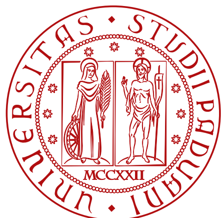


DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Framework per lo sviluppo di App Mobile Cross-Platform: un'esperienza reale con Flutter

Relatore
Prof. Mauro Migliardi

Laureando
Umberto Bianchin

ANNO ACCADEMICO 2023-2024

Data di laurea 15/07/2024

A mamma Stefania e papà Loris

Abstract

L’evoluzione tecnologica degli ultimi anni, in particolare nel campo dei dispositivi mobili come smartphone e tablet, ha portato ad un aumento significativo nello sviluppo di software e applicazioni mobili per i più disparati utilizzi.

Nell’ambito di questa trasformazione digitale, lo scopo di questa tesi è esplorare i principali framework per lo sviluppo di applicazioni cross-platform, con un focus particolare su Flutter. Flutter, introdotto da Google nel 2015 e giunto alla sua prima versione stabile nel 2018, si distingue come una soluzione innovativa per la creazione di app multipiattaforma utilizzando il linguaggio di programmazione Dart.

Sono state successivamente sviluppate e analizzate due applicazioni per il settore della ristorazione utilizzando Flutter.

Con l’obiettivo di fornire una comprensione approfondita delle potenzialità di Flutter e del suo ecosistema, questa tesi esamina dettagliatamente le caratteristiche, i vantaggi e le sfide legate allo sviluppo di applicazioni cross-platform. Attraverso l’analisi delle due applicazioni create, si discutono i principali aspetti tecnici e le migliori pratiche per lo sviluppo di app efficaci e performanti, sottolineando come Flutter possa rappresentare una soluzione ottimale per rispondere alle esigenze degli utenti.

Indice

Introduzione **1**

1 Architetture di sviluppo mobile **3**

1.1 Sviluppo nativo 4
1.2 Progressive Web Apps 5
1.3 Sviluppo cross-platform 6

2 Framework cross-platform principali **9**

2.1 React Native 9
2.2 Xamarin 10
2.3 Apache Cordova 11
2.4 Evoluzione dei Framework 13
2.4.1 Sviluppi Futuri 13

3 Flutter **15**

3.1 Introduzione a Dart 15
3.1.1 Piattaforme supportate 15
3.2 Introduzione a Flutter 16
3.2.1 Perché Flutter usa Dart 18

4 Architettura e strumenti di Flutter **19**

4.1 Architettura 19
4.2 Widget 20
4.3 Rendering 22
4.4 Test e Debugging in Flutter 23
4.5 Interazione con piattaforme native 23
4.5.1 Platform Channels 23

5 Sviluppo di un’applicazione in Flutter **25**
5.1 Panoramica delle applicazioni . . . 25
5.2 Scelta tecnologica . . . 26
5.3 Applicazione cliente . . . 27
5.4 Applicazione gestionale . . . 32
5.5 Gestione dello Stato e della Logica di Business . . . 34
5.6 Integrazione di Firebase per il backend e l’autenticazione . . . 36
5.6.1 Autenticazione con Firebase Authentication . . . 36
5.6.2 Backend con Firebase Firestore . . . 39
5.7 Aggiornamenti futuri . . . 44

6 Conclusioni **45**

Bibliografia **47**

Ringraziamenti **49**

Elenco delle figure

1.1 Download di app mondiali 2016-2023. Fonte [17] 3
1.2 Architettura di base di un'applicazione nativa. Fonte [9] 4
1.3 Architettura di base di una progressive web app. Fonte [9] 5
1.4 Architettura di base di un'applicazione sviluppata con Flutter. Fonte [9] 6

2.1 Architettura di base di un'applicazione sviluppata con React Native. Fonte [9] . 10
2.2 Architettura di un'applicazione sviluppata con Xamarin. Fonte [10] 11
2.3 Architettura di base di un'applicazione sviluppata con Cordova. Fonte [14] . . 12

3.1 Dart logo 15
3.2 Cross-platform mobile framework utilizzati in tutto il mondo dal 2019 al 2022. Fonte [7] 17

4.1 Layers architetturali in Flutter. Fonte [15] 20
4.2 Widget tree del codice 4.1 22
4.3 Sequenza degli step del rendering. Fonte [15] 22

5.1 Prima e seconda schermata principale dell'applicazione cliente 27
5.2 Terza e quarta schermata principale dell'applicazione cliente 28
5.3 Map Location Picker 31
5.4 Schermata di login 32
5.5 Schermata degli ordini 33
5.6 Schermata del menu 33
5.7 Schermata della mappa 34
5.8 Notifica ricevuta tramite Cloud Function 44

Introduzione

Secondo quanto riportato da Ansa nel 2020 [13], il numero di smartphone in Italia supera il numero di abitanti, con un totale di circa 80 milioni di dispositivi mobile rispetto ai 60 milioni di abitanti. Quasi 50 milioni di persone (84% della popolazione) risultano giornalmente attive su internet, e 35 milioni sono costantemente impegnate sui propri social network. Queste statistiche sottolineano come queste tecnologie si stiano sempre più affermando nelle vite quotidiane, rendendo quasi impossibile passare un'intera giornata senza cercare qualcosa su internet, aprire un'applicazione (app) social o solamente inviare un messaggio ad un amico.

Il presente elaborato di tesi tratta il tema dello sviluppo di applicazioni cross-platform, andando poi a focalizzare l'attenzione sul framework Flutter.

Il primo capitolo fornisce un'introduzione generale ai vari tipi di sviluppo di applicazioni mobili, analizzando lo sviluppo nativo, quello tramite progressive web app e infine lo sviluppo multipiattaforma. Ognuna di queste sezioni riporta un'analisi dei pro e contro di ogni modo di creare app.

Questa tesi continua poi con un'analisi approfondita dei principali framework di sviluppo cross-platform, includendo React Native, Xamarin e Apache Cordova. Inoltre, viene discussa l'evoluzione dei framework multipiattaforma e i possibili sviluppi futuri in questo campo.

Il terzo capitolo è dedicato a Flutter; inizia con una panoramica su Dart, il linguaggio di programmazione utilizzato dal framework, evidenziandone le caratteristiche principali e spiegando perché è stato scelto da Google. Vengono illustrate le piattaforme supportate da Flutter, mettendo in risalto la flessibilità e la potenza di questo strumento.

Successivamente, il quarto capitolo introduce Flutter in dettaglio, descrivendo la sua architettura e il modo in cui semplifica il processo di sviluppo. Viene spiegato come Flutter utilizza i widget per costruire l'interfaccia utente e come questi possano essere combinati e personalizzati per creare applicazioni altamente performanti e con interfacce anche complesse. Si prosegue con una sezione dedicata al testing e debugging, evidenziando gli strumenti integrati che questo framework mette a disposizione per gli sviluppatori e i vari tipi di test che consentono di mantenere alta la qualità del codice. Infine, viene trattata l'interazione con le piattaforme native tramite i Platform Channels, un meccanismo che permette di invocare codice nativo da Dart,

estendendo così le capacità e l'integrazione delle applicazioni Flutter.

L'ultimo capitolo presenta un caso di studio relativo allo sviluppo di due applicazioni in Flutter. Si inizia con un panoramica delle due app: la prima è destinata ai clienti e offre la possibilità di ordinare pizze, consultare il menu e accedere alle informazioni principali della pizzeria. La seconda applicazione, invece, è uno strumento gestionale per i ristoratori, che consente di modificare il menu e monitorare gli ordini ricevuti. Si prosegue descrivendo le funzionalità principali di queste applicazioni, per poi illustrare la scelta tecnologica che ha portato all'adozione di questo framework. Segue poi una descrizione dettaglia dell'applicazione cliente e di quella gestionale, con particolare attenzione alla logica di business. Infine viene discussa l'integrazione con Firebase per il backend e la gestione dell'autenticazione. Il capitolo si conclude con una panoramica sugli aggiornamenti futuri previsti per l'applicazione, delineando le funzionalità aggiuntive e le ottimizzazioni pianificate per migliorare ulteriormente l'esperienza utente.

Capitolo 1

Architetture di sviluppo mobile

Nell'ultimo decennio, l'evoluzione tecnologica dei telefoni ha visto una notevole trasformazione, passando da dispositivi usati principalmente per chiamare ed inviare messaggi, a veri e propri "computer tascabili", capaci di navigare su internet, riprodurre musica e video e gestire innumerevoli applicazioni per la vita quotidiana.

In seguito allo sviluppo degli smartphone, è cresciuto anche il settore della produzione di applicazioni mobile. Si pensi solamente che ormai esiste un'applicazione per qualsiasi cosa, dall'app social per restare sempre in contatto con gli amici, all'app che ricorda di bere, a quella che monitora gli allenamenti e così via.

Secondo quanto riportato da Statista, noto sito web tedesco di raccolta dati, il numero mondiale di download di applicazioni nel 2016 è stato di 140 miliardi, mentre nel 2023 è quasi raddoppiato, con un totale di 257 miliardi di download. Si può vedere l'andamento di questi anni in figura 1.1.

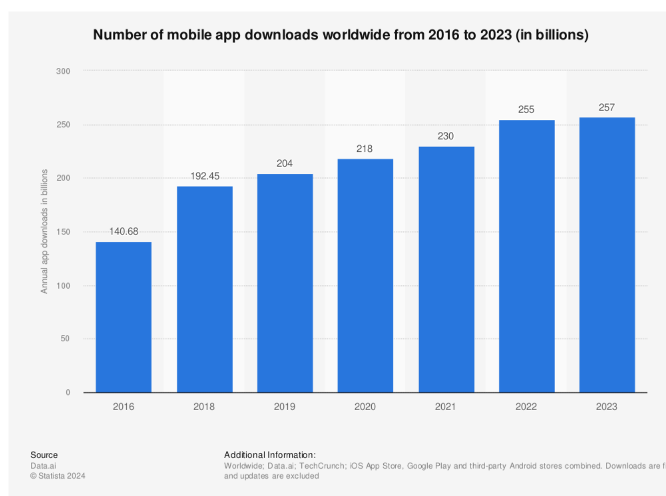


Figura 1.1: Download di app mondiali 2016-2023. Fonte [17]

A questa grande crescita è corrisposta anche un'evoluzione delle varie tecnologie per sviluppare queste app. Quasi tutte le applicazioni si possono riassumere in tre macro-gruppi:

- applicazioni native;
- progressive web apps;
- applicazioni cross-platform.

1.1 Sviluppo nativo

Le applicazioni native nascono per essere eseguite su un determinato sistema operativo, ad esempio Android piuttosto che iOS, e al momento comprendono la maggioranza delle app in circolazione. Sono scritte nel linguaggio specifico della piattaforma, come ad esempio Swift per iOS e Java per Android, e ciascuno di questi linguaggi fornisce librerie dedicate per poter utilizzare a pieno l'hardware e il software del dispositivo. L'approccio al rendering[1], così come la comunicazione di base con l'hardware del dispositivo delle app native è descritto in figura 1.2.

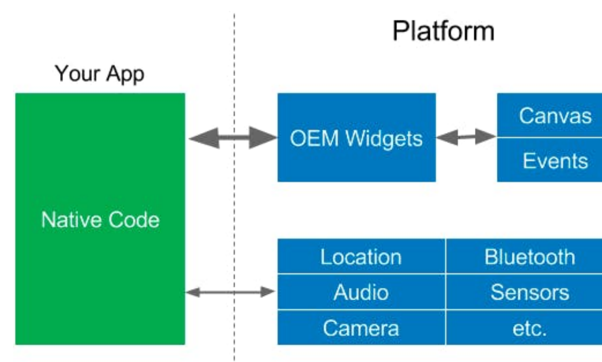


Figura 1.2: Architettura di base di un'applicazione nativa. Fonte [9]

Ci sono molti vantaggi nello sviluppare applicazioni in modo nativo:

- software veloce e reattivo;
- interfaccia coerente con quella del sistema operativo, che rende l'esperienza di utilizzo molto omogenea;
- minore necessità di hardware potenti.

[1]Letteralmente "restituzione grafica". Identifica il processo di generazione di un'immagine a partire da una descrizione matematica di una scena tridimensionale, interpretata da algoritmi che definiscono il colore in ogni punto dell'immagine digitale.

Questo sviluppo presenta però alcuni svantaggi:

- alto costo di manutenzione. Questo costo è dovuto al continuo aggiornamento dei vari sistemi operativi ed alle differenze tra questi, che possono portare al sorgere di problemi diversi che richiedono di volta in volta soluzioni altrettanto diverse (e quindi maggior lavoro per gli sviluppatori);
- mancanza di portabilità, in quanto gli sviluppatori devono realizzare un'applicazione per ogni piattaforma con la difficoltà di dover conoscere molti linguaggi di programmazione e tool di sviluppo;

1.2 Progressive Web Apps

Le così dette "progressive web apps", note anche come PWA, sono applicazioni che non devono essere scaricate sul dispositivo, ma possono essere usate tramite i browser di sistema; sono sviluppate principalmente usando HTML, CSS e JavaScript. Uno degli esempi più famosi è Twitter, che ha adottato le PWA nel 2017 permettendo così agli utenti di accedere al sito tramite un browser mobile. L'architettura di base delle PWA è definita in figura 1.3.

