage">

    <StackLayout BackgroundColor="White">
        <Image x:Name="MainImage" VerticalOptions="CenterAndExpand" HorizontalOptions="CenterAndExpand"></Image>
        <Button x:Name="resultBtn" VerticalOptions="Center" HorizontalOptions="Center" TranslationY="-150"
Clicked="DisplayGUIResult"></Button>
    </StackLayout>
</ContentPage>
```

Di seguito invece viene riportato il codice C# che definisce la classe MainPage che, a sua volta, rappresenta la pagina principale dell'applicazione MAUI. Nel costruttore, vengono inizializzati i componenti UI e impostato il testo dell'etichetta e dei pulsanti. Il metodo ExecuteMAUILoad viene eseguito quando l'utente fa clic sul pulsante e simula un carico computazionale utilizzando la classe Stopwatch per misurare il tempo impiegato. Infine, viene visualizzato un messaggio di alert con il tempo di esecuzione.

Il metodo ExecuteGUILoad mostra a video la pagina su cui viene eseguito il rendering di un'immagine.

```
using System.Diagnostics;

namespace MauiPerformance
{

    public partial class MainPage : ContentPage
    {
        public MainPage()
        {
            InitializeComponent();
            mainLabel.Text = "Applicazione MAUI";
            mainBtn.Text = "Esegui carico (CPU)";
            guiBtn.Text = "Esegui carico (UI)";
        }

        void ExecuteMAUILoad(System.Object sender, System.EventArgs e)
        {
```

```
        var stopwatch = Stopwatch.StartNew();

        // Simula un carico pesante
        for (int i = 0; i < 2000000; i++)
        {
            // Operazione di carico simulata
            var result = Math.Sin(i) + Math.Cos(i) + Math.Pow(i,(1/3));
        }

        stopwatch.Stop();
        DisplayAlert("Carico Completato", $"Tempo impiegato: {stopwatch.ElapsedMilliseconds} ms", "OK");
    }

    void ExecuteGUILoad(System.Object sender, System.EventArgs e)
    {
        Navigation.PushAsync(new GuiPage());
    }
  }
}
```

Viene riportato inoltre il codice C# che definisce la classe GuiPage. Nel costruttore, anche in questo caso, vengono inizializzati i componenti UI ma al contempo anche dei parametri utili per la valutazione del tempo di rendering dell'immagine tra cui il tempo stesso in millisecondi.

All'atto dell'inizializzazione dell'immagine viene anche avviato un cronometro mediante la classe Stopwatch in modo tale da misurare il tempo che impiega l'immagine per renderizzarsi.

Nel momento in cui l'immagine si sarà renderizzata completamente, verrà attivato l'evento OnPageSizeChanged e di conseguenza alla ricezione di quest'ultimo il cronometro viene interrotto ed il tempo impiegato viene salvato. Infine, alla pressione del pulsante dichiarato in precedenza, viene richiamato il metodo DisplayGUIResult che mostra a video un messaggio di alert (similmente al caso precedente) contenente il tempo di rendering dell'immagine.

```
using System.Diagnostics;

namespace MauiPerformance
{
```

```
public partial class GuiPage : ContentPage
{
    private Stopwatch;
    private bool imageRendered;
    private long tempoRendering;

    public GuiPage()
    {
        InitializeComponent();
        tempoRendering= 0;
        imageRendered = false;
        MainImage.Source = "san_giovanni.png";
        resultBtn.Text = "Mostra risultati";
        stopwatch = new Stopwatch();
        stopwatch.Start();

        //Misuro il tempo di rendering ascoltando l'evento SizeChanged
        SizeChanged += OnPageSizeChanged;

    }

    void DisplayGUIResult(System.Object sender, System.EventArgs e)
    {
        if (imageRendered)
        {
            DisplayAlert("Rendering Completato", $"Tempo impiegato: {tempoRendering} ms", "OK");
        } else
        {
            DisplayAlert("Potrebbe essersi verificato un problema", "Il rendering dell'immagine non è ancora stato completato", "OK");
        }
    }

