ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea Triennale in Ingegneria e Scienze Informatiche

Sviluppo cross-platform di un e-commerce: un caso di studio basato su Flutter

Tesi di laurea in:
Programmazione ad Oggetti

Relatore
Prof. Mirko Viroli

Candidato
Paolo Pietrelli

II Sessione di Laurea
Anno Accademico 2023-2024

Abstract

In questa tesi, inizieremo con una panoramica complessiva delle tecnologie mobili, esaminando il loro sviluppo storico e l'evoluzione fino alle tecnologie che utilizziamo quotidianamente. Analizzeremo la loro influenza e le tendenze attuali