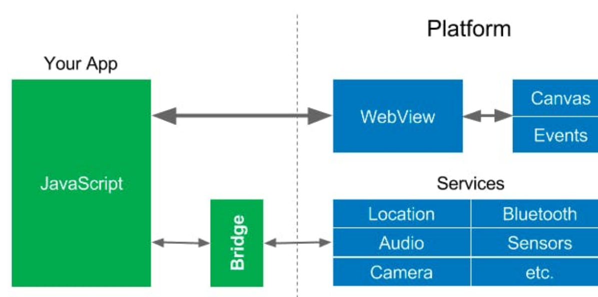


Figura 1.3: Architettura di base di una progressive web app. Fonte [9]

I vantaggi delle PWA sono:

- non occupare memoria nel cellulare o computer su cui vengono utilizzate;
- essere sempre aggiornate, in quanto ogni volta che vi si accede verrà caricata la versione più recente;
- sfruttare le misure di sicurezza integrate nei browser moderni, offrendo un livello di protezione costantemente aggiornato contro le vulnerabilità più recenti, senza che l'utente debba effettuare aggiornamenti manuali.

- usare il protocollo HTTPS (che incorpora SSL/TLS) per molte delle loro funzionalità, garantendo così che la comunicazione tra l'utente e il servizio sia criptata e autenticata.

Naturalmente ci sono anche lati negativi:

- scarsa visibilità di queste applicazioni, dovuta all'assenza di esse all'interno degli app store dei dispositivi;
- non avere accesso completo all'hardware del dispositivo, quindi non poter utilizzare determinati sensori come l'accelerometro o il giroscopio;
- pur essendo accessibili da ogni dispositivo, potrebbero non funzionare correttamente su smartphone più vecchi.

1.3 Sviluppo cross-platform

Infine, l'ultimo tipo di applicazioni sono quelle cross-platform (multipiattaforma). Questo approccio permette di sviluppare un'unica app che può essere eseguita su molteplici sistemi operativi, senza avere la necessità di essere riscritta o adattata ad ognuno di loro. Questo tipo di sviluppo è nato in seguito alla sempre più ampia offerta di dispositivi con sistemi operativi differenti, tra i principali Android e iOS per i dispositivi mobile e Windows, MacOS e Linux per i computer. L'architettura di base di un'applicazione sviluppata con un framework cross-platform (in questo caso Flutter) è definita in figura 1.4.

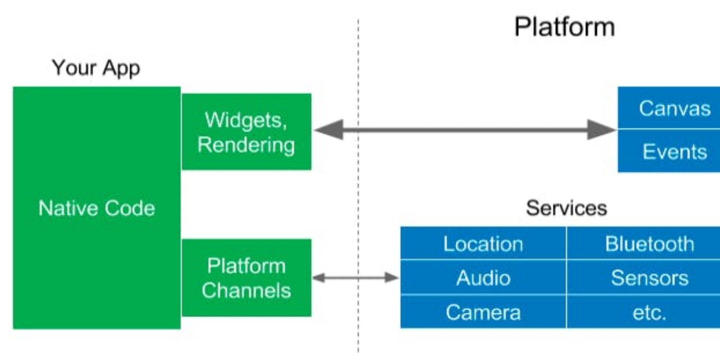


Figura 1.4: Architettura di base di un'applicazione sviluppata con Flutter. Fonte [9]

In questo caso i vantaggi sono:

- risparmio di tempo e risorse, dal momento che non è necessario avere versioni separate dell'app per ciascuna piattaforma;

- gli utenti possono godere di un'esperienza uniforme indipendentemente dal device che stanno usando;
- tempestività nel risolvere bug o implementare nuove funzioni, visto che una modifica al codice sorgente si riflette su tutti i dispositivi.

Ci sono però alcune sfide principali nello sviluppo di software multipiattaforma:

- rallentamento delle releases (pubblicazioni) dei vari aggiornamenti: quando Google o Apple rilasciano un aggiornamento importante per i vari sistemi operativi, gli sviluppatori potrebbero dover aggiornare l'app per poter utilizzare le ultime novità, ma prima di poterlo fare devono aspettare che il framework di sviluppo utilizzato venga aggiornato, e ciò potrebbe rallentare il lavoro.
- prestazioni più lente, poiché, al contrario di un'applicazione nativa che comunica direttamente con il dispositivo, i framework cross-platform necessitano di un cosiddetto bridge (ponte) per comunicare con il sistema operativo. Questo ponte è possibile grazie al cross-compiler (compilatore incrociato) che, partendo dal file sorgente, genera vari file binari nei linguaggi compatibili con i diversi sistemi operativi.

Lo sviluppo di applicazioni cross-platform si sta sempre più espandendo negli ultimi anni, grazie anche alla nascita di framework come Flutter e React Native (introdotto da Facebook) che rendono più efficace e veloce la scrittura del codice di queste applicazioni. Nel prossimo capitolo verranno elencati i principali framework per lo sviluppo cross-platform, analizzando i pro e i contro di ognuno di essi.

Capitolo 2

Framework cross-platform principali

In un'era digitale in continua evoluzione, la necessità di creare applicazioni mobile che possano funzionare senza problemi su diverse piattaforme ha portato alla nascita e allo sviluppo di vari framework cross-platform. Tra tutti quelli disponibili, oltre a Flutter, React Native, Xamarin e Apache Cordova sono risultati essere quelli più utilizzati dagli sviluppatori. Questo capitolo si propone di esplorare le caratteristiche, i vantaggi e le sfide associate a ciascuno di questi framework.

2.1 React Native

React Native è un framework software open-source sviluppato da Meta Platforms, Inc. (precedentemente Facebook, Inc.) e rilasciato nel 2015. Permette di sviluppare applicazioni mobile tramite l'uso di React, una libreria JavaScript utile per creare interfacce utente per dispositivi mobili [6].

Uno dei principali vantaggi nello sviluppare applicazioni con questo framework consiste nel bridge (figura 2.1): il bridge di React Native richiama le API[1] native per il rendering, in Objective-C per iOS o in Java per Android; in questo modo l'app viene visualizzata utilizzando i componenti nativi del sistema operativo, assomigliando a qualsiasi altra applicazione di quel dispositivo. React espone anche interfacce JavaScript per le API della piattaforma, in modo che l'applicazione possa accedere a funzionalità del dispositivo come la fotocamera o la geolocalizzazione.

Per fare tutto questo, l'applicazione deve eseguire su due threads differenti [8]: in uno viene eseguito la parte nativa mentre nell'altro il codice JavaScript. Tutte le comunicazioni tra queste

[1] Application Programming Interface. Sono un insieme di procedure che consentono allo sviluppatore di eseguire azioni predefinite senza dover capire in che modo farle.

avvengono attraverso il bridge e ciò può gravare sulle prestazioni del software, creando i cosiddetti *bottleneck* (collo di bottiglia), un fenomeno che si verifica quando le prestazioni di un sistema sono vincolate da un singolo componente. La principale sfida nello sviluppo con questo framework è quindi riuscire ad ottimizzazione le comunicazioni tra il codice JavaScript e la parte nativa, ad esempio minimizzando il trasferimento di dati e utilizzando soluzioni alternative come il ”code splitting” per caricare solo le parti del codice necessarie in un dato momento.

Negli ultimi anni, React Native ha visto anche l’introduzione di *Fabric*[2], un nuovo sistema di rendering che mira a rimpiazzare il bridge con un approccio più diretto e performante alla comunicazione tra il codice JavaScript e le API native. Fabric è progettato per ottimizzare la reattività e le prestazioni delle applicazioni React Native, riducendo l’overhead e migliorando l’efficienza della renderizzazione.

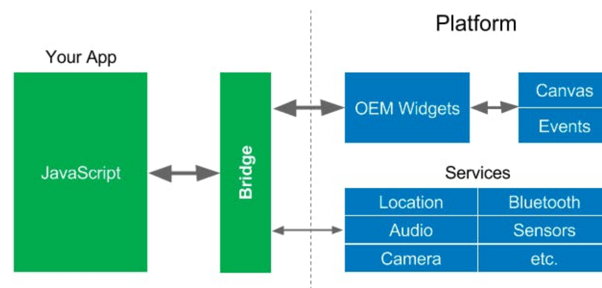


Figura 2.1: Architettura di base di un’applicazione sviluppata con React Native. Fonte [9]

2.2 Xamarin

Xamarin, creato dall’omonima azienda Xamarin nel 2011 (acquistata poi da Microsoft Corporation nel 2016), è un framework che permette lo sviluppo di applicazioni mobile cross-platform; utilizza C# e un’implementazione open source della piattaforma .NET, chiamata Mono [12]. Similmente a ciò che accade in React, anche in Xamarin le API native vengono avvolte in un wrapper[3] C# (figura 2.2).

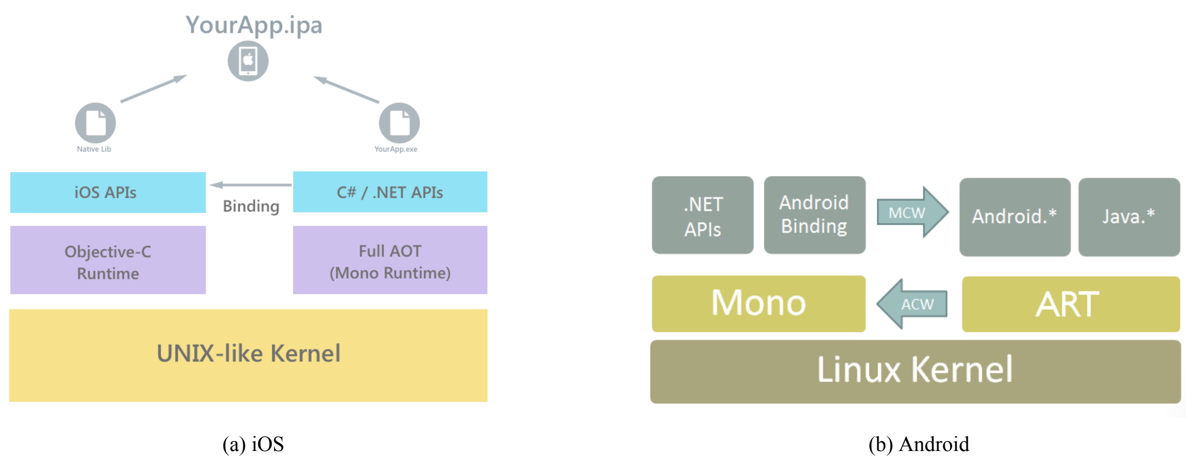
Gli svantaggi maggiori di questo framework sono legati alle prestazioni e all’uso della memoria: innanzitutto ci sono molte librerie runtime che devono essere distribuite o collegate all’applicazione e ciò influisce molto sulla taglia finale dell’app. In secondo luogo, soprattutto quando si parla di sviluppo per Android, devono essere allocati molti oggetti C# e Java per po-

[2] https://reactnative.dev/architecture/fabric-renderer

[3] Dal verbo inglese ”to wrap”, avvolgere. In informatica si tratta di un livello software che nasconde l’implementazione effettiva delle funzionalità e le presenta attraverso un’interfaccia ben definita.

ter raggiungere gli obiettivi prefissati e questo causa problemi di prestazioni e di allocazione della memoria.

Per affrontare queste questioni legate alle prestazioni e all'utilizzo della memoria, Xamarin ha introdotto varie ottimizzazioni. Una di queste è il Xamarin.Forms[4], un framework che permette la creazione di interfacce utente in modo dichiarativo e riutilizzabile su iOS, Android e Windows. Questo approccio non solo semplifica lo sviluppo di UI cross-platform ma aiuta a ridurre l'impatto delle dimensioni dell'applicazione, poiché consente la condivisione di una maggiore quantità di codice.



(a) iOS (b) Android

Figura 2.2: Architettura di un'applicazione sviluppata con Xamarin. Fonte [10]

2.3 Apache Cordova

Apache Cordova (all'inizio chiamato PhoneGap) è un framework per lo sviluppo di applicazioni ibride[5], creato da un'azienda canadese chiamata Nitobi Software nel 2009 e poi donato alla fondazione Apache Software Foundation [18]. Tra i vari framework citati è generalmente considerato quello meno performante, a causa del fatto che crea app ibride.

La principale innovazione portata da questo software consiste nella possibilità di accedere alle API e all'hardware (ad esempio la fotocamera) del dispositivo, contrariamente a quanto avviene nelle web application "basiche". Questo è possibile grazie ad una suite di API JavaScript implementate in due parti: una che espone le capacità native alla web application e l'altra che

[4] https://dotnet.microsoft.com/en-us/apps/xamarin/xamarin-forms

[5] Le applicazioni ibride sono sviluppate utilizzando principalmente tecnologie web (HTML, CSS e JavaScript), sono poi incapsulate in un contenitore nativo e accedute tramite una WebView

implementa la parte nativa. Una descrizione ad alto livello dell'architettura di un'applicazione sviluppata con Cordova può essere visionata in figura 2.3.

Come per gli altri framework, anche per Cordova ci sono degli svantaggi nello sviluppare applicazioni ibride. In primo luogo, molte funzionalità avanzate richiedono l'uso di plugin esterni per funzionare e nel caso uno di questi non venga più mantenuto dal suo sviluppatore o non sia più compatibile con una determinata versione del sistema operativo, sarà necessario cambiarlo o addirittura svilupparlo autonomamente; inoltre negli ultimi anni ha visto un calo nel supporto della comunità a favore di scelte più moderne. Un altro svantaggio consiste nell'esperienza utente, che potrebbe non essere fluida o coerente come nelle app native; infine, l'ultimo grosso problema sono le performance, sulle quali potrebbe influire la complessità dell'interfaccia grafica o la quantità di elaborazione dati richiesta.