    private async void OnPageSizeChanged(object sender, EventArgs e)
    {
        //Quando viene generato l'evento interrompo il cronometro e mostro il tempo di rendering in millisecondi
        stopwatch.Stop();
        imageRendered = true;
        tempoRendering=stopwatch.ElapsedMilliseconds;
    }
  }
}
```

Infine, il codice viene eseguito e testato su un dispositivo reale per verificare il corretto funzionamento dell'applicazione su una piattaforma fisica. In questo caso specifico, l'applicazione è stata realizzata impiegando .NET 7 e sarà eseguita su un Samsung Galaxy

Z Fold 5 che utilizza Android 14 come sistema operativo.

L'applicazione MAUI si presenta sul dispositivo con la schermata seguente, dove premendo su "Esegui carico (CPU)" viene avviato un ciclo di calcolo e successivamente viene mostrato a video un popup che visualizza il tempo in millisecondi impiegato per completare l'operazione.

Figura 5.1 – Interfaccia Grafica della pagina principale dell'Applicazione sviluppata con .NET MAUI

Premendo invece il pulsante "Esegui carico (UI)" viene immediatamente caricata una nuova pagina contenente un'immagine e valutato il tempo impiegato affinché quest'ultima renderizzi completamente. È inoltre presente un pulsante che in seguito ad una pressione mostra un popup contenente il parametro misurato.

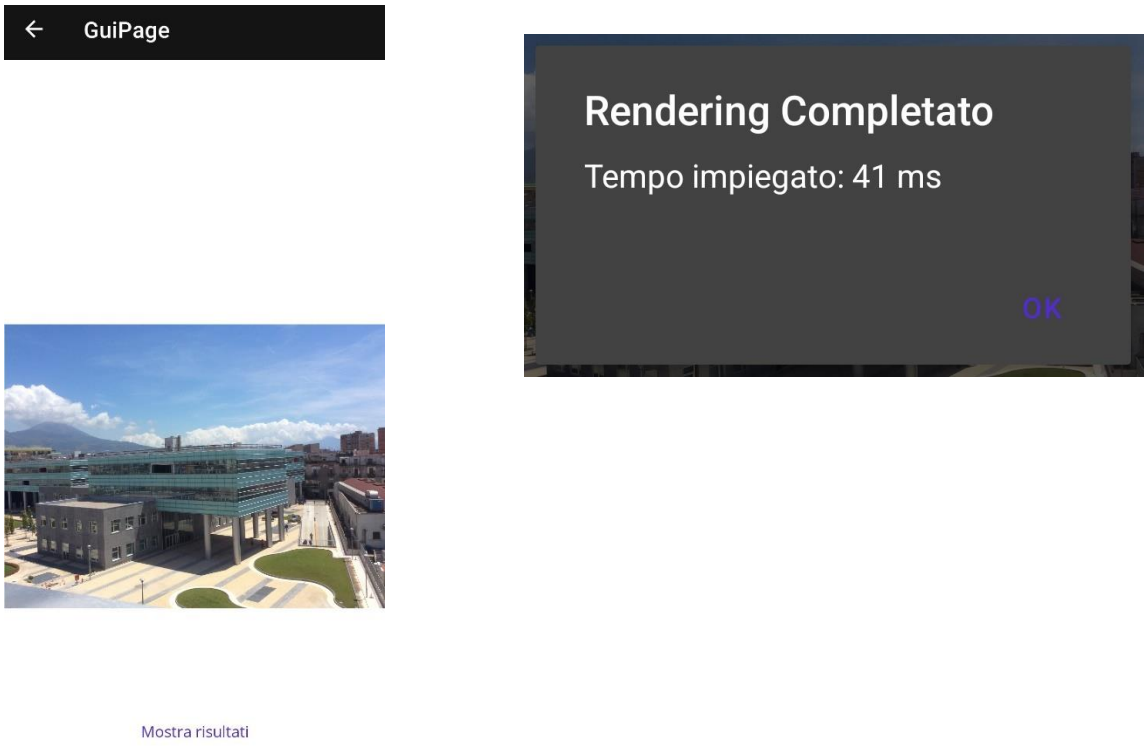

Figura 5.2 – Interfaccia Grafica della GuiPage e del popup che appare alla pressione del pulsante

3.2.2. Implementazione dell'Applicazione con Flutter

Per realizzare un'applicazione su Flutter innanzitutto viene modificato il file build.gradle che viene utilizzato per configurare le impostazioni di compilazione dell'applicazione per la piattaforma Android. Questo file si trova nella directory /android/app/ del progetto Flutter. Una delle configurazioni chiave nel file build.gradle è la specifica delle versioni minime e di destinazione di Android attraverso i parametri minSdkVersion e targetSdkVersion. Questi parametri determinano la versione minima di Android richiesta per eseguire l'applicazione e la versione di Android che viene presa come destinazione per lo sviluppo dell'applicazione. Ad esempio, nel codice fornito, la versione minima di Android è impostata su Android 5 (API Level 21), mentre la versione di Android scelta come target è Android 13 (API Level 33).

Questo significa che l'applicazione Flutter sarà compatibile con dispositivi Android che eseguono Android 5 o versioni successive e che sarà ottimizzata per funzionare al meglio su dispositivi con Android 13.

La modifica di queste impostazioni nel file build.gradle consente agli sviluppatori di specificare i requisiti di compatibilità e le funzionalità di destinazione dell'applicazione per la piattaforma Android. Questa scelta è a discrezione dello sviluppatore e dipende dalle esigenze dell'applicazione e dal pubblico di destinazione.

Di seguito è riportato il codice in linguaggio Dart che definisce un'applicazione Flutter con una schermata principale contenente due pulsanti. Quando il primo pulsante viene premuto, viene eseguita una simulazione di carico pesante utilizzando funzioni matematiche. Alla fine della simulazione, viene mostrato un dialogo con il tempo impiegato per il carico.

All'atto della pressione del secondo pulsante, viene mostrata una nuova pagina in cui viene caricata un'immagine ed un pulsante.

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

void main() {
  WidgetsFlutterBinding.ensureInitialized();
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: MyPerformancePage(),
    );
  }
}

class MyPerformancePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Applicazione Flutter'),
      ),
      body: Center(
```

```
      child:Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
        ElevatedButton(
          onPressed: () => simulateLoad(context),
          child: Text('Esegui Carico (CPU)'),
        ),
        SizedBox(height: 20),
        ElevatedButton(
          onPressed: () => Navigator.push(context,MaterialPageRoute(builder: (context) => MyGuiPage())),
          child: Text('Esegui carico (UI)'),
      ),
      ],
      ),
      ),
    );
}

void simulateLoad(BuildContext context) {
  final stopwatch = Stopwatch()..start();

  // Simula un carico pesante
  for (int i = 0; i < 2000000; i++) {
    // Operazione di carico simulata
    final result = sin(i.toDouble())+cos(i.toDouble())+pow(i,(1/3));
  }

  stopwatch.stop();
  showDialog(
    context: context,
    builder: (BuildContext context) {
      return AlertDialog(
        title: Text('Carico Completato'),
        content: Text('Tempo impiegato: ${stopwatch.elapsedMilliseconds} ms'),
        actions: [
          TextButton(
            onPressed: () {
              Navigator.of(context).pop();
            },
            child: Text('OK'),
          ),
        ],
      );
    },
  );
 }
}
```

```
class Renderizzatore {
 final String imagePath;
 Stopwatch = Stopwatch();
 bool renderingCompleted = false;
 int renderingTime = 0;

 Renderizzatore({required this.imagePath});

 Future<void> loadAndRender() async {
  stopwatch.start();

  try {
   final image = await Image.asset(imagePath);
   renderingCompleted = true;
   renderingTime = stopwatch.elapsedMilliseconds;
   stopwatch.reset();
  } catch (error) {
   print("Si è verificato un errore durante il caricamento dell'immagine: $error");
  }
 }

 void displayLoadResult(BuildContext context) {
  if (renderingCompleted) {
   showDialog(
    context: context,
    builder: (BuildContext context) {
     return AlertDialog(
      title: Text('Rendering completato'),
      content: Text('Tempo impiegato: $renderingTime ms'),
      actions: [
       TextButton(
        onPressed: () => Navigator.of(context).pop(),
        child: Text('OK'),
       ),
      ],
     );
    },
   );
  } else {
   showDialog(
    context: context,
    builder: (BuildContext context) {
     return AlertDialog(
      title: Text('Potrebbe essersi verificato un problema'),
      content: Text('Il rendering non è ancora stato completato'),
      actions: [
       TextButton(
        onPressed: () => Navigator.of(context).pop(),
```

```
              child: Text('OK'),
            ),
          ],
        );
      },
    );
  }
 }
}

class MyGuiPage extends StatelessWidget {

 @override
 Widget build(BuildContext context) {
  final imageRenderer = Renderizzatore(imagePath: 'assets/san_giovanni.jpg');

  return Scaffold(
    appBar: AppBar(
      title: Text('GuiPage'),
    ),
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          FutureBuilder<void>(
            future: imageRenderer.loadAndRender(),
            builder: (context, snapshot) {
              if (snapshot.connectionState == ConnectionState.done) {
                return Image.asset('assets/san_giovanni.jpg');
              } else {
                return CircularProgressIndicator(); // Mostro un indicatore
              }
            },
          ),
          SizedBox(height: 60),
          ElevatedButton(
            onPressed: () => imageRenderer.displayLoadResult(context),
            child: Text('Mostra risultati'),
          ),
        ],
      ),
    ),
  );
 }

}
```

Questo codice definisce un'applicazione Flutter con una schermata principale che contiene due pulsanti.

Il primo è denominato "Esegui Carico (CPU)" mentre il secondo è denominato “Esegui Carico (UI)”.

Quando l'utente preme il primo pulsante, viene eseguita una simulazione di carico pesante. Questa simulazione coinvolge un ciclo for che esegue un gran numero di iterazioni (2 milioni di volte) e durante ciascuna iterazione vengono eseguite diverse operazioni matematiche utilizzando le funzioni sin(), cos() e pow().

Una volta completata la simulazione, viene mostrato un dialogo con il tempo impiegato per eseguire il carico. Il tempo viene misurato utilizzando un cronometro che inizia quando viene premuto il pulsante e si ferma quando la simulazione è completata. Infine, il dialogo offre un pulsante "OK" che permette all'utente di chiudere il dialogo e tornare alla schermata principale dell'applicazione.

Quando invece l’utente preme il secondo pulsante, viene caricata una nuova pagina in cui sono presenti un’immagine ed un pulsante.

Nel momento in cui l’immagine viene inizializzata, similmente al caso precedente viene avviato un cronometro che si interromperà nel momento in cui l’immagine viene caricata completamente.

All’atto della pressione del pulsante con etichetta “Mostra risultati” viene mostato un dialogo con il tempo impiegato per eseguire il carico.

L’applicazione Flutter presenta una interfaccia utente simile a quella dell’applicazione con .NET MAUI, come riportato nelle figure 6.1 e 6.2.