Nonostante le critiche relative alle prestazioni e all'esperienza utente, Apache Cordova ha trovato una nicchia solida tra gli sviluppatori grazie alla sua flessibilità e al basso costo di ingresso per lo sviluppo di applicazioni mobile. Cordova offre un approccio vantaggioso per piccole aziende e start-up che desiderano raggiungere rapidamente il mercato con applicazioni che non richiedono elevate prestazioni o accesso intensivo all'hardware.

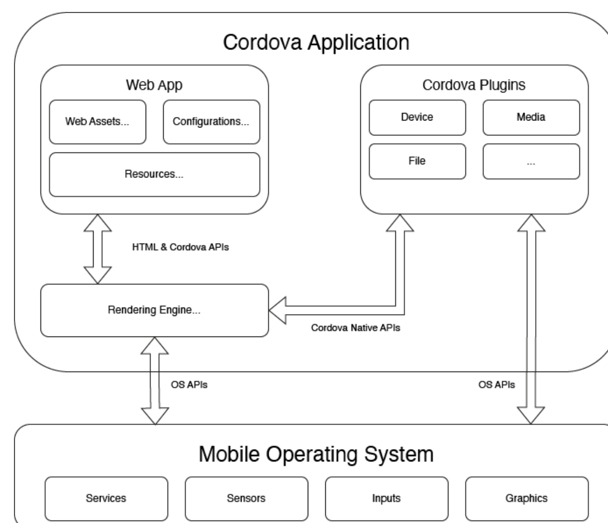


Figura 2.3: Architettura di base di un'applicazione sviluppata con Cordova. Fonte [14]

2.4 Evoluzione dei Framework

Ognuno dei framework sopra citati ha visto, negli ultimi anni, evoluzioni importanti e ha cercato di adattarsi agli standard tecnologici emergenti. React Native ha semplificato lo stato e il ciclo di vita dei componenti con *Hooks*[6] ed ha introdotto una nuova architettura per migliorare le prestazioni di rendering delle UI, *Fabric*[7], rendendolo ideale per progetti che beneficiano di rapide iterazioni di sviluppo. L'approccio basato sui componenti e il vasto ecosistema di librerie lo rendono particolarmente adatto per applicazioni che richiedono una forte personalizzazione dell'interfaccia utente e una performance fluida su tutte le piattaforme mobile.

Xamarin, con l'integrazione di .NET MAUI[8], si posiziona come una scelta robusta per sviluppatori .NET che cercano di estendere le loro applicazioni al mobile mantenendo un singolo stack tecnologico. La compatibilità con l'intero ecosistema .NET e la possibilità di condividere il codice non solo tra iOS e Android ma anche con Windows e MacOS lo rendono particolarmente adatto per progetti enterprise che necessitano di un'ampia distribuzione cross-platform.

Apache Cordova si adatta meglio a progetti con risorse limitate o a situazioni in cui è necessario portare rapidamente un prodotto sul mercato. La sua capacità di incapsulare codice web in un'app mobile lo rende ideale per progetti che possono sacrificare qualche aspetto delle performance per velocizzare lo sviluppo, specialmente quando già si dispone di un sito web funzionante che si desidera convertire in un'app mobile.

2.4.1 Sviluppi Futuri

Per quanto riguarda gli sviluppi futuri, React Native si sta muovendo verso una maggiore integrazione con le tecnologie web moderne, facilitando la transizione degli sviluppatori web al mobile e puntando su una maggiore efficienza nello sviluppo di app con performance quasi native. L'attenzione è focalizzata anche sull'ottimizzazione delle prestazioni attraverso la riduzione del tempo di caricamento delle app e l'incremento della fluidità delle interazioni utente.

Xamarin, attraverso .NET MAUI, punta a unificare ulteriormente lo sviluppo di applicazioni per tutte le piattaforme Microsoft, semplificando il processo di creazione di interfacce utente e migliorando l'accesso alle funzionalità native. L'obiettivo è di rendere lo sviluppo cross-platform ancora più efficiente, mantenendo al contempo alte prestazioni e una buona esperienza utente.

[6]https://react.dev/reference/react/hooks

[7]https://reactnative.dev/architecture/fabric-renderer

[8]https://learn.microsoft.com/it-it/dotnet/maui/what-is-maui?view=net-maui-8.0

Cordova sta esplorando modi per migliorare l'integrazione con altre tecnologie web e framework front-end, oltre a rafforzare la sicurezza e le performance. La sfida è mantenere la propria rilevanza in un panorama tecnologico in rapida evoluzione, focalizzandosi sull'ottimizzazione delle performance e sulla facilità di accesso alle funzionalità hardware dei dispositivi.

In conclusione, la scelta dei framework cross-platform dipende da una varietà di fattori, inclusi i requisiti del progetto, le competenze dei team di sviluppo, il budget disponibile e le aspettative in termini di performance ed esperienza utente. Ogni framework offre punti di forza unici e scenari di utilizzo ideali, e gli sviluppi futuri promettono di ampliare ulteriormente le loro capacità e applicabilità.

Capitolo 3

Flutter

3.1 Introduzione a Dart

Dart è un linguaggio di programmazione object oriented (orientato agli oggetti), una metodologia di programmazione che facilita la strutturazione del software come un insieme di entità denominate ”oggetti”. Ogni oggetto è una combinazione di dati, chiamati attributi, e di funzionalità sotto forma di funzioni o ”metodi”. Questo approccio consente agli sviluppatori di creare componenti software modulari e riutilizzabili.

Dart è stato progettato con un focus particolare per le applicazioni utente (viene infatti definito un linguaggio client-optimized), come quelle per dispositivi mobili e web. Offre prestazioni elevate e una progettazione dell'interfaccia grafica attentamente ottimizzata, garantendo così un'esperienza utente fluida e reattiva. Grazie alla sua capacità di compilazione in codice nativo per diverse piattaforme, Dart consente la creazione di applicazioni che possono essere eseguite in modo efficiente su un'ampia gamma di dispositivi. [11].

Figura 3.1: Dart logo

3.1.1 Piattaforme supportate

Dart è un linguaggio molto flessibile, infatti una volta che il codice è stato scritto può essere distribuito in diversi modi:

- compilazione Just In Time (JIT): nel contesto di Dart, durante la fase di sviluppo, il codice viene spesso eseguito in una Dart Virtual Machine (DVM) che utilizza la compilazione

JIT. Questo processo compila il codice sorgente in linguaggio macchina al momento dell'esecuzione, permettendo rapidi cicli di iterazione e la capacità di apportare e visualizzare modifiche in tempo reale attraverso la funzione di *hot reload*[1]. È un approccio efficiente in termini di sviluppo, perché non richiede un lungo processo di compilazione prima che le modifiche possano essere testate.

- compilazione Ahead Of Time (AOT): a differenza della JIT, la compilazione AOT avviene prima del lancio dell'applicazione. Dart consente di pre-compilare il codice in file binari eseguibili specifici per un dato sistema operativo. Questo processo produce un avvio più rapido e prestazioni migliorate poiché l'applicazione non deve compilare il codice al momento dell'esecuzione. La compilazione AOT è particolarmente utile per la creazione di applicazioni mobili con Flutter, dove le prestazioni e la rapidità di avvio sono essenziali.
- web: grazie al tool *dart2js* è possibile "convertire" un progetto Dart in codice JavaScript, rendendo possibile l'esecuzione dell'applicazione in browser diversi. Questo consente di mantenere coerenza nell'interfaccia utente su vari dispositivi, pur offrendo un'esperienza simile a quella di una Progressive Web App, benché le due tecnologie siano concettualmente diverse e abbiano architetture sottostanti distinte. Le PWA tendono ad essere più leggere e sono progettate per funzionare in modo più integrato con il browser e il sistema operativo dell'utente, beneficiando anche di capacità offline. Al contrario, un'applicazione per browser sviluppata in Dart è essenzialmente un'applicazione web che emula un'esperienza nativa.

3.2 Introduzione a Flutter

Tra tutte le numerose opzioni per lo sviluppo cross-platform, Flutter ha avuto la maggiore crescita in termini di adozione di massa. Google ha rilasciato la prima versione stabile di Flutter il 4 Dicembre 2018 [1], e al momento della scrittura di questo documento supporta la produzione di applicazioni per Android, iOS, Web, Windows, macOS e Linux. Come possiamo vedere dalla figura 3.2, nel 2022 Flutter ha superato React Native nella classifica dei framework più utilizzati per lo sviluppo multipiattaforma, con il 46% del market share.

Flutter si è rapidamente affermato come uno dei framework più popolari per lo sviluppo cross-platform, grazie alla sua capacità unica di compilare direttamente in codice nativo. Questo assicura prestazioni eccezionali e un'esperienza utente senza compromessi. Il suo ricco set di widget altamente personalizzabili consente di creare interfacce eleganti e reattive su qualsiasi

[1] https://docs.flutter.dev/tools/hot-reload

piattaforma. La caratteristica Hot Reload trasforma il processo di sviluppo, permettendo agli sviluppatori di apportare modifiche in tempo reale senza riavviare l'applicazione o perdere lo stato. Fortemente supportato da Google, Flutter inoltre vanta un ecosistema esteso e una comunità vivace, rendendolo una soluzione completa e all'avanguardia per la creazione di applicazioni mobili, web e desktop.

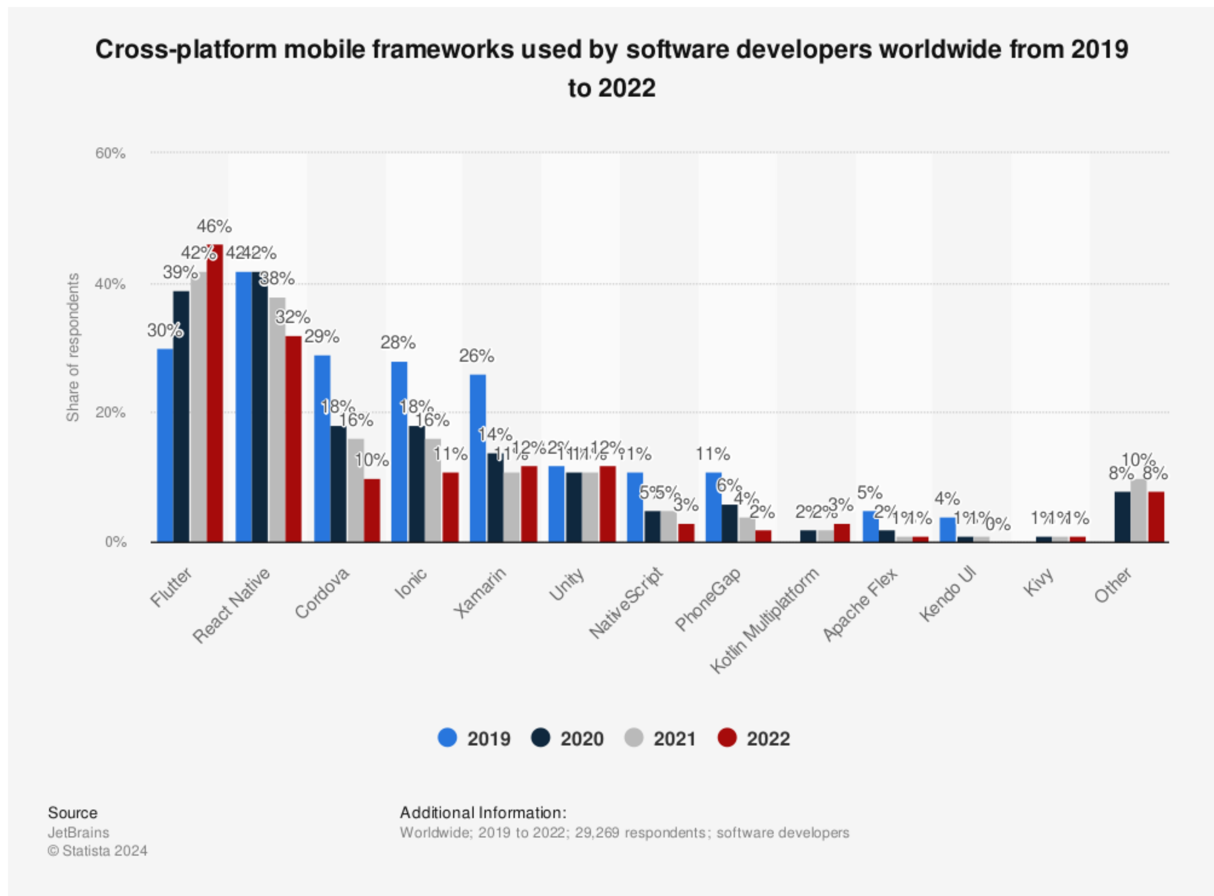


Figura 3.2: Cross-platform mobile framework utilizzati in tutto il mondo dal 2019 al 2022. Fonte [7]

Panoramica dei Framework Cross-Platform

- Ionic: un framework open-source per lo sviluppo di applicazioni mobile ibride che utilizza tecnologie web come HTML, CSS e JavaScript, con l'integrazione di Angular per la strutturazione dell'applicazione.

- Unity: piattaforma di sviluppo leader per creare giochi e esperienze interattive in 2D, 3D, realtà virtuale e realtà aumentata, che offre anche supporto per lo sviluppo cross-platform.

- NativeScript: un framework open-source che consente agli sviluppatori di utilizzare JavaScript, o linguaggi come TypeScript e Angular, per costruire applicazioni mobile native per iOS e Android.
- PhoneGap: era il predecessore di Apache Cordova, come spiegato nella sezione 2.3.
- Kotlin Multiplatform: una caratteristica del linguaggio di programmazione Kotlin che consente di condividere la logica del business tra le piattaforme mantenendo allo stesso tempo la possibilità di sviluppare interfacce utente native o specifiche per la piattaforma.
- Apache Flex (precedentemente Adobe Flex): un SDK (software development kit) principalmente orientato allo sviluppo di applicazioni internet ricche (Rich Internet Applications, RIA) per il web, e più recentemente anche per mobile, basato su Flash/AIR.
- Kendo UI: un insieme completo di componenti UI JavaScript, che include anche librerie per Angular, React, e Vue, progettati per velocizzare lo sviluppo web e permettere la costruzione di interfacce eleganti e funzionali.
- Kivy: un framework Python open-source per lo sviluppo di interfacce grafiche portabili tra diverse piattaforme. È noto per la sua semplicità e la capacità di eseguire su molteplici sistemi operativi, compresi Android e iOS.

3.2.1 Perché Flutter usa Dart

Il team di sviluppo di Flutter ha valutato numerosi linguaggi, ma alla fine ha deciso di usare Dart per diverse ragioni [4]. Alcune tra queste sono:

1. **prestazioni:** Dart garantisce alte e costanti prestazioni, in modo da evitare drop di frame durante l'esecuzione dell'applicazione. Inoltre, gestisce in modo efficiente la memoria, riuscendo a supportare anche allocazioni brevi e di piccole dimensioni;
2. **object-orientation:** dal momento che la maggior parte degli sviluppatori ha abilità nella programmazione ad oggetti, Dart sarebbe stato un linguaggio più semplice da imparare rispetto a dover imparare un modo di programmare completamente diverso;
3. **produttività:** Flutter permette agli sviluppatori di scrivere codice per molteplici piattaforme mantenendo le stesse prestazioni e aspetto in ognuna di esse, quindi l'utilizzo di un linguaggio ad alta produttività come Dart accelera il processo di scrittura;
4. Sia Flutter che Dart sono sviluppati da Google, che può liberamente decidere cosa aggiornare, rimuovere e migliorare in base alle preferenze della community.

Capitolo 4

Architettura e strumenti di Flutter

Flutter è un framework open-source per sviluppare applicazioni compilate in modo nativo per dispositivi mobili, computer e web partendo dalla scrittura di codice in un solo linguaggio, Dart. Al contrario delle app native, quelle cross-platform non possono fare affidamento diretto sugli OEM widget[1] o sulle API native, dal momento che questi si differenziano in base al sistema operativo e sarebbe impossibile scrivere un unico codice che sfrutti queste funzionalità tenendo conto delle differenze.

4.1 Architettura

Come affermato dal Flutter Team, «The goal is to enable developers to deliver high-performance apps that feel natural on different platforms, embracing differences where they exist while sharing as much code as possible.».

Dalla figura 4.1 notiamo come Flutter è composto da una serie di librerie indipendenti tra loro, le quali dipendono dallo strato sottostante. Il livello più basso è composto dall'*embedder*, che è specifico della piattaforma; ciò significa che, in base al sistema operativo, verrà utilizzato quello scritto nel linguaggio specifico per un certo dispositivo. L'embedder si occupa di coordinare le operazioni a basso livello, come ad esempio renderizzare superfici o gestire input. In questo modo il sistema operativo sottostante percepisce l'applicazione scritta usando Flutter come una qualsiasi applicazione nativa.

Il layer superiore all'embedder è costituito dall'*engine* (motore), il cuore di Flutter; prevalentemente scritto in C/C++, supporta tutte le primitive necessarie alle applicazioni per essere

[1] Original Equipment Manufacturer. I così detti OEM widget sono i componenti grafici previsti dal sistema operativo sottostante che permetto di disegnare l'interfaccia utente e di rispondere ad eventi come il tocco dello schermo con le dita.

eseguite. Fornisce l'implementazione a basso livello delle principali API, per il rendering della grafica, file e rete I/O e text layout.

Nello strato più esterno si trova il *framework*, scritto in Dart, ed è ciò con cui interagisce direttamente lo sviluppatore. Nelle prossime sezioni si andranno ad analizzare alcuni tra i più rilevanti componenti di questo framework.

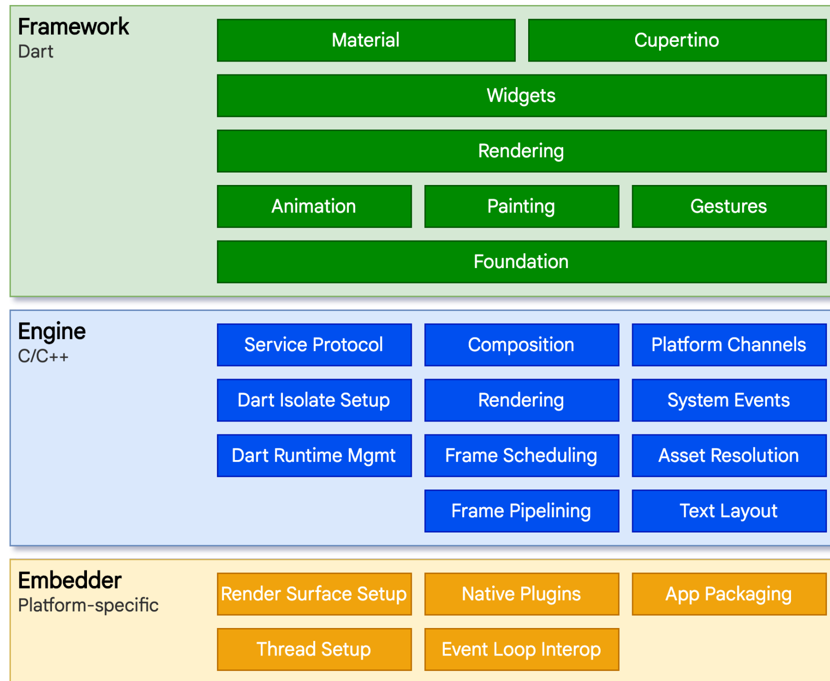


Figura 4.1: Layers architetturali in Flutter. Fonte [15]

4.2 Widget

In Flutter, ogni cosa che appare sullo schermo viene chiamata "widget", poiché tecnicamente è un discendente della classe Widget[2]. Più praticamente, un widget è una descrizione immutabile di una parte dell'interfaccia utente e tramite la composizione di quelli più piccoli e semplici si possono ottenere schermate anche molto complesse. Quando si vanno a comporre questi oggetti si ottiene un widget tree (albero di widget), in cui ci sono dei genitori e dei figli; quest'ultimi possono ereditare proprietà dai loro genitori, come ad esempio la grandezza all'interno dello schermo. È possibile vedere un esempio di widget tree nel codice sottostante e in figura 4.2: *root* rappresenta il livello più alto che contiene tutta l'applicazione, ed è gestito dal framework stesso,

[2]https://api.flutter.dev/flutter/widgets/Widget-class.html

poi si trova *MyApp*, che estende StatelessWidget[3] (widget immutabile, senza stato interno che cambia) e agisce da punto di ingresso dell'applicazione. Dopo c'è *Material App* che implementa il design Material[4] e funge da contenitore per l'app, *Scaffold* definisce un layout di base per la schermata, mentre *Column* è un widget che dispone i suoi figli in verticale, in questo caso due widget *Text* che visualizzano le due stringhe.