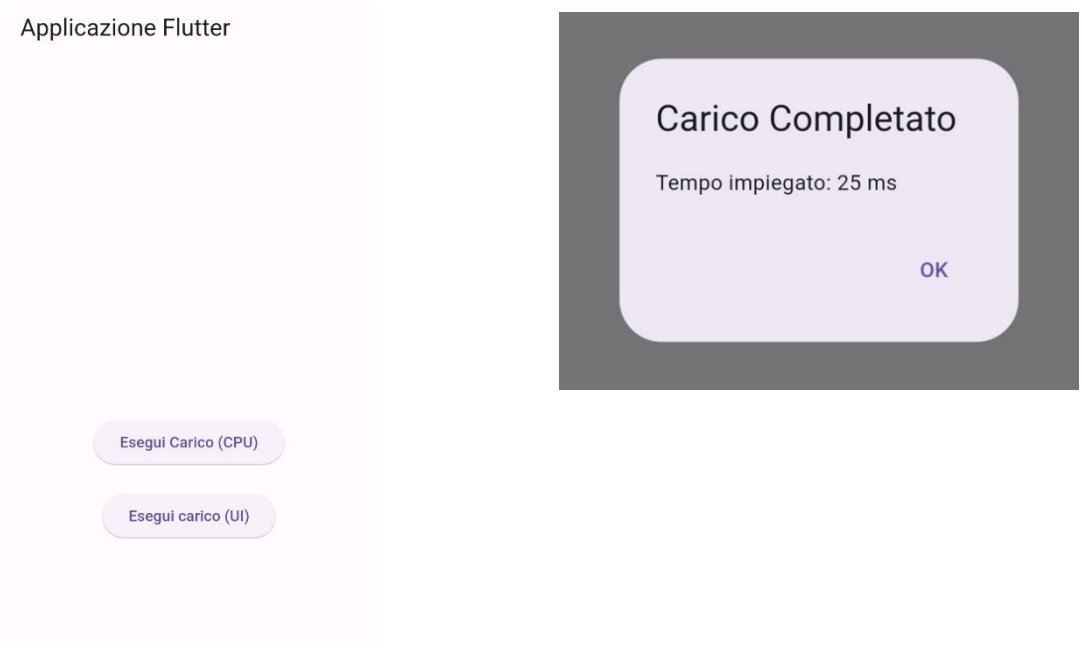

Figura 6.1 – Interfaccia Grafica dell'Applicazione con Flutter

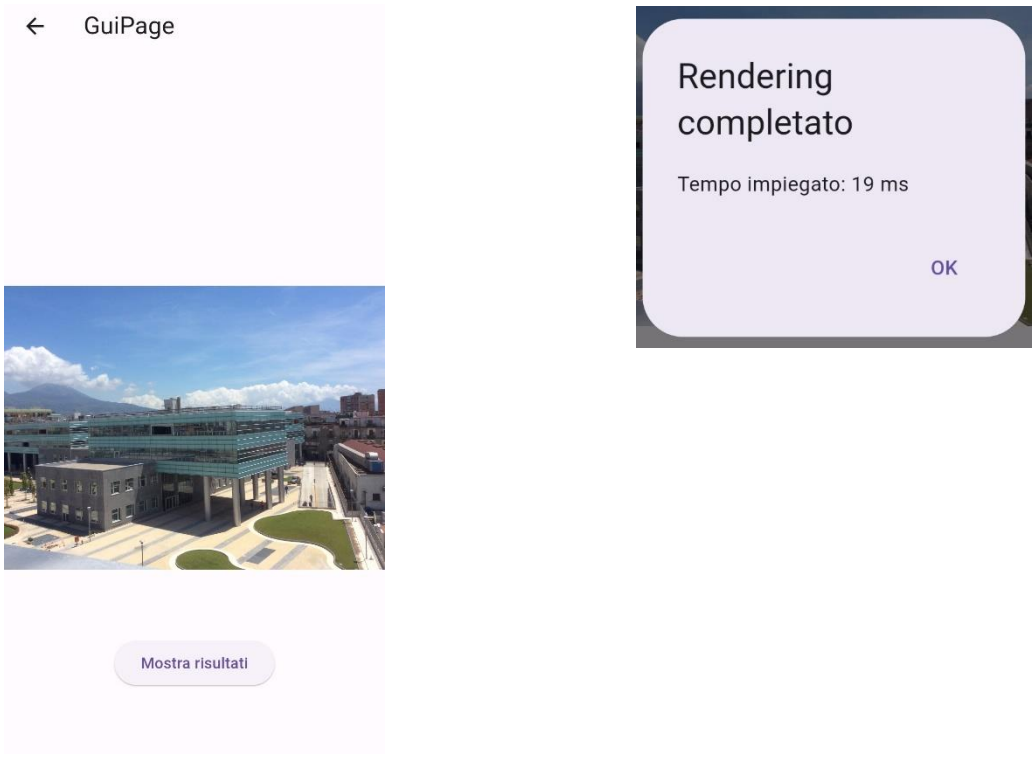

Figura 6.2 – Interfaccia Grafica della GuiPage in Flutter e del popup che appare alla pressione del pulsante

3.2.3. Implementazione dell'Applicazione Android Nativa

Infine, è stata sviluppata anche l'applicazione nativa per Android, impiegando il linguaggio Java ed XML.

Nello sviluppo nativo Android, il codice XML serve a definire il layout delle varie Activity. In questa applicazione abbiamo realizzato due Activity: MainActivity e GuiActivity.

Il codice XML seguente definisce l'interfaccia grafica della MainActivity utilizzando il linguaggio XML. La struttura principale è un RelativeLayout che contiene un TextView e due Button. Il TextView mostra un titolo centrato verticalmente nella schermata, mentre i Button sono posizionati sotto il TextView e al centro della schermata.

Il titolo indica che si tratta di un'applicazione Android nativa ed i Button sono etichettati come "Esegui Carico (CPU)" ed "Esegui Carico (UI)".

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

    <TextView
        android:id="@+id/titleTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="270dp"
        android:text="Applicazione Android (Nativa)"
        android:textAlignment="center"
        android:textSize="23sp" />

    <Button
        android:id="@+id/simulateCPULoadButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/titleTextView"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="20dp"
        android:text="Esegui carico (CPU)" />

    <Button
        android:id="@+id/simulateUILoad"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/simulateCPULoadButton"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="10dp"
        android:text="Esegui carico (UI)" />

</RelativeLayout>
```

Il codice Java definisce l'attività principale dell'applicazione Android. All'interno del metodo onCreate(), viene inizializzata l'interfaccia utente recuperando il TextView e il Button dal layout XML e impostando i listener per i Button. Quando il Button relativo al carico della CPU viene premuto, viene avviato un nuovo AsyncTask chiamato SimulateLoadTask per eseguire la simulazione di carico in background.

Invece, nel caso del button relativo alla UI viene creato un Intent che successivamente permetterà di avviare la GuiActivity.

Il metodo doInBackground() dell'AsyncTask esegue la simulazione di carico, calcolando il tempo impiegato per completare l'operazione. Una volta completata la simulazione, il metodo onPostExecute() viene chiamato per mostrare un dialogo con il tempo impiegato e aggiornare il titolo con un messaggio di "Simulazione Completata".

Il codice gestisce anche la liberazione delle risorse quando l'attività viene distrutta, e utilizza una funzione di utility per mostrare il dialogo e aggiornare l'interfaccia utente con il titolo.

```
package com.simonezurlo.naperformance;

import android.content.Intent;
import android.os.AsyncTask;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {
    private TextView titleTextView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        titleTextView = findViewById(R.id.titleTextView);
        Button simulateCPULoadButton = findViewById(R.id.simulateCPULoadButton);
        simulateCPULoadButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                new SimulateLoadTask().execute();
            }
        });
        Button simulateUILoadButton = findViewById(R.id.simulateUILoad);
        simulateUILoadButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent = new Intent(MainActivity.this, GuiActivity.class);
                startActivity(intent);
            }
```

```
        });
    }

    @Override
    protected void onDestroy() {
        // Libera eventuali risorse quando l'attività viene distrutta
        super.onDestroy();

    }

    private class SimulateLoadTask extends AsyncTask<Void, Void, Long> {
        @Override
        protected Long doInBackground(Void... voids) {
            long startTime = System.nanoTime();

            // Simula un carico pesante
            for (int i = 0; i < 2000000; i++) {
                double result = Math.sin(i)+Math.cos(i)+Math.pow(i,(1/3));
            }

            long endTime = System.nanoTime();
            return endTime - startTime;
        }