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

void main() {
  runApp(const MyApp());
}

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

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      title: 'Flutter Demo',
      home: Scaffold(
        body: Column(
          children: [
            Text("Questa è una demo"),
            Text("Per mostrare un esempio di widget tree"),
        ]),
    ));
}}
```

Codice 4.1: Esempio di widget tree in Dart

[3]https://api.flutter.dev/flutter/widgets/StatelessWidget-class.html

[4]https://docs.flutter.dev/ui/design/material

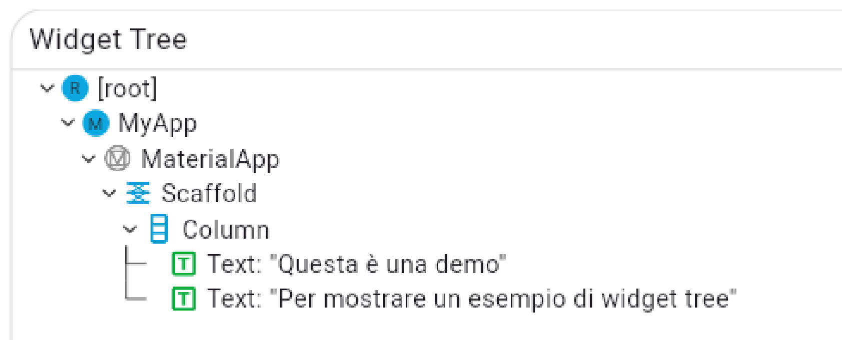


Figura 4.2: Widget tree del codice 4.1

4.3 Rendering

Per quanto riguarda il rendering, ossia gli step con cui una gerarchia di widget viene convertita negli attuali pixel disegnati sullo schermo, Flutter utilizza un approccio completamente diverso (vedi figura 1.4) rispetto a quello che viene usato, ad esempio, in React:

Flutter cerca di minimizzare l'astrazione che solitamente si crea tra le librerie native del sistema operativo, utilizzando il suo set di widget piuttosto che gli OEM widget. Il codice Dart è compilato in codice nativo, che usa un proprio motore di rendering molto efficiente, chiamato *Skia*[5], per disegnare l'interfaccia utente. Così facendo, lo sviluppatore ha pieno controllo su ogni singolo pixel dello schermo, e ogni input o evento viene gestito direttamente dal codice dell'app. Il principio su cui si è basata questa scelta è «Simple is fast», e da questo è dovuto anche la sequenza di step che traducono un input da parte dell'utente in istruzioni per la GPU (Figura 4.3).

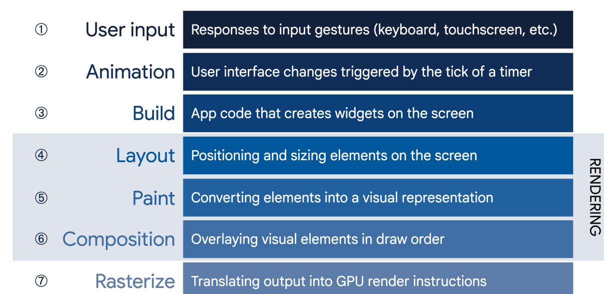


Figura 4.3: Sequenza degli step del rendering. Fonte [15]

[5]Dalla versione 3.10 di Flutter, il motore di rendering predefinito per iOS è diventato *Impeller*, mentre deve ancora essere rilasciato per Android.

4.4 Test e Debugging in Flutter

Il framework Flutter offre un solido supporto per il test e il debugging[3], consentendo agli sviluppatori di creare applicazioni affidabili e prive di errori. Flutter integra diversi tipi di test: unit test per la logica di business, widget test per testare singoli widget, e integration test per testare l'applicazione completa su un dispositivo reale o un emulatore.

Per i test unitari e di widget, Flutter fornisce il pacchetto ”flutter_test”, che include la classe ”WidgetTester” per interagire con i widget nell'ambiente di test. Questo permette di verificare le proprietà dei widget e simulare gli input dell'utente.

I test di integrazione in Flutter, gestiti dal pacchetto ”integration_test”, sono cruciali per verificare il comportamento complessivo dell'applicazione, inclusi il flusso utente e l'interazione con i servizi esterni. Flutter supporta l'automazione dei test di integrazione, che possono essere eseguiti su dispositivi fisici o su simulatori, garantendo che l'app funzioni correttamente in scenari d'uso reali.

Per il debugging, Flutter si avvale di *Dart DevTools*[2], una suite di strumenti di performance e debugging per applicazioni Dart e Flutter. Dart DevTools offre funzionalità come l'ispezione dell'albero dei widget, il profiling delle prestazioni, la visualizzazione della timeline degli eventi e l'analisi della memoria. Questi strumenti sono essenziali per identificare e risolvere problemi di prestazione o bug nell'applicazione.

4.5 Interazione con piattaforme native

L'interazione con le piattaforme native è una delle caratteristiche distintive di Flutter che lo rende un framework potente per lo sviluppo cross-platform. Sebbene Flutter permetta agli sviluppatori di costruire app ricche di funzionalità con un unico codice in Dart, ci sono scenari in cui è necessario accedere direttamente alle API native del sistema operativo per utilizzare funzionalità specifiche della piattaforma non disponibili direttamente nel framework. Questo può includere l'accesso a sensori hardware specifici, l'interazione con il software di sistema, o l'utilizzo di SDK di terze parti che richiedono chiamate native.

4.5.1 Platform Channels

Flutter affronta questa sfida attraverso l'uso dei *Platform Channels* (lett. canali di piattaforma)[5], che permettono la comunicazione bidirezionale tra il codice Dart e il codice nativo della piattaforma. Utilizzando questo meccanismo, gli sviluppatori possono implementare funzionalità native specifiche della piattaforma in linguaggi come Swift o Objective-C per iOS e Kotlin o Java per Android, e poi richiamare queste funzionalità dal codice Dart.

I Platform Channels si basano su un modello di messaggistica asincrona, utilizzando messaggi passati tra il lato client (codice Dart) e il lato host (codice nativo della piattaforma). Questi messaggi sono serializzati e deserializzati dal framework per essere trasmessi attraverso il canale, permettendo così la comunicazione tra i due ambienti di esecuzione. Attraverso questo meccanismo, un'app Flutter può richiedere dati da sensori hardware, accedere a servizi di localizzazione, utilizzare API di autenticazione specifiche della piattaforma e molto altro, superando così i limiti del framework pur mantenendo un unico codice di base.

Capitolo 5

Sviluppo di un’applicazione in Flutter

In questo capitolo verrà illustrata la creazione di applicazioni mobili dedicate alla digitalizzazione e ottimizzazione dei servizi di ristorazione, con un focus specifico su una pizzeria. In quest’epoca dominata dalla tecnologia, in cui il desiderio di comodità ed efficienza da parte dei consumatori non ha precedenti, sviluppare strumenti digitali che rispondano a tali esigenze è diventato cruciale per il successo di qualsiasi attività commerciale, in particolare nel settore della ristorazione.

5.1 Panoramica delle applicazioni

Ci si concentrerà sullo sviluppo di due applicazioni distinte ma complementari, entrambe realizzate con il framework Flutter, scelto per la sua capacità di fornire un’esperienza utente fluida e coesa su diversi dispositivi e piattaforme. La prima applicazione è rivolta ai clienti della pizzeria, permettendo loro di consultare il menu, effettuare ordini, e accedere a informazioni essenziali sul locale in modo semplice e intuitivo. Questa app mira a migliorare l’esperienza del cliente, rendendo il processo di ordinazione più veloce, piacevole e senza intoppi.

La seconda applicazione è progettata per il lato gestionale della pizzeria, offrendo agli amministratori e al personale un potente strumento per la gestione del menu, la visualizzazione e la conferma degli ordini, e la supervisione delle operazioni quotidiane. Questa app rappresenta un elemento fondamentale per migliorare l’efficienza operativa, consentendo al personale di gestire le richieste dei clienti in maniera più efficace e di adattarsi rapidamente a eventuali cambiamenti di menu o di preferenze dei consumatori.

Entrambe le applicazioni sono il risultato di un’attenta analisi delle esigenze sia dei clienti che dei gestori della pizzeria, e sono state sviluppate con l’obiettivo di massimizzare la soddisfazione del cliente e ottimizzare i flussi di lavoro interni. Attraverso questo approccio, si

intende non solo migliorare l’esperienza complessiva offerta dalla pizzeria, ma anche esplorare le potenzialità delle tecnologie mobili nel rinnovare e valorizzare il settore della ristorazione.

Il presente capitolo illustra in dettaglio le fasi di sviluppo di entrambe le applicazioni, dalla concezione alla realizzazione, evidenziando le scelte progettuali, le sfide incontrate e le soluzioni adottate per superarle. Attraverso questo esame approfondito, si vuole fornire un contributo significativo al campo dello sviluppo di software cross-platform, dimostrando come l’innovazione tecnologica possa trasformare positivamente le dinamiche di interazione tra le imprese e i loro clienti.

5.2 Scelta tecnologica

La scelta del framework giusto è cruciale per il successo di qualsiasi progetto software, soprattutto nel contesto dell’evoluzione rapida delle tecnologie mobili. La decisione di adottare Flutter per lo sviluppo delle due applicazioni è stata basata su un’attenta analisi dei requisiti che le due app devono avere, come la necessità di fornire un’esperienza utente superiore, mantenere l’efficienza dello sviluppo e garantire la flessibilità dell’applicazione su diverse piattaforme.

- Performance native-like: tramite Flutter è possibile sviluppare applicazioni quasi indistinguibili da quelle native, grazie alla sua capacità di compilare il codice Dart in codice macchina nativo per dispositivi mobili, web e desktop. Questa caratteristica garantisce un’esperienza utente fluida e reattiva su tutte le diverse piattaforme.

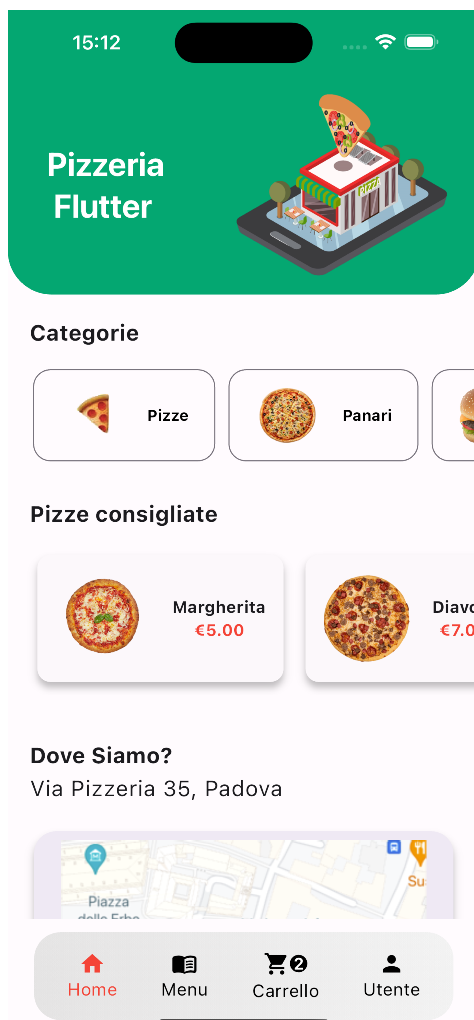
- Sviluppo rapido e iterativo: grazie alla funzione di hot reload di Flutter è possibile vedere in tempo reale le modifiche apportate al codice. L’hot reload funziona inserendo il codice sorgente aggiornato nella DVM in esecuzione, e, una volta che le classi vengono aggiornate con le nuove versioni, il framework ri-costruisce automaticamente il widget-tree, permetto allo sviluppatore di vedere immediatamente i cambiamenti effettuati. Questa funzionalità è molto più veloce di un restart completo, perché non perde lo stato dell’applicazione e non ricompila il codice nativo.

- UI consistente e personalizzabile: uno degli aspetti più critici delle applicazioni è l’interfaccia utente, che deve in primo luogo rispecchiare l’identità del proprietario (in questo caso la pizzeria), e deve poi fornire un’esperienza utente semplice ma allo stesso tempo immersiva e fluida. Con il suo ricco set di widget personalizzabili e con la sua potente engine di rendering, Flutter offre una flessibilità senza precedenti nella progettazione di UI ricche e visivamente accattivanti.

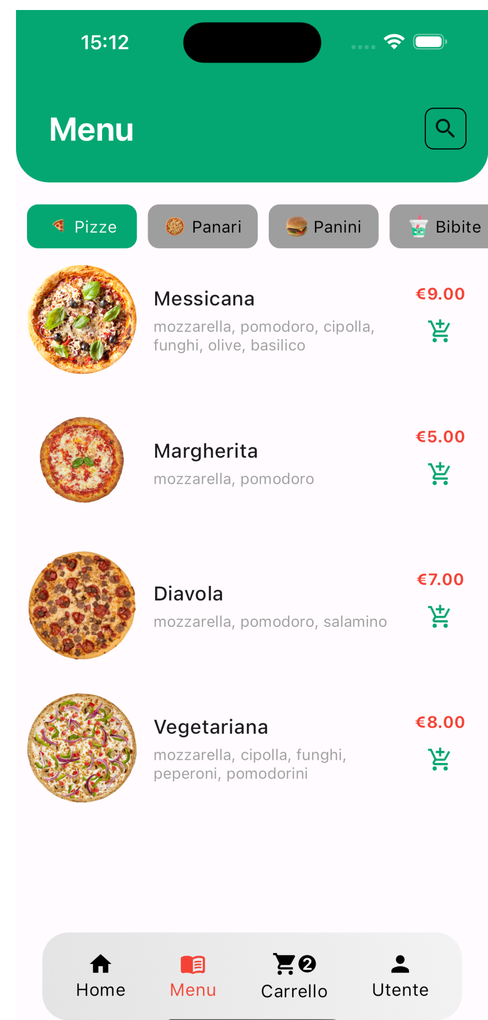
5.3 Applicazione cliente

L'app cliente è composta principalmente da 4 schermate: la schermata principale (figura 5.1a), dove è possibile vedere le varie categorie (Pizze, Panari, Panini e Bibite), le pizze consigliate e la posizione della pizzeria, grazie ad una API di Google Maps. È presente poi la schermata del menu (figura 5.1b), in cui è possibile consultare le varie pizze, selezionarle ed aggiungerle al carrello; successivamente si trova la sezione dedicata al carrello (figura 5.2a), con un riepilogo degli elementi selezionati e il costo totale dell'ordine ed infine c'è la pagina dedicata all'account personale (figura 5.2b), in cui è possibile inserire le proprie informazioni personali.

Questa applicazione è stata sviluppata principalmente per dispositivi mobili, ma si adatta perfettamente anche all'uso su un computer o un tablet.

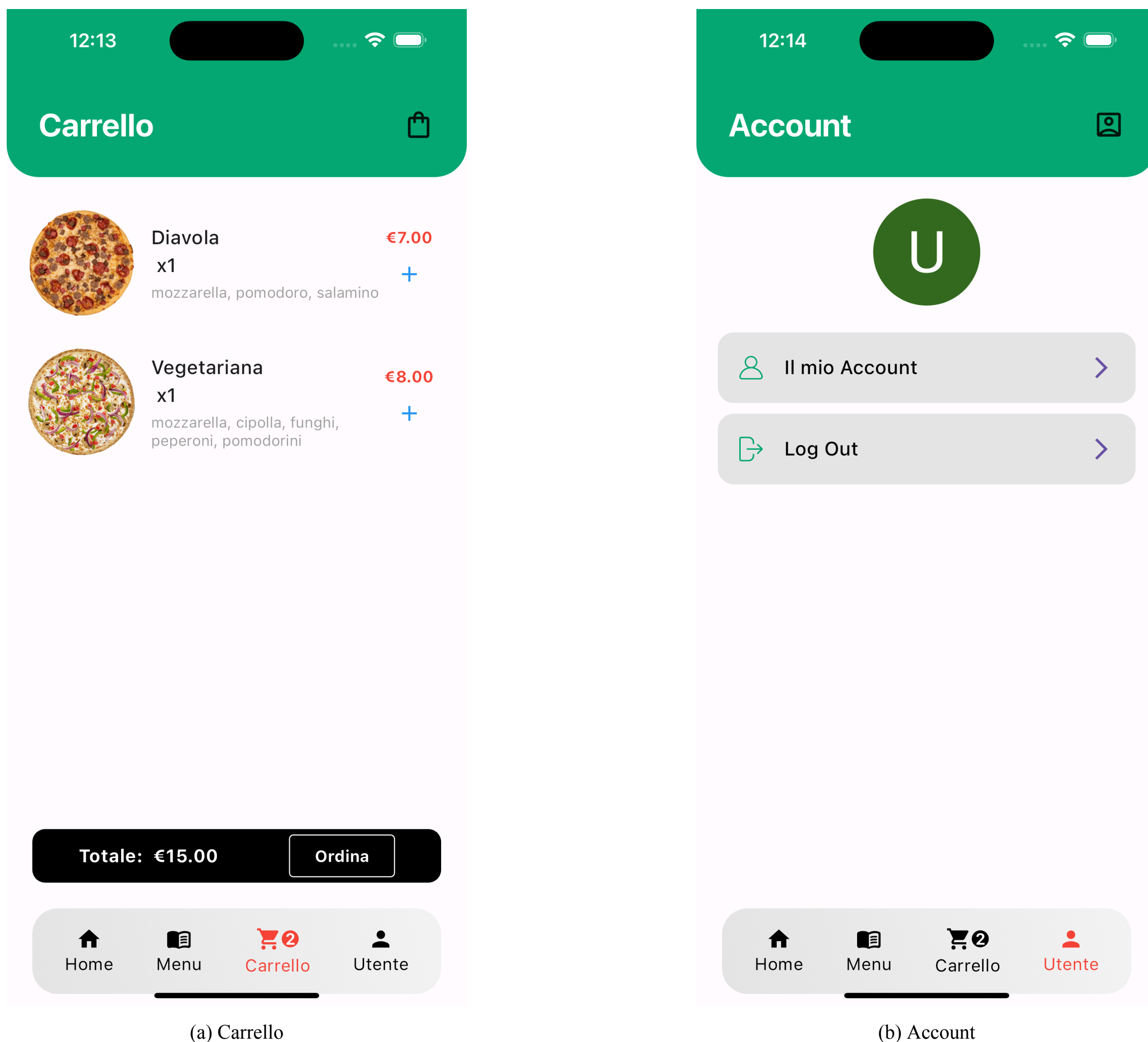


(a) Home



(b) Menu

Figura 5.1: Prima e seconda schermata principale dell'applicazione cliente



(a) Carrello

(b) Account

Figura 5.2: Terza e quarta schermata principale dell'applicazione cliente

Tutte le schermate sono state create utilizzando widget di Flutter ed alcuni plugin esterni, come il *Map Location Picker*[1], utilizzato nel codice 5.1 che si occupa di generare la schermata 5.3. Questo tool serve all'utente per selezionare l'indirizzo al quale, nel caso scelga la modalità di consegna a domicilio in fase di ordinazione, deve essere consegnato l'ordine. La selezione dell'indirizzo è raggiungibile dalla pagina "Account" e poi "Il mio Account".

```
1 // Classe per generare la schermata di selezione dell'indirizzo utente
2
3 // Importazione dei pacchetti Flutter e del plugin Map Location Picker
4 import 'package:flutter/material.dart';
```

[1]https://pub.dev/packages/map_location_picker

```
import 'package:map_location_picker/map_location_picker.dart';

// Definizione delle coordinate iniziali per la mappa
const double klatitude = 45.40799448013583;
const double klongitude = 11.893889699335174;

// Definizione del widget LocationPicker che estende StatefulWidget
class LocationPicker extends StatefulWidget {
  // Costruttore del widget che accetta una funzione callback onChanged
  const LocationPicker(this.onChanged, {super.key});
  final ValueChanged<String> onChanged;

  @override
  State<LocationPicker> createState() => _LocationPickerState();
}

// Stato associato al widget LocationPicker
class _LocationPickerState extends State<LocationPicker> {
  // Variabili per tenere traccia dell'indirizzo selezionato e
  // del luogo auto-completato
  String address = "null";
  String autocompletePlace = "null";

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Theme.of(context).primaryColor,
        toolbarHeight: 0,
      ),
      // Corpo principale del widget dentro SafeArea per rispettare
      // le aree sicure dello schermo
      body: SafeArea(
        child: Center(
          child: MapLocationPicker(
            // Setting e personalizzazione del widget
```

```
41              hideMapTypeButton: true,
42              hideLocationButton: true,
43              topCardColor: Colors.lightBlue[200]!.withOpacity(0.9),
44              bottomCardColor:
                  ↪  Theme.of(context).primaryColor.withOpacity(0.9),
45              language: "it",
46              region: "it",
47              searchHintText: "Cerca il tuo indirizzo",
48              apiKey: "API GOOGLE MAPS",
49              popOnNextButtonTaped: true,
50              bottomCardIcon: const Icon(
51                Icons.send,
52                color: Colors.black,
53              ),
54              // Imposta le coordinate iniziali della mappa
55              currentLatLng: const LatLng(klatitude, klongitude),
56              // Funzione chiamata quando l'utente seleziona "Avanti" nel
                  ↪  picker
57              onNext: (GeocodingResult? result) {
58                if (result != null) {
59                  setState(() {
60                    // Aggiorna l'indirizzo selezionato con il risultato
                        ↪  della geocodifica
61                    address = result.formattedAddress ?? "";
62                    // Invoca la callback onChanged per notificare
                        ↪  l'indirizzo selezionato
63                    widget.onChanged(address);
64                  });
65                }
66              },
67              hideMoreOptions: true,
68              // Gestione della selezione di un suggerimento di ricerca
69              onSuggestionSelected: (PlacesDetailsResponse? result) {
70                if (result != null) {
71                  setState(() {
```

```
72                    // Aggiorna il luogo auto-completato con il risultato
                      ↪  selezionato
73                    autocompletePlace = result.result.formattedAddress ??
                      ↪  "";
74                  });
75              }}),
76      )));
77  }}
```

Codice 5.1: Classe per la selezione dell'indirizzo

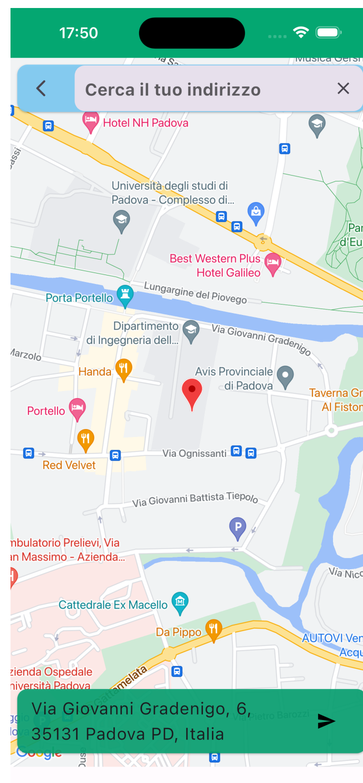


Figura 5.3: Map Location Picker

Tutti i dati relativi al menu (completamente modificabili dall'applicazione gestionale) e agli utenti sono conservati nel database di Firebase[2], approfondito nella sezione 5.6

[2]https://firebase.google.com/

5.4 Applicazione gestionale

L'applicazione gestionale è stata sviluppata per permettere ai gestori della pizzeria di poter controllare in autonomia tutto ciò che riguarda l'attività, come il menu, l'orario di apertura (per permettere ai clienti di poter ordinare solo per un certo lasso di tempo) e gli ordini. È composta da quattro schermate principali: la prima è la schermata di login (figura 5.4), che permette di accedere solo agli utenti che hanno il ruolo admin. Una volta effettuato l'accesso si arriva nella pagina della visione e gestione ordini (figura 5.5), in cui l'admin può visionare e modificare l'ordine e confermarlo o rifiutarlo. Un'altra schermata molto importante è quella della mappa (figura 5.7), dalla quale si ha una visione immediata degli ordini a domicilio, per un'organizzazione degli orari delle consegne più efficiente; infine si trova la schermata dedicata alla gestione del menu della pizzeria (figura 5.6), dalla quale è possibile modificare ingredienti, pizze e altri prodotti.

Questa applicazione è stata sviluppata principalmente per dispositivi con uno schermo grande e da usare in modalità landscape (orizzontale), come tablet o computer, ma grazie alla flessibilità di Flutter (e con qualche accorgimento) potrebbe essere eseguita anche su dispositivi mobili con uno schermo più piccolo.

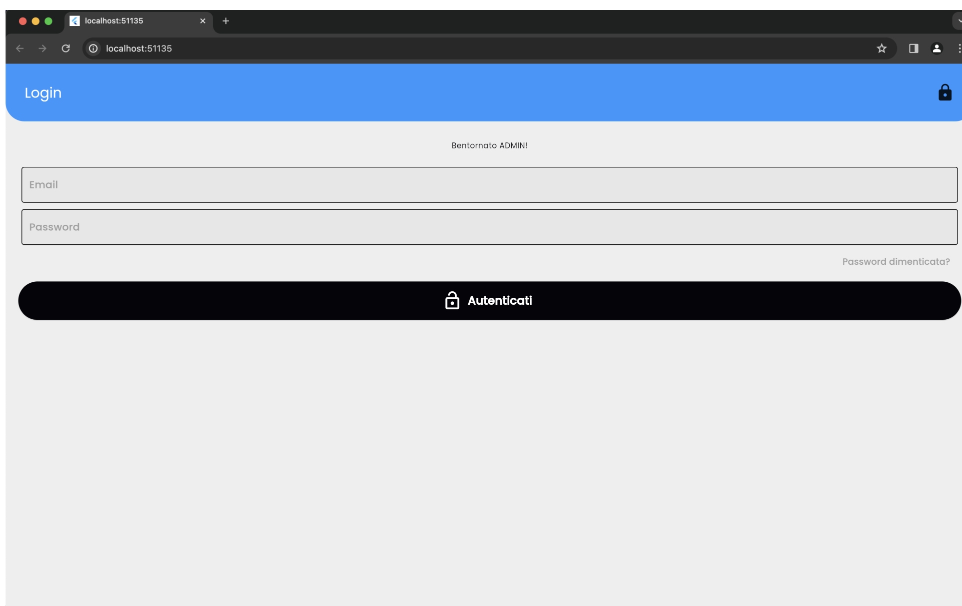


Figura 5.4: Schermata di login

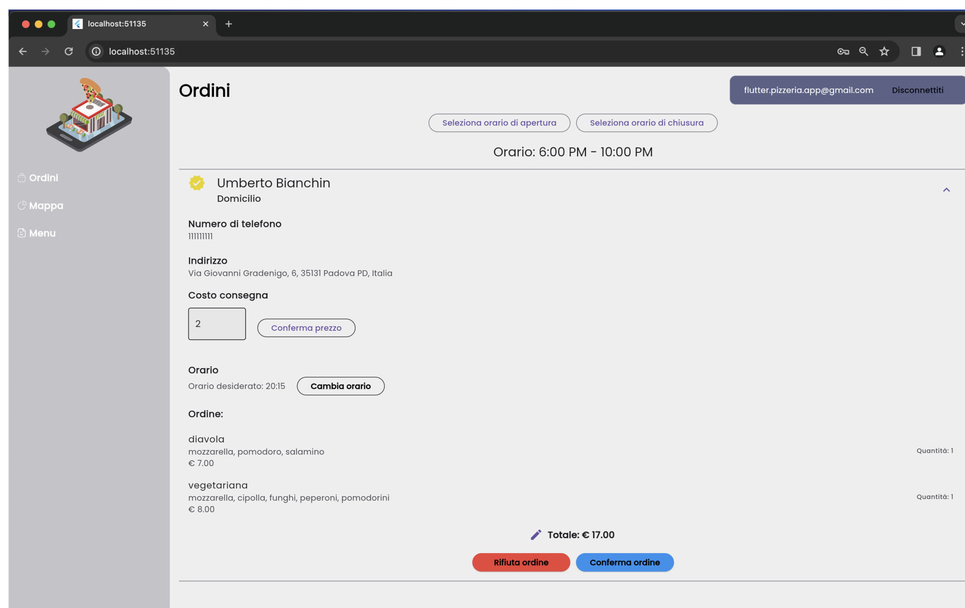


Figura 5.5: Schermata degli ordini

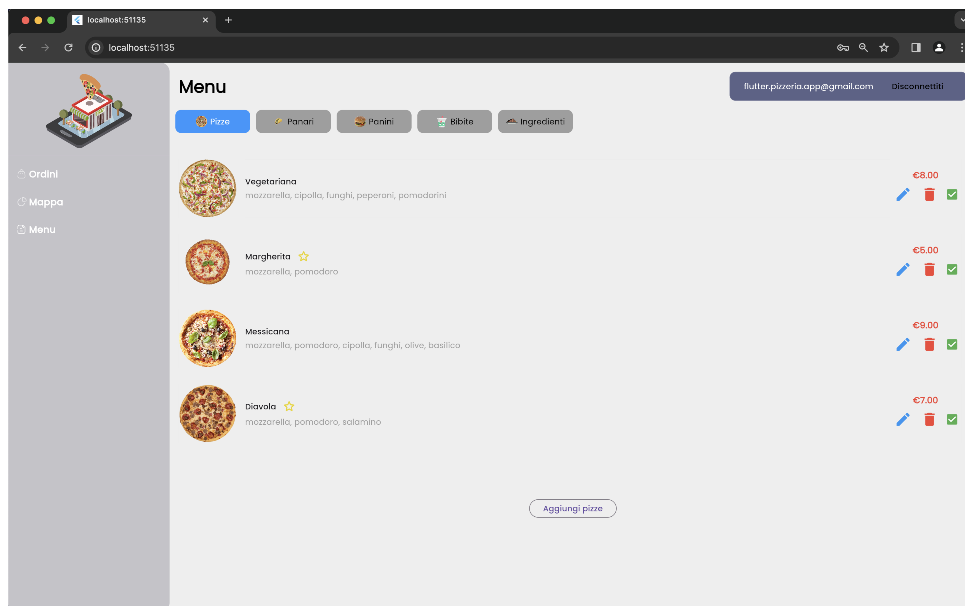


Figura 5.6: Schermata del menu

Come per l’applicazione del cliente, anche il gestionale è collegato al database di Firebase (sezione 5.6), e ciò permette una perfetta integrazione e comunicazione in tempo reale tra le due app.

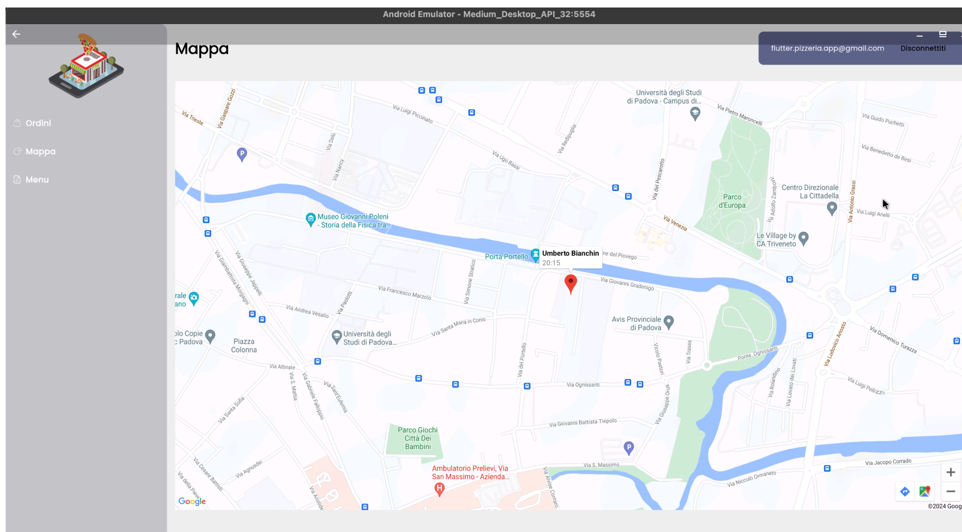


Figura 5.7: Schermata della mappa

5.5 Gestione dello Stato e della Logica di Business

La gestione dello stato e la logica di business in un’applicazione Flutter rappresentano aspetti cruciali per garantire che l’app sia reattiva e facilmente scalabile. Nello sviluppo delle applicazioni dedicate alla pizzeria, è stata usata la classe *ChangeNotifier*[16], fornita direttamente da Flutter per una gestione dello stato pulita ed efficiente. L’adozione di ChangeNotifier permette di incapsulare la logica di business e lo stato dell’applicazione in modelli ”ascoltabili”, che notificano gli ascoltatori iscritti (in questo caso, i widget dell’interfaccia utente) ogni volta che si verifica una modifica, assicurando così aggiornamenti reattivi e coerenti. Gli sviluppatori possono definire modelli o provider di stato che estendono ChangeNotifier e utilizzano il metodo *notifyListeners()* per segnalare che una o più proprietà dell’oggetto sono cambiate. Questo pattern si adatta perfettamente alle esigenze delle applicazioni mobili moderne, dove l’aggiornamento dinamico dell’interfaccia grafica in risposta alle interazioni dell’utente o ai cambiamenti dei dati è fondamentale per una buona esperienza utente.

In entrambe le applicazioni sono state create diverse classi che estendono ChangeNotifier. Nell’applicazione cliente, per esempio, questa classe viene utilizzata per gestire l’aggiornamento del menu. Sfruttando la potenza di Firebase Cloud Firestore, il metodo ascolta in tempo reale i cambiamenti nei documenti ’elements’ ed ’ingredients’ della collezione ’menu’. Quando una modifica è rilevata, il metodo ”updateMenu” (codice 5.2), situato dentro la classe ”MenuProvider” (che estende ChangeNotifier) avvia le funzioni ”retrieveMenu” e ”retrieveIngredients”, responsabili del recupero dei dati aggiornati dal database. Questo assicura che l’applicazione disponga sempre della versione più recente del menu e della lista degli ingredienti, cruciali per offrire agli utenti un’esperienza accurata e aggiornata. Infine il metodo invoca ”notifyLi-

steners()" per notificare gli ascoltatori registrati di una modifica dello stato. Questo causa la ricostruzione dei widget che dipendono dal MenuProvider, come le schermate di visualizzazione del menu e del carrello, garantendo che l'interfaccia utente sia sempre allineata con i dati più recenti.