        @Override
        protected void onPostExecute(Long elapsedTime) {
            super.onPostExecute(elapsedTime);
            long elapsedTimeMs = elapsedTime / 1000000;
            String message = "Tempo impiegato: " + elapsedTimeMs + " ms";
            showDialog(message);
        }
    }

    private void showDialog(String message) {
        new androidx.appcompat.app.AlertDialog.Builder(this)
                .setTitle("Simulazione Completata")
                .setMessage(message)
                .setPositiveButton("OK", null)
                .show();
    }
}
```

Per quanto riguarda la GuiActivity, il codice XML seguente definisce anch'essa come struttura principale un RelativeLayout che contiene una ImageView ed un Button.

L'ImageView rappresenta un "contenitore" dell'immagine e come negli esempi precedenti è centrata orizzontalmente mentre il Button è posizionato al di sotto dell'ImageView, è centrato orizzontalmente ed è etichettato come "Mostra risultati".

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".GuiActivity">
```

```
    <ImageView
        android:id="@+id/mainImage"
        android:layout_width="413dp"
        android:layout_height="313dp"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="124dp"
        android:contentDescription="Immagine principale usata per il rendering"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/btnMostraRisultati"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="72dp"
        android:text="Mostra risultati"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.497"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/mainImage" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Nel caso della GuiActivity, il codice Java è il seguente:

```
package com.simonezurlo.naperformance;
import android.os.SystemClock;
import android.view.View;
import android.widget.Button;
import android.widget.ImageView;
import android.os.Bundle;
import android.view.MenuItem;
import androidx.appcompat.app.AppCompatActivity;
import java.util.Objects;

public class GuiActivity extends AppCompatActivity {

    private long renderingTime;
    private boolean renderingCompletato=false;
    private ImageView;

    private void showDialog() {
        if(renderingCompletato){
            new androidx.appcompat.app.AlertDialog.Builder(this)
                    .setTitle("Rendering completato")
                    .setMessage("Tempo impiegato: "+renderingTime+" ms")
                    .setPositiveButton("OK", null)
                    .show();
        }else{
            new androidx.appcompat.app.AlertDialog.Builder(this)
                    .setTitle("Potrebbe essersi verificato un problema")
                    .setMessage("Il rendering non è ancora stato completato.")
                    .setPositiveButton("OK", null)
                    .show();
        }

    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) { //Quando premo il tasto per tornare indietro l'azione di default
```

```
onBackPressed() viene invocata
        if (item.getItemId() == android.R.id.home) {
            getOnBackPressedDispatcher().onBackPressed();
            return true;
        }
        return super.onOptionsItemSelected(item);
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_gui);
        Objects.requireNonNull(getSupportActionBar()).setDisplayHomeAsUpEnabled(true); //Mi fa ottenere il pulsante indietro.
        //Inizializzo il pulsante che mostra i risultati
        Button btnMostraRisultati = findViewById(R.id.btnMostraRisultati);
        btnMostraRisultati.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                showDialog();
            }
        });
        //Inizializzo il contenitore dell'immagine (imageview)
        imageView = findViewById(R.id.mainImage);
        imageView.setImageResource(0); //Carico l'immagine
        long startTime = SystemClock.elapsedRealtime(); //Recupero il tempo di inizio del rendering
        imageView.setImageResource(R.mipmap.san_giovanni); //Carico l'immagine
        long endTime = SystemClock.elapsedRealtime(); //Recupero il tempo di fine del rendering
        renderingTime = endTime - startTime;
        renderingCompletato = true;
    }
}
```

All'interno del metodo onCreate() viene inizializzata ancora una volta l'interfaccia utente recuperando l'ImageView ed il Button dal layout XML ed impostato il listener per il Button. In questo caso, la GuiActivity non è l'activity principale dell'applicazione quindi mediante il metodo setDisplayHomeAsUpEnabled() presente all'interno di onCreate() richiedo che questa pagina abbia il classico pulsante in alto a sinistra per poter tornare indietro.

Inoltre mediante il metodo onOptionItemSelected() richiedo che alla pressione del pulsante ottenuto in precedenza l'azione ottenuta sia per l'appunto quella di "tornare indietro".

Successivamente l'immagine viene inizializzata ed il tempo impiegato per renderizzare l'immagine viene valutato similmente al caso del carico sulla CPU.

L'applicazione nativa presenta una UI simile a quelle precedenti e, ancora una volta, premendo su "Esegui carico (CPU)" otterremo un popup che riporta il risultato del test.

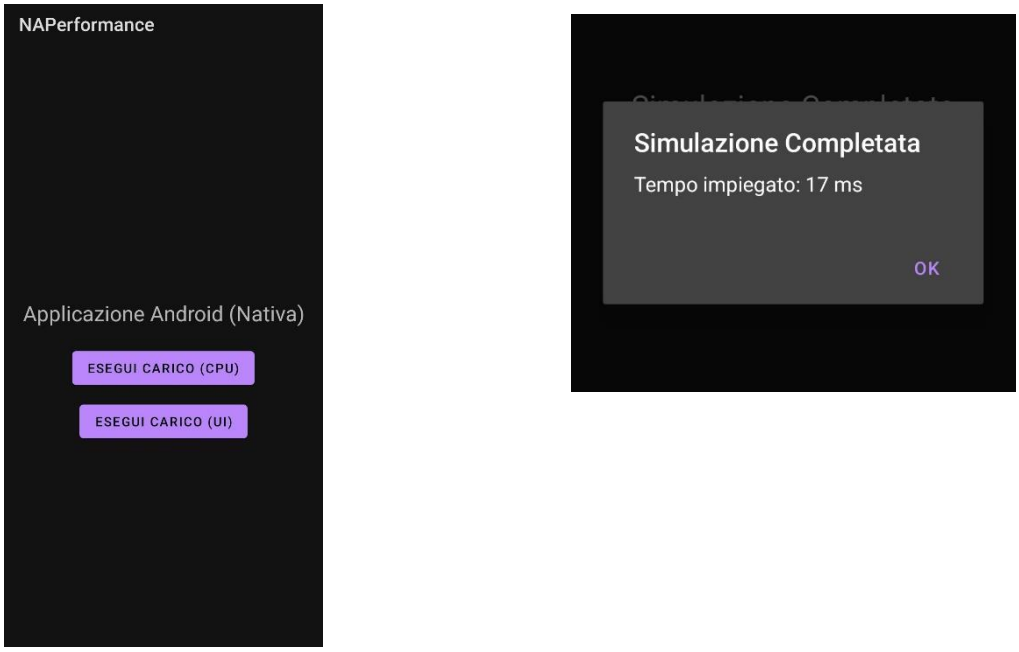

Figura 7.1 – Interfaccia Grafica dell'Applicazione realizzata nativamente

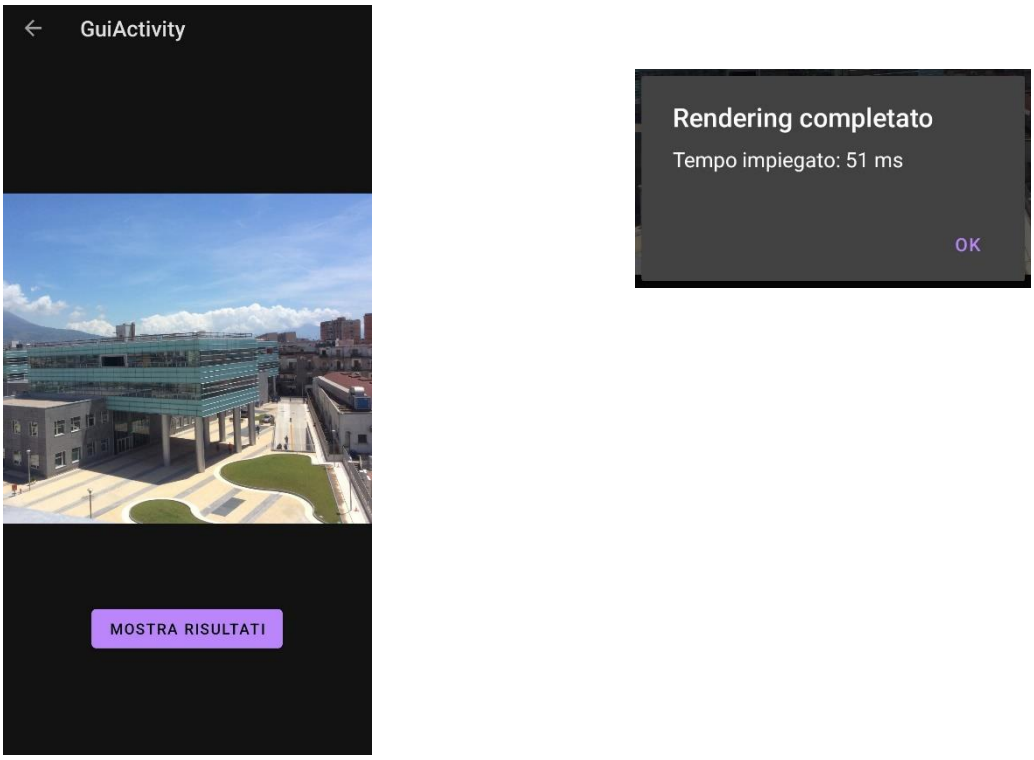

Figura 7.2 – Interfaccia Grafica della GuiActivity e del popup che viene mostrato alla pressione del pulsante

3.3. Risultati dei Test

Si vuole ora eseguire una comparazione dei tre diversi approcci di sviluppo di un'applicazione Android.

L'obiettivo principale del test è valutare diversi aspetti delle performance, tra cui il **tempo impiegato per l'esecuzione del carico cpu-bound**, **tempo di rendering**, la **dimensione del file .apk generato** e la **quantità di memoria (RAM) impiegata** dall'applicazione durante l'esecuzione. Questi aspetti sono fondamentali per determinare l'efficienza e l'ottimizzazione delle tre diverse metodologie di sviluppo.

Per valutare il tempo impiegato per l'esecuzione, il carico viene eseguito otto volte e si calcola il tempo medio impiegato. Questo fornisce un'indicazione affidabile delle prestazioni medie delle diverse implementazioni. Complessivamente, questo test fornisce una panoramica completa delle prestazioni delle tre diverse metodologie di sviluppo, consentendo di identificare punti di forza e debolezza di ciascuna e prendere decisioni informate sulle migliori pratiche da adottare nello sviluppo di applicazioni Android.

Di seguito sono riportati i dati acquisiti relativi al tempo di esecuzione per ciascuna delle 8 ripetizioni ed ognuna delle configurazioni di sviluppo, prima in forma tabellare e poi come grafico (figura 8.1):

Ripetizioni	**Tempo di Esecuzione (Carico CPU)**		
	.NET MAUI [ms]	***Flutter [ms]***	***Applicazione Nativa [ms]***
1	126	32	17
2	130	35	18
3	129	34	19
4	132	37	17
5	130	34	21
6	129	35	17
7	131	35	14
8	126	32	19
Media	**129,125**	**34,25**	**17,75**

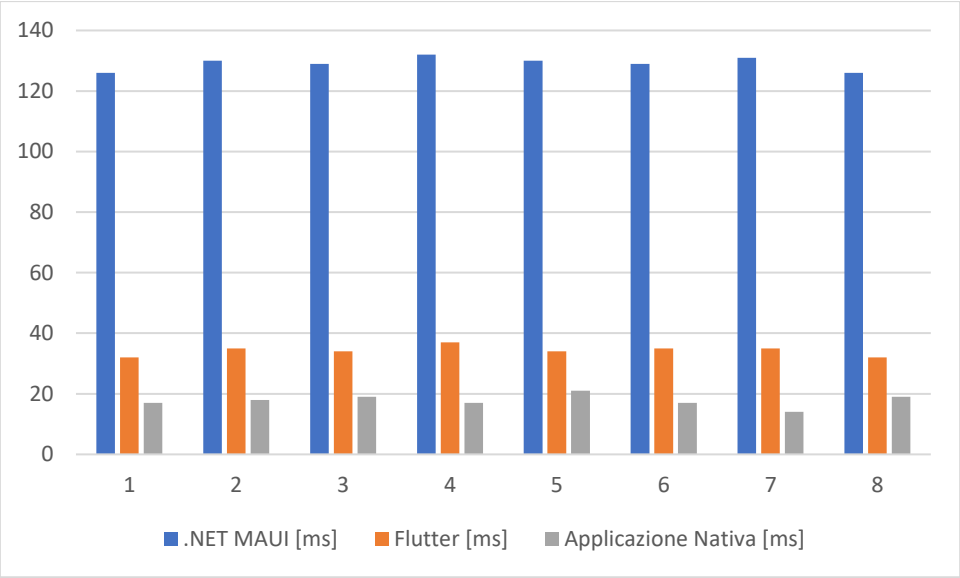

Figura 8.1 Rappresentazione in forma grafica del tempo di esecuzione del carico cpu-bound per ciascuna delle 8 ripetizioni nelle tre configurazioni di sviluppo

Per valutare invece il tempo di rendering dell'immagine, l'immagine verrà fatta renderizzare per 5 volte ed anche in questo caso valuteremo il tempo medio impiegato.

Ad ogni tentativo l'applicazione sarà chiusa completamente e poi aperta nuovamente.

Di seguito sono riportati i dati acquisiti relativi al tempo di rendering per ciascuna della 8 ripetizioni ed ognuna delle configurazioni di sviluppo ancora una volta in forma tabellare e poi come grafico (Figura 8.2).

Ripetizioni	**Tempo di Rendering**		
	.NET MAUI [ms]	***Flutter [ms]***	***Applicazione Nativa [ms]***
1	79	53	39
2	85	58	48
3	78	59	42
4	79	56	49
5	87	56	39
Media	**81,6**	**56,4**	**43,4**

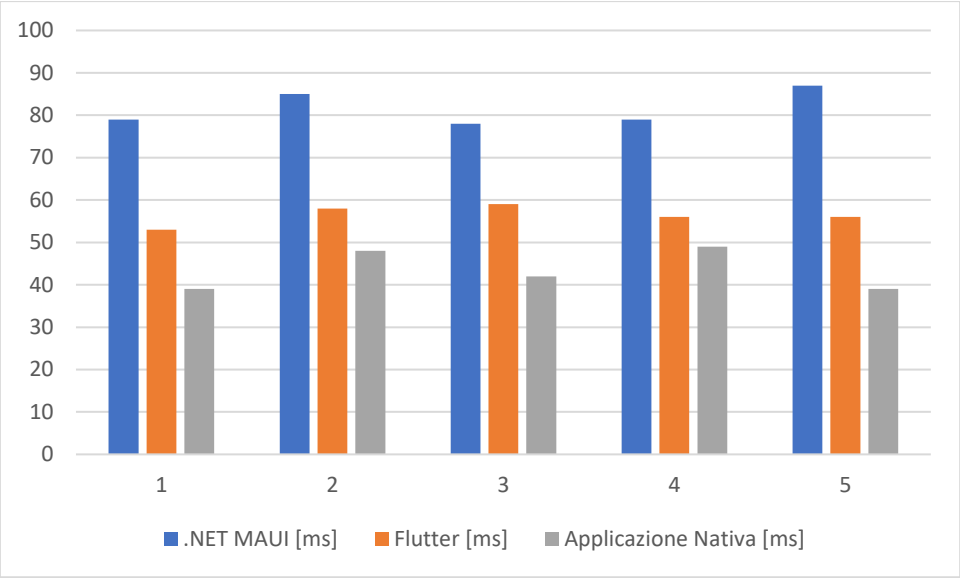

Figura 8.2 Rappresentazione in forma grafica del tempo di rendering per ciascuna delle 8 ripetizioni nelle tre configurazioni di sviluppo

La dimensione del file .apk generato fornisce informazioni sulla leggerezza o pesantezza dell'applicazione e può influenzare il tempo di download e l'utilizzo di risorse di archiviazione del dispositivo.