```dart
void updateMenu(BuildContext context) {
    // Ottiene un riferimento al documento 'elements' nella collezione 'menu'
    final menu = FirebaseFirestore.instance.collection('menu').doc("elements");

    // Ottiene un riferimento al documento 'ingredients' nella collezione 'menu'
    final ingredients =
        FirebaseFirestore.instance.collection('menu').doc("ingredients");

    // Imposta un listener su 'menu' per ascoltare i cambiamenti in tempo reale
    menu.snapshots().listen((querySnapshot) async {
        // Quando un cambiamento avviene, recupera i dati aggiornati del menu
        await retrieveMenu();

        if (context.mounted) {
            // Valida gli elementi presenti nel carrello dell'utente in base ai nuovi dati del menu
            validateCartMenu(context);
        }
        notifyListeners();
    });

    // Imposta un listener su 'ingredients' per ascoltare i cambiamenti in tempo reale.
    ingredients.snapshots().listen((querySnapshot) async {
        // Quando un cambiamento avviene, recupera i dati aggiornati degli ingredienti
        await retrieveIngredients();
```

```
25
26        if (context.mounted) {
27            // Valida gli elementi presenti nel carrello dell'utente in
               ↪  base ai nuovi dati degli ingredienti
28            validateCartIngredients(context);
29        }
30        notifyListeners();
31    });
32 }
```

Codice 5.2: Metodo che aggiunge un prodotto al carrello e avvisa gli ascoltatori

In conclusione, utilizzando ChangeNotifier non c’è bisogno di dover passare gli oggetti manualmente attraverso l’albero dei widget o ricorrere a callback complessi; in questo modo si facilita lo sviluppo, consentendo una separazione chiara tra logica di business e UI, un principio fondamentale per la creazione di applicazioni robuste e facilmente estendibili.

5.6 Integrazione di Firebase per il backend e l’autenticazione

Questa sezione discute come Firebase, parte della piattaforma Google Cloud, abbia non solo semplificato il processo di sviluppo backend ma abbia anche arricchito le applicazioni con una suite di potenti strumenti e servizi.

5.6.1 Autenticazione con Firebase Authentication

Grazie all’integrazione con Firebase Authentication[3], per i clienti dell’app è possibile registrarsi sfruttando vari metodi di autenticazione, come l’accesso tramite email e password, o attraverso provider esterni, come Google e Facebook.

Tramite il pacchetto *google_sign_in*[4] l’utente può registrarsi utilizzando il proprio account Google, e tutto questo processo è gestito dal metodo ”googleLogin” (codice 5.3). Allo stesso modo, grazie al pacchetto flutter_facebook_auth[5] l’utente può registrarsi utilizzando il proprio account Facebook, procedimento gestito dal metodo ”facebookLogin” (codice 5.4).

[3] https://firebase.google.com/docs/auth?hl=it

[4] https://pub.dev/packages/google_sign_in

[5] https://pub.dev/packages/flutter_facebook_auth

Tutto ciò è reso possibile dagli standard di autenticazione utilizzati da Firebase Authentication, come *OAuth 2.0*[6] e *OpenID Connect*[7]. Il primo è uno standard di autorizzazione che permette ad applicazioni di terze parti di ottenere accesso limitato a un servizio web, sia per conto di un utente, sia per conto dell'applicazione stessa. Sfruttando OAuth 2.0 l'applicazione non utilizza direttamente le credenziali di un utente, ma riceve un "token di accesso" che le permette di di accedere ad una specifica parte del suo account e per un periodo di tempo limitato. OpenID Connect è invece un protocollo di autenticazione basato su OAuth 2.0, fornisce un modo per le applicazioni di verificare l'identità dell'utente finale basandosi sull'autenticazione realizzata da un server di autorizzazione, oltre a ottenere informazioni di base sul profilo dell'utente.

L'integrazione di OAuth 2.0 e OpenID Connect con Firebase Authentication consente agli sviluppatori di costruire applicazioni che non solo proteggono le risorse e i dati degli utenti attraverso meccanismi di autorizzazione robusti, ma forniscono anche un'esperienza di autenticazione fluida e sicura per gli utenti finali. Gli sviluppatori possono concentrarsi sullo sviluppo delle funzionalità principali dell'applicazione, lasciando a Firebase il compito di gestire gli aspetti complessi dell'autenticazione e dell'autorizzazione.

```
1  // Metodo per effettuare il login tramite Google
2  Future googleLogin() async {
3      try {
4          final googleUser = await googleSignIn.signIn();
5          if (googleUser == null) return;
6          _user = googleUser;
7
8          // Si recuperano le credenziali di autenticazione dall'account
9          // Google dell'utente.
10         final googleAuth = await googleUser.authentication;
11
12         // Si crea un nuovo record di credenziali utilizzando il token
13         // di accesso e l'ID token di Google.
14         final credential = GoogleAuthProvider.credential(
15         accessToken: googleAuth.accessToken,
16         idToken: googleAuth.idToken,
17         );
```

[6]https://oauth.net/2/

[7]https://openid.net/

```
18
19          // Si utilizzano queste credenziali per accedere nell'istanza
20          // di FirebaseAuth
21          await FirebaseAuth.instance.signInWithCredential(credential);
22
23          // Si salva il tipo di registrazione in un metodo custom,
24          // per uso futuro
25          saveRegType("google");
26      } catch (e) {
27          // Gestione errori
28          return;
29      }
30
31      // Si imposta lo stato di login su vero e si notificano i listeners
32      // che ci sono stati cambiamenti
33      _isLogged = true;
34      notifyListeners();
35  }
```

Codice 5.3: Metodo che gestisce il login tramite Google

```
1   // Metodo per effettuare il login tramite Facebook
2   Future facebookLogin(BuildContext context) async {
3       try {
4
5           final LoginResult loginResult = await FacebookAuth.instance
6               .login(permissions: ["public_profile", "email"]);
7
8           // Si recuperano le credenziali di autenticazione dall'account
9           // Facebook dell'utente.
10          final OAuthCredential facebookAuthCredential =
11          FacebookAuthProvider.credential(loginResult.accessToken!.token);
12
13          // Si utilizzano queste credenziali per accedere nell'istanza
14          // di FirebaseAuth
15          await FirebaseAuth.instance
16              .signInWithCredential(facebookAuthCredential);
17
```

```
18          // Si salva il tipo di registrazione in un metodo custom,
19          // per uso futuro
20          saveRegType("facebook");
21      } on FirebaseAuthException catch (e) {
22          // Gestione errori
23          return;
24      }
25
26      // Si imposta lo stato di login su vero e si notificano i listeners
27      // che ci sono stati cambiamenti
28      _isLogged = true;
29      notifyListeners();
30  }
```

Codice 5.4: Metodo che gestisce il login tramite Facebook

5.6.2 Backend con Firebase Firestore

Firebase Firestore[8], noto anche come Cloud Firestore, è un database cloud NoSQL offerto da Firebase. Grazie alla sua flessibilità e scalabilità è possibile usarlo dall'inizio alla fine dello sviluppo di un'applicazione, senza dover gestire manualmente la scalabilità del database.

Nelle due applicazioni presentate sopra è stato usato principalmente per conservare tutte le informazioni riguardanti il menu e per permettere ai clienti di vedere direttamente le modifiche apportate dai gestori del locale. Il metodo descritto nel codice 5.5 viene chiamato quando, dal gestionale, viene aggiunto o modificato un elemento esistente; la logica è molto semplice: si crea una mappa chiave-valore in cui la chiave è il nome dell'elemento, e il valore è a sua volta un'altra mappa contenente tutte le proprietà che lo riguardano. Una volta aggiunti tutti gli elementi presenti nella lista, si carica il menu nel database, in uno specifico documento.

Il metodo descritto nel codice 5.6 viene invece invocato ogni volta che l'utente accede all'applicazione e (nel caso l'app sia già aperta) quando viene apportata una modifica dal gestore, sincronizzando correttamente gli elementi. La logica, anche in questo caso, è piuttosto diretta: il metodo si connette ad un documento specifico in Firestore (riga 7 e 8), estrae i dati (riga 9), e poi li itera per creare oggetti "DataItem" che popolano la lista "menu", che verrà poi usata per creare l'interfaccia grafica. Un aspetto importante di questo metodo è l'utilizzo della parola chiave "Future", insieme ad "async" e "await": "async" sta ad indicare che "getMenu" è un metodo asincrono, "Future" indica che il metodo restituisce appunto un oggetto "Future" che,

[8] https://firebase.google.com/docs/firestore?hl=it

una volta completato, conterrà un valore di tipo "List<DataItem>". Quando poi si va a chiamare "getMenu" in un altra parte del codice, si deve utilizzare la parola "await": invece di procedere immediatamente alla prossima linea di codice, Dart aspetta che il "Future" si risolva in un valore (o in un errore). Una volta che è completato, la funzione asincrona riprende l'esecuzione, e il valore risultante dal Future può essere usato come se fosse stato restituito in modo sincrono.