Nelle figure 9, 10 e 11 sono rispettivamente riportati i dati relativi alla dimensione delle applicazioni sviluppate.

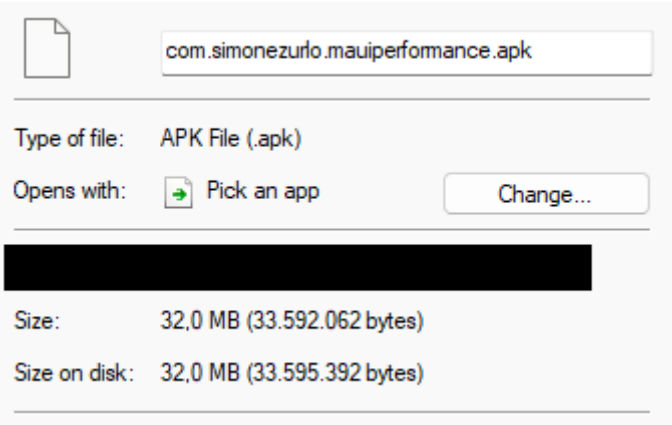

Figura 9. Spazio impiegato dall'applicazione sviluppata con .NET MAUI

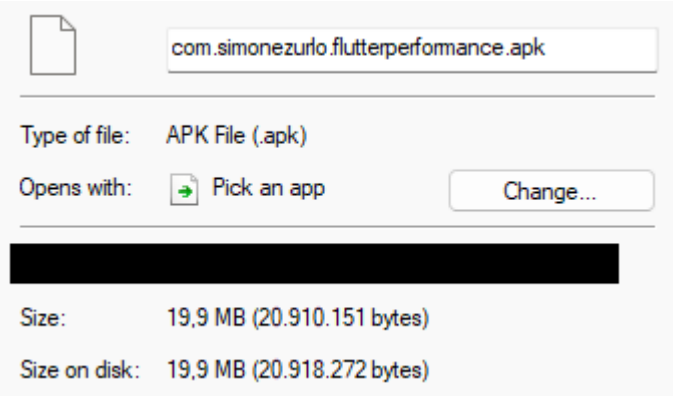

Figura 10 Spazio impiegato dall'applicazione sviluppata con Flutter

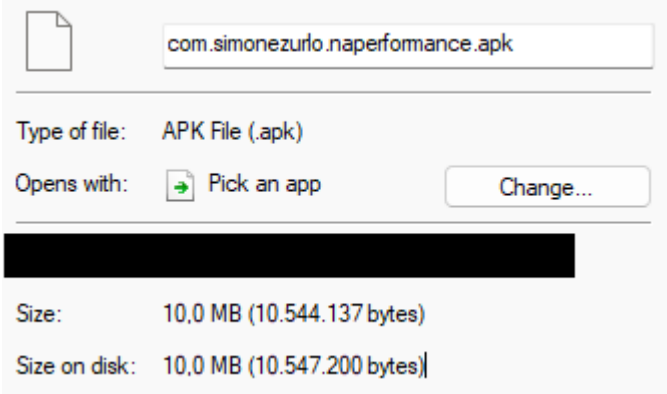

Figura 11 Spazio impiegato dall'applicazione sviluppata con l'approccio nativo

Infine, l'analisi della quantità di memoria (RAM) impiegata durante l'esecuzione fornisce informazioni cruciali sul consumo di risorse dell'applicazione. Accedendo con un Profiler mentre l'applicazione è in esecuzione, è possibile monitorare la quantità di memoria RAM utilizzata e identificare eventuali problemi di consumo eccessivo di risorse.

Di seguito nelle figure 12, 13 e 14 sono riportati gli andamenti delle risorse allocate nelle tre configurazioni di sviluppo.

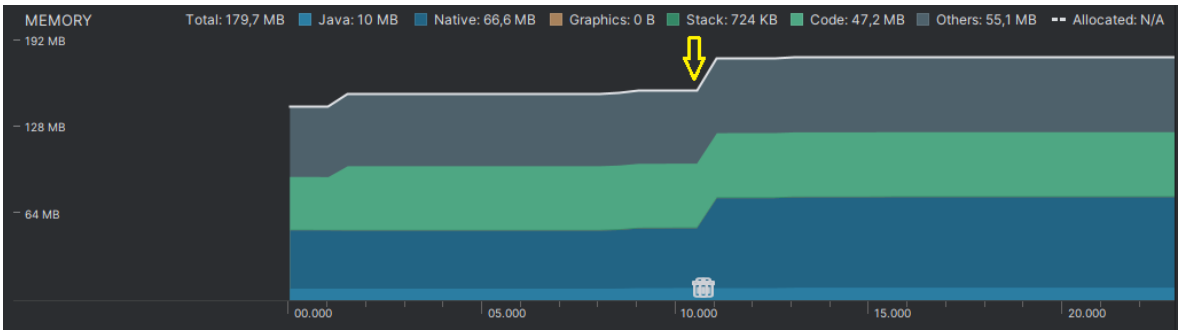

Figura 12 – Quantità di memoria allocata nell'applicazione .NET MAUI

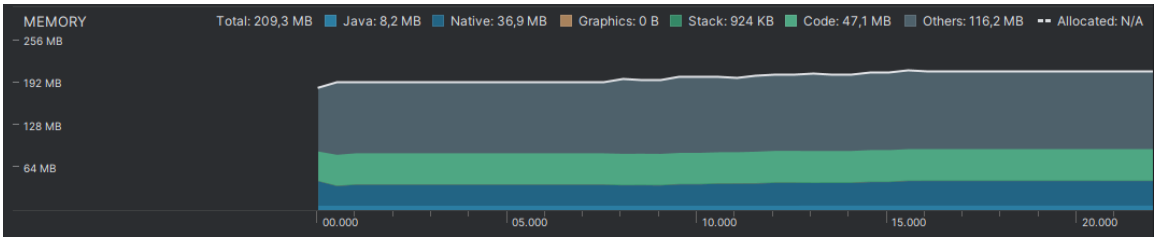

Figura 13 – Quantità di memoria allocata nell'applicazione Flutter

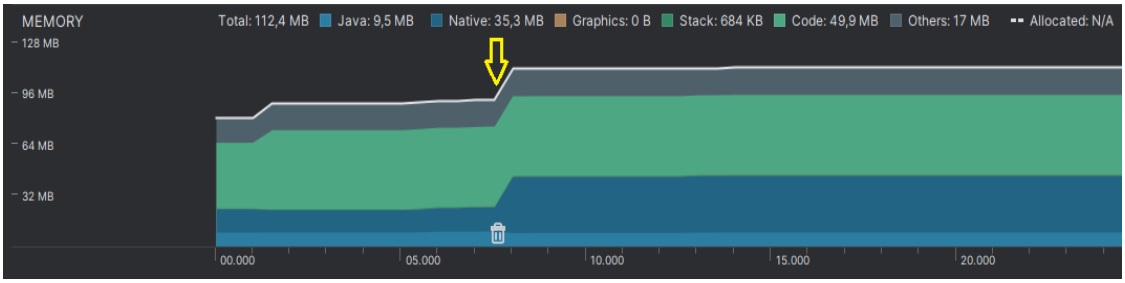

Figura 14 – Quantità di memoria allocata nell'applicazione Android nativa

Si può notare come nel caso dell'applicazione sviluppata con MAUI e con l'approccio nativo si verifichi un lieve incremento nell'utilizzo di RAM (segnalato in prossimità della freccia gialla) nel momento in cui si sceglie di aprire la pagina in cui viene poi renderizzata l'immagine.

La tabella e la figura 15 riepilogano i dati relativi alle dimensioni del file .apk e della memoria impiegata quando il sistema è a regime, nelle tre applicazioni.

Framework	RAM Allocata (in MB)	Dimensione file .apk (in MB)
.NET MAUI	179,7	32,0
Flutter	209,3	19,9
Nativa Android	112,4	10,0

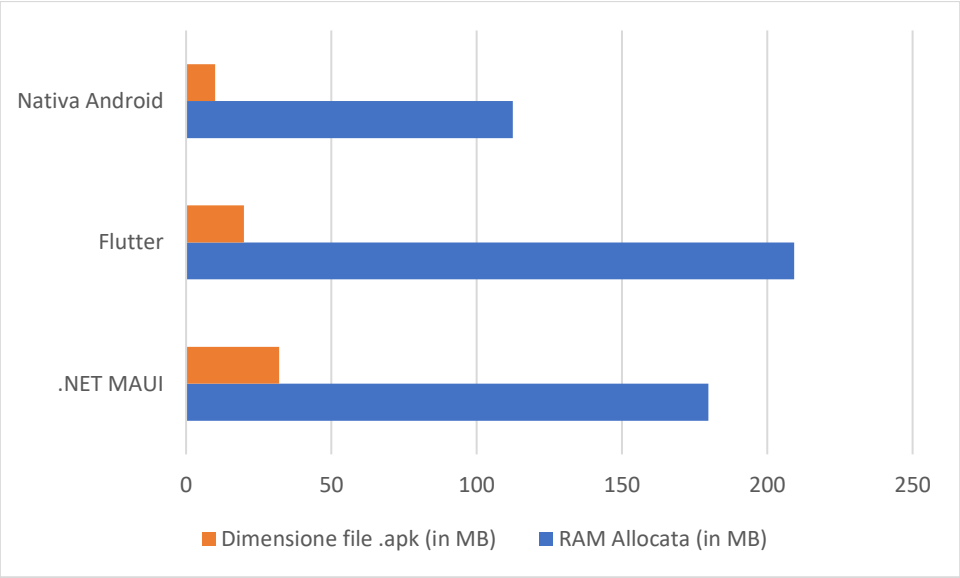

Figura 15 – Dimensioni del file .apk e della memoria impiegata nelle tre applicaizoni

I risultati ottenuti durante il test confermano le aspettative iniziali e forniscono una panoramica chiara delle prestazioni dei tre approcci di sviluppo.

Dal punto di vista della velocità di esecuzione, l'applicazione nativa si conferma come la più veloce in tutti i casi. Questo risultato era atteso poiché lo sviluppo nativo offre un accesso diretto alle risorse del dispositivo e una maggiore ottimizzazione del codice per la piattaforma specifica. Flutter, con la sua struttura di rendering e il suo motore Dart, si colloca al secondo posto in termini di velocità, mentre MAUI, essendo un framework multipiattaforma basato su .NET, risulta leggermente più lento.

Anche la dimensione del file APK segue la stessa tendenza. L'applicazione nativa tende ad avere un file APK più piccolo rispetto a quelle sviluppate con Flutter e MAUI. Questo è dovuto al fatto che lo sviluppo nativo consente di ottimizzare l'applicazione per la piattaforma specifica, evitando l'inclusione di librerie e risorse aggiuntive che potrebbero aumentare le dimensioni del file APK.

Dall'analisi della quantità di RAM impiegata durante l'esecuzione, emerge che l'applicazione nativa utilizza meno risorse di memoria rispetto a Flutter e MAUI. Anche in questo caso, ciò è attribuibile alla maggiore ottimizzazione e controllo offerti dallo sviluppo

nativo, che consente di gestire in modo più efficiente le risorse di sistema.

In sintesi, i risultati dei test confermano che lo sviluppo nativo offre prestazioni superiori in termini di velocità di esecuzione, dimensioni del file APK e utilizzo di memoria RAM. Questo evidenzia l'importanza di considerare le esigenze specifiche del progetto e le caratteristiche delle piattaforme di destinazione nella scelta dell'approccio di sviluppo più adatto. Sebbene Flutter e MAUI offrano vantaggi in termini di sviluppo multipiattaforma e produttività dello sviluppatore, lo sviluppo nativo rimane la scelta migliore per applicazioni che richiedono prestazioni ottimali e un utilizzo efficiente delle risorse di sistema.

Conclusioni

Il presente lavoro di tesi ha incluso lo sviluppo di un'applicazione Android con tre diversi approcci: sviluppo nativo, sviluppo con Flutter e sviluppo con .NET MAUI.

Lo scopo del test era valutare le prestazioni di queste tre modalità di sviluppo da diversi punti di vista: velocità di esecuzione, velocità nel rendering, dimensione del file APK generato e quantità di memoria RAM impiegata.

La motivazione alla base di questo test risiede nella necessità di comprendere quale approccio di sviluppo fornisca le migliori prestazioni in termini di velocità, efficienza delle risorse e dimensioni dell'applicazione. Questo è particolarmente importante perché le prestazioni dell'applicazione possono influenzare direttamente l'esperienza dell'utente finale, determinando la sua soddisfazione e la propensione all'utilizzo dell'applicazione stessa.

Per valutare le prestazioni cpu-bound è stato creato un carico di lavoro simulato all'interno dell'applicazione, consistente nell'addizione di un seno, un coseno e una radice cubica eseguiti su una variabile contatore per 2 milioni di volte. Questo carico di lavoro è stato ripetuto otto volte e ne è stato calcolato il tempo medio impiegato.

Per quanto riguarda le prestazioni su elementi grafici, è stato scelto di valutare il tempo che impiega un’immagine per renderizzarsi completamente. Questo carico di lavoro è stato eseguito 5 volte e per ogni iterazione l’applicazione è stata chiusa completamente.

Inoltre, durante l'esecuzione dell'applicazione, è stato utilizzato un profiler per monitorare

la quantità di memoria RAM impiegata dall'applicazione stessa.

I risultati ottenuti dal test forniscono importanti considerazioni dal punto di vista dello sviluppo software.

In primo luogo, dimostrano l'importanza di valutare attentamente le prestazioni durante la scelta dell'approccio di sviluppo. Sebbene framework come Flutter e MAUI offrano vantaggi in termini di produttività e sviluppo multipiattaforma, se prestazioni sono una priorità assoluta, lo sviluppo nativo può essere la scelta migliore, come confermato dai risultati del test che mostrano una maggiore velocità di esecuzione e un utilizzo più efficiente della memoria RAM.

D'altra parte, se il progetto non richiede particolari prestazioni o se è necessario un rapido sviluppo per più piattaforme, l'approccio cross-platform può risultare vantaggioso in termini di tempo e risorse.

Pertanto, la scelta tra sviluppo nativo e cross-platform deve essere guidata dalle esigenze specifiche del progetto, bilanciando le prestazioni desiderate con i vincoli di tempo e risorse disponibili.

È opportuno sottolineare che esistono delle limitazioni, relative alla generalità e all'equivalenza dei risultati dei test quando si confronta lo sviluppo nativo con i framework cross-platform quali ad esempio .NET MAUI e Flutter, intrinseche nel raggiungere un'equivalenza perfetta. Ogni approccio (Nativo, .NET MAUI, Flutter, ecc.), infatti, utilizza meccanismi interni differenti per il rendering e il threading, il che può introdurre variazioni minori nonostante gli sforzi per allinearli. Inoltre, i linguaggi di programmazione sottostanti utilizzati (ad esempio Swift/Kotlin/Java per il nativo, C# per .NET MAUI e Dart per Flutter) hanno i loro set di ottimizzazioni che possono influenzare le prestazioni.

Per quanto riguarda l'equivalenza tecnica, abbiamo seguito con attenzione la coerenza del codice, replicando la complessità algoritmica e la logica in tutte le implementazioni (Nativa, .NET MAUI e Flutter). Le stesse operazioni matematiche e di rendering sono state utilizzate in tutte le versioni dell'app per mantenere equo il confronto delle prestazioni. Inoltre, tutti i test sono stati eseguiti sullo stesso hardware, non in ambiente simulato e nelle stesse

condizioni per evitare discrepanze causate da differenze nell'ambiente di test. Le ottimizzazioni sono state esaminate per garantire che fossero comparabili tra le applicazioni native e quelle sviluppate con framework cross-platform, ad esempio, tutte le applicazioni sono state compilate in modalità release con ottimizzazioni attive.

Per quanto riguarda l'equivalenza funzionale, le applicazioni di esempio hanno implementato una funzionalità simile su tutte e tre le piattaforme per garantire che affrontassero carichi computazionali comparabili. Questo ha incluso gli stessi componenti dell'interfaccia utente, operazioni di backend e meccanismi di gestione dei dati. Inoltre le interfacce sono state progettate per essere quanto più simili possibile.

Inoltre, l'incremento delle capacità grafiche dei dispositivi moderni, in gran parte dovuto anche al diffondersi del fenomeno del gaming portatile, può effettivamente mitigare alcuni degli effetti dovuti alla poca generalità delle prove effettuate nel nostro studio. Consideriamo prima di tutto che i dispositivi moderni sono dotati di hardware sempre più potente, con GPU avanzate e CPU multicore che possono gestire carichi elevati di calcolo e rendering con maggiore efficienza.

Nel contesto delle prove CPU-intensive e di rendering che abbiamo condotto, la maggiore potenza delle GPU può alleggerire il carico della CPU nei compiti di rendering. Questo significa che, mentre oggi il rendering può ancora rappresentare un carico significativo per la CPU, in futuro la maggior parte di questo carico potrebbe spostarsi sulla GPU, rendendo meno rilevanti le differenze di prestazioni osservate tra le varie piattaforme (native e cross-platform).

Va considerato inoltre che i moderni framework di sviluppo stanno progressivamente migliorando nell'uso ottimale delle risorse hardware. Ad esempio, Flutter e .NET MAUI sono in continua evoluzione ed in futuro potrebbero incorporare tecnologie che sfruttano meglio le capacità grafiche avanzate dei dispositivi moderni.

Tuttavia, è importante riconoscere che non tutte le applicazioni beneficiano in egual misura dell'aumento delle capacità grafiche. Applicazioni che richiedono elaborazione intensiva di dati, algoritmi complessi o che si basano su operazioni di backend “pesanti” potrebbero non

trarre lo stesso vantaggio dalle GPU avanzate di conseguenza non viene eliminata completamente le necessità di considerare le diverse capacità delle CPU e le ottimizzazioni specifiche offerte dai diversi framework di sviluppo.

Le prestazioni dell'applicazione sono certamente un aspetto critico da considerare, specialmente se l'applicazione richiede elevati livelli di velocità ed efficienza nell'esecuzione delle operazioni. Tuttavia, oltre alle prestazioni, è altrettanto importante valutare altri aspetti quali la familiarità dello sviluppatore con l'approccio scelto.