```
// Metodo per caricare il menu nel database
void saveMenu(List<DataItem> menu) {
    // Inizializza un dizionario vuoto che conterrà il menu da caricare
    // dynamic permette di inserire tipi diversi come valori di ogni
    ↪ chiave
    final Map<String, dynamic> jsonMenu = {};

    // I valori di ogni chiave (nome dell'elemento) sono tutte le
    ↪ diverse
    // proprietà che devono essere salvate sul database
    for (DataItem item in menu) {
        jsonMenu[item.name.toLowerCase()] = {
        "price": item.initialPrice,
        "ingredients": item.ingredients.join(', '),
        "category": item.category.name,
        "imageUrl": item.image.url,
        "important": item.important,
        "available": item.available,
        };
    }

    // Carica il dizionario jsonMenu nel documento "elements" della
    ↪ collezione "menu" in Firestore.
    // Se il documento già esiste, verrà sovrascritto con i nuovi dati.
    firestoreInstance.collection("menu").doc("elements").set(jsonMenu);
}
```

Codice 5.5: Metodo per caricare il menu nel database

```
// Metodo per scaricare il menu dal database
// Future viene usato per effettuare operazioni asincrone
Future<List<DataItem>> getMenu() async {
```

```
4       // Inizializza una lista vuota di DataItem, che rappresenta un
        ↪  elemento del menu
5       final List<DataItem> menu = [];
6       // Ottiene uno snapshot del documento specifico 'elements' dalla
        ↪  collezione 'menu' in Firestore
7       final snapshot = await FirebaseFirestore.instance
8           .collection('menu').doc("elements").get();
9
10      // Estrae i dati dallo snapshot come un dizionario (Map)
11      final menuData = snapshot.data();
12
13      if (menuData != null) {
14          // Ogni chiave rappresenta un elemento del menu
15          for (String element in menuData.keys) {
16              Categories category = Categories.values
17                  .firstWhere((value) => value.name ==
                    ↪  menuData[element]["category"]);
18
19              // Verifica se l'elemento è disponibile
20              if (menuData[element]["available"]) {
21                  // Aggiunge un nuovo DataItem alla lista del menu
22                  menu.add(
23                  DataItem(
24                      key: UniqueKey(),
25                      name:
26                          element.substring(0, 1).toUpperCase() +
                            ↪  element.substring(1),
27                      ingredients:
                        ↪  menuData[element]["ingredients"].split(", "),
28                      initialPrice: menuData[element]["price"].toDouble(),
29                      category: category,
30                      image: NetworkImage(menuData[element]["imageUrl"]),
31                      important: menuData[element]["important"]),
32                  );
33          }}
34      }
```

```
35     return menu;
36 }
```

Codice 5.6: Metodo per scaricare il menu dal database

Cloud Function

Dopo aver esplorato le capacità del backend con Firebase, è essenziale menzionare le Cloud Functions[9], una funzionalità che eleva ulteriormente l'approccio allo sviluppo backend. Attraverso l'uso di queste funzioni è possibile reagire dinamicamente a eventi generati da Firebase o a richieste HTTPS attraverso il cloud, eseguendo il codice backend in maniera completamente serverless.

Nelle due applicazioni viene usata una Cloud Function, scritta in JavaScript, chiamata "sendNotification" (codice 5.7), che è progettata per essere invocata tramite una chiamata HTTPS. Riceve in input due parametri, il titolo della notifica e il token (simile ad un id univoco) che identifica il dispositivo target. Questa funzione viene chiamata dall'omonimo metodo "sendNotification" (codice 5.8), invocato ogni volta che dal gestionale viene confermato o rifiutato un ordine. In questo modo si permette al cliente di sapere immediatamente, senza dover aprire l'applicazione, lo stato del suo ordine tramite la notifica ricevuta (figura 5.8). Naturalmente, questa operazione è possibile solo se l'utente ha accettato (in fase di registrazione) di ricevere notifiche dall'app.

```
1  // Cloud function in JavaScript per inviare notifiche
2
3  // Importazione dei moduli necessari da Firebase Functions e Firebase
   ↪  Admin SDK
4  const functions = require('firebase-functions');
5  const admin = require('firebase-admin');
6
7  // Ottiene il modulo di messaggistica da Firebase Admin
8  const { getMessaging } = require('firebase-admin/messaging');
9
10 // Inizializza l'app Firebase Admin per accedere alle sue API da Cloud
   ↪  Function
11 admin.initializeApp();
12
13 // Definisce una Cloud Function HTTPS chiamata "sendNotification"
```

[9] https://firebase.google.com/docs/functions?hl=it

```
14 exports.sendNotification = functions.https.onCall((data, context) => {
15
16     // Prepara il messaggio di notifica usando i dati ricevuti dalla
       ↪  chiamata
17     const message = {
18         notification: {
19             title: data["title"],
20             body: "Controlla la sezione ordine per saperne di più",
21         },// Il token del dispositivo a cui inviare la notifica
22         token: data["token"],
23     };
24
25     // Invia il messaggio utilizzando Firebase Cloud Messaging
26     return getMessaging().send(message).catch(error => {
27         // Gestisce gli errori lanciando un errore HTTP
28         throw new functions.https.HttpsError('internal', error);
29     });
30 });
```

Codice 5.7: Cloud function in JavaScript

```
1  // Metodo per inviare notifiche tramite la Cloud Function
2  void sendNotification(String title, String token) async {
3    try {
4      await
       ↪  FirebaseFunctions.instance.httpsCallable('sendNotification').call({
5        "title": title,
6        "token": token,
7        });
8    } catch (error) {
9      // Gestione errori
10 }}
```

Codice 5.8: Metodo per inviare una notifica

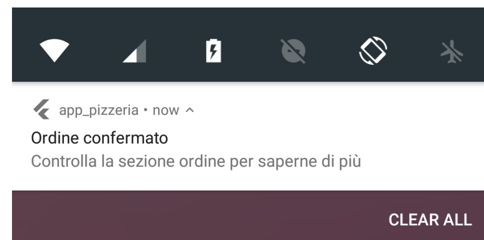


Figura 5.8: Notifica ricevuta tramite Cloud Function

5.7 Aggiornamenti futuri

Nonostante l’applicazione cliente attualmente offra una piattaforma comoda per visionare il menu ed effettuare ordini, un’area chiave per miglioramenti futuri riguarda l’integrazione di un metodo di pagamento in-app. Implementando una soluzione di pagamento digitale, ad esempio attraverso l’uso di piattaforme come Stripe, l’applicazione non solo potrà offrire un’esperienza utente ancora più integrata, ma anche aumentare la soddisfazione del cliente eliminando la necessità di transazioni fisiche. Questo aggiornamento faciliterebbe inoltre un processo di checkout più veloce e sicuro, allineando l’applicazione con le aspettative moderne dei consumatori per transazioni online immediate e protette.

Per quanto riguarda l’applicazione gestionale, una possibile evoluzione consiste nella creazione di un sistema avanzato per la gestione delle consegne. Attualmente, nella sezione mappa è possibile solamente visualizzare le posizioni degli ordini a domicilio, ma l’introduzione di un algoritmo per il raggruppamento degli ordini basato sugli orari e per il calcolo del percorso ottimale per le consegne non solo migliorerebbe la puntualità e ridurrebbe i costi di consegna, ma aumenterebbe anche la soddisfazione del cliente grazie alla capacità di gestire più ordini in modo più efficiente. L’integrazione di questa tecnologia rappresenterebbe un passo avanti significativo nell’ottimizzazione delle operazioni quotidiane della pizzeria.

Queste proposte di aggiornamento sottolineano l’impegno verso un miglioramento continuo e l’adozione di tecnologie innovative per rispondere alle esigenze e alle aspettative in evoluzione dei clienti e della gestione dell’attività. Introducendo queste funzionalità, le applicazioni non solo rimarrebbero all’avanguardia nel contesto tecnologico attuale, ma potrebbero anche offrire un modello replicabile per altri nel settore della ristorazione.

Capitolo 6

Conclusioni

Questa tesi ha inizialmente esplorato il panorama dello sviluppo di applicazioni mobili, analizzando i pro e contro dei vari tipi di programmazione, per poi focalizzarsi sullo sviluppo cross-platform, incentrando l'attenzione sul framework Flutter. L'obiettivo principale è stato quello di mostrare come Flutter possa semplificare la creazione di app multipiattaforma, attraverso l'analisi di due applicazioni nel contesto della ristorazione.

È stata svolta un'analisi approfondita dei componenti principali di Flutter come i widget, che permettono la creazione di interfacce ricche e complesse tramite la composizione di elementi più semplici. Si è poi descritto come venga semplificata anche la parte di testing dei vari componenti o dell'applicazione nel suo complesso, ed anche dell'integrazione con il codice nativo di una piattaforma.

Dal punto di vista personale, l'esperienza con Flutter è stata estremamente positiva. Nonostante un'iniziale curva di apprendimento abbastanza importante, la documentazione ben fatta e le risorse disponibili, quali pacchetti esterni ed esempi, hanno giocato un ruolo fondamentale nello studio e nella comprensione di questo framework. Varie funzionalità hanno sicuramente velocizzato il processo di sviluppo delle applicazioni: in primo luogo la funzione di hot reload, che ha permesso di apportare modifiche vedendo subito i loro effetti, senza perdere lo stato dell'applicazione o dover fare un restart completo. Seguono poi i plugin esterni disponibili per Flutter, come il *Map Location Picker* utilizzato per la selezione dell'indirizzo nell'applicazione cliente, che rendono possibile l'integrazione di funzionalità aggiuntive in minor tempo, ed inoltre, essendo (quasi tutti) open-source, è possibile modificarli a proprio piacimento aggiungendo o cambiando parti di essi.

Un altro punto a favore di Flutter è la sua integrazione con Firebase, che mette a disposizione diversi strumenti per sviluppare la propria applicazione. Lo strumento utilizzato maggiormente in questa tesi è stato Firebase Firestore, un database non relazionale; grazie a questo è stato possibile costruire in modo semplice ed intuitivo un database per la conservazione dei dati relativi al

menu, agli ordini e agli utenti della pizzeria, permettendo modifiche real-time sia lato gestionale sia lato cliente. Un altro servizio di Firebase molto utile sono state le Cloud Functions, che in questo contesto permettono l'invio di una notifica al cliente quando il suo ordine viene accettato o rifiutato, semplicemente scrivendo una funzione JavaScript.

Uno svantaggio incontrato durante la creazione di queste due applicazioni con Flutter è stato la creazione di schermate particolarmente complesse, specialmente in relazione al codice Dart. Componendo i widget tra loro è risultato a volte difficile capire quale modificare per ottenere un determinato effetto, se andare a cambiare il widget più interno piuttosto che quello più esterno, ma tuttavia con un po' di pratica e di prove diventa un problema facilmente superabile.

In conclusione, questo lavoro fornisce una valida dimostrazione delle potenzialità di Flutter, contribuendo al campo dello sviluppo di applicazioni cross-platform. Lo sviluppo multipiattaforma, in particolare, si conferma come un ambito ricco di opportunità, dove l'innovazione tecnologica e la creatività degli sviluppatori si incontrano per creare soluzioni che migliorano concretamente la vita delle persone.

Bibliografia

[1] Google Developers. *Flutter Live - Flutter Announcements and Updates (Livestream)*. 2018. URL: https://www.youtube.com/watch?v=NQ5HVyqg1Qc&t=4842s.

[2] Google Developers. *Dart DevTools*. 2024. URL: https://dart.dev/tools/dart-devtools.

[3] Google Developers. *Testing Flutter apps*. 2024. URL: https://docs.flutter.dev/testing/overview.

[4] Google Developers. *Why did Flutter choose to use Dart?* 2024. URL: https://docs.flutter.dev/resources/faq#why-did-%EF%AC%82utter-choose-to-use-dart.

[5] Google Developers. *Writing custom platform-specific code*. 2024. URL: https://docs.flutter.dev/platform-integration/platform-channels.

[6] B. Eisenman. *Learning React Native: Building Native Mobile Apps with JavaScript*. O'Reilly Media, 2018. ISBN: 978-1491989142.

[7] JetBrains. *Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2022*. 2022. URL: https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/.

[8] T. Kol. *Performance Limitations of React Native and How to Overcome Them*. 2016. URL: https://talkol.medium.com/performance-limitations-of-react-native-and-how-to-overcome-them-947630d7f440.

[9] W. Leler. *What's Revolutionary about Flutter*. 2017. URL: https://hackernoon.com/whats-revolutionary-about-flutter-946915b09514/.

[10] Microsoft. *Documentazione di Xamarin*. 2024. URL: https://learn.microsoft.com/it-it/xamarin/.

[11] A. Miola. *Flutter Complete Reference: Create beautiful, fast and native apps for any device*. Independently published, set. 2020. ISBN: 979-8691939952.

[12] M. Reynolds. *Xamarin Essentials: Learn how to efficiently develop Android and iOS apps for deployment using the Xamarin platform*. Packt Publishing, 2014. ISBN: 978-1783550838.

[13] SEO Cube S.r.l. *Italia, più smartphone che abitanti: il panorama digitale 2020*. 2020. URL: https://www.ansa.it/pressrelease/lifestyle/2020/11/30/italia-piu-smartphone-che-abitanti-il-panorama-digitale-2020_f8f9d1a2-3895-4f87-b9fc-6af8ddaf5c48.html.

[14] Cordova Team. *Cordova Documentation*. 2024. URL: https://cordova.apache.org/docs/en/10.x/guide/overview/#architecture.

[15] Flutter Team. *Flutter architectural overview*. 2024. URL: https://docs.flutter.dev/resources/architectural-overview.

[16] Flutter Team. *Simple app state management*. 2024. URL: https://docs.flutter.dev/data-and-backend/state-mgmt/simple.

[17] TechCrunch. *Number of mobile app downloads worldwide from 2016 to 2023*. 2024. URL: https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/.

[18] J. M. Wargo. *Apache Cordova 4 Programming*. Addison-Wesley Professional, 2015. ISBN: 978-0134048192.

Ringraziamenti

Al termine di questo elaborato, mi è d'obbligo ringraziare tutte le persone che mi hanno sostenuto durante il mio percorso, universitario e di vita.

Per prima cosa, vorrei ringraziare il mio relatore Mauro Migliardi, per i suoi preziosi consigli e per la sua disponibilità. Grazie per avermi fornito spunti fondamentali nella stesura di questo lavoro.

Non posso non menzionare i miei genitori che da sempre mi sostengono nella realizzazione dei miei progetti. Non finirò mai di ringraziarvi per avermi permesso di arrivare fin qui e per aver creduto in me anche nei momenti di difficoltà.

Ringrazio la mia ragazza Elisa per avermi accompagnato in questo percorso e per aver tirato fuori il meglio di me. Grazie per tutto il tempo che mi hai dedicato e grazie perché ci sei sempre stata.

Grazie al mio collega Ivan che in questi tre anni è stato un esempio da seguire. Grazie a tutti gli amici che mi hanno accompagnato in questo percorso, grazie per essere stati una parte essenziale di questa avventura.