Inoltre, la scalabilità dell'applicazione è un'altra considerazione critica, specialmente se si prevede una crescita significativa nel numero di utenti o nel volume di dati gestiti dall'applicazione nel tempo. Un'architettura scalabile e flessibile può consentire all'applicazione di adattarsi alle crescenti esigenze e di supportare un numero sempre maggiore di utenti e carichi di lavoro.

In conclusione, la scelta dell'approccio di sviluppo dipende dalle esigenze specifiche del progetto, con considerazioni importanti da fare riguardo alle prestazioni, alla produttività dello sviluppatore e alla scalabilità dell'applicazione. È importante valutare attentamente queste considerazioni per garantire il successo del progetto e soddisfare le aspettative degli utenti finali.

Bibliografia

[1] Allen, S., Graupera, V., & Lundrigan, L. (2010). The smartphone is the new PC. In Pro smartphone cross-platform development: iPhone, blackberry, windows mobile and android development and distribution. Apress.

[2] Alvarez, O. D. G., Riofrío, M. I. P., Cabezas, N. T. M., Alvarez, I. M. G., Joel, V. Ñ. E., Ariel, C. V. K., & Angel, C. C. L. (2023). Comparative Analysis of Cross-Platform Frameworks. Journal of Namibian Studies: History Politics Culture, 33, 2465-2486.

[3] Aurelius, M. (2020). Mobile Application Development: Framework with key criteria for choosing native or cross-platform application development.

[4] Bhagat, S. A., Dudhalkar, S. G., Kelapure, P. D., Kokare, A. S., & Bachwani, P. S. (2022). Review on Mobile Application Development Based on Flutter Platform. International Journal for Research in Applied Science and Engineering Technology, 10(1), 803-809.

[5] Biørn-Hansen, A., Rieger, C., Grønli, T. M., Majchrzak, T. A., & Ghinea, G. (2020). An empirical investigation of performance overhead in cross-platform mobile development frameworks. Empirical Software Engineering, 25, 2997-3040.

[6] Carius, L., Eichhorn, C., Plecher, D. A., & Klinker, G. (2022, March). Cloud-based cross-platform collaborative ar in flutter. In 2022 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW) (pp. 682-683). IEEE.

[7] Choi, Y., Yang, J.-s., & Jeong, J. (2009). Application Framework for Multi Platform Mobile Application Software Development. 2009 11th International Conference on Advanced Communication Technology, 208-213.

[8] Darji, V., Khan, A., & Dongaonkar, A. (2021). Mobile Application Development-

Native App. International Research Journal of Innovations in Engineering and Technology, 5(12), 91.

[9] Estdale, J., & Georgiadou, E. (2018). Applying the ISO/IEC 25010 quality models to software product. In Systems, Software and Services Process Improvement: 25th European Conference, EuroSPI 2018, Bilbao, Spain, September 5-7, 2018, Proceedings 25 (pp. 492-503). Springer International Publishing.

[10] Falco, M., & Robiolo, G. (2021, July). Product Quality Evaluation Method (PQEM): A Comprehensive Approach for the Software Product Life Cycle. In 7th International Conference on Software Engineering (SOFT).

[11] Fojtík, R., & Pawlas, J. (2023, May). Cross-Platform And Native Mobile App Development. In 4th International conference on Decision making for Small and Medium-Sized (p. 61). Silesian University in Opava, School of Business Administration in Karviná Page 4.

[12] Haider, A. (2021). Evaluation of cross-platform technology Flutter from the user's perspective.

[13] Holzinger, A., Treitler, P., & Slany, W. (2012). Making Apps Useable on Multiple Different Mobile Platforms: On Interoperability for Business Application Development on Smartphones. CD-ARES, (pp. 176-189). Prague

[14] Hvenfelt, L. (2023). Survey on the state of cross-platform mobile development frameworks.

[15] ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality requirements and Evaluation (SQuaRE) — System and software quality models - ISO/IEC 25010. (2011)

[16] Mannino, R. G., Arconada Alvarez, S. J., Greenleaf, M., Parsell, M., Mwalija, C., & Lam, W. A. (2023). Navigating the complexities of mobile medical app development from idea to launch, a guide for clinicians and biomedical researchers. BMC medicine, 21(1), 109.

[17] MAUI Docs. (2023). .Net Multi-platform App UI Documentation. Retrieved May 29,

2023, from Microsoft Learn - .NET MAUI: https://learn.microsoft.com/en-us/dotnet/maui

[18] Palmqvist, L. (2023). Evaluating. NET MAUI as a replacement for native Android mobile application development with focus on performance.

[19] Ponomarev, I. V. (2023). Features Of The. Net Maui Framework For Creating A Cross-Platform ApplicationS. System technologies, 1(144), 51-57.

[20] Raj, R. C., & Tolety, S. B. (2012). A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. 2012 Annual IEEE India Conference (INDICON), 625-629.

[21] Systä, K., & Sand, A. (2023). Experiments Comparing Cross-Platform And Native Mobile Applications.

[22] Ye, R. (2023). NET MAUI Cross-Platform Application Development: Leverage a first-class cross-platform UI framework to build native apps on multiple platforms.

[23] Zupancic, K. (2021). Next Steps–Continuity After Going Digital. Digital Transformation of the Laboratory: A Practical Guide to the Connected Lab, 277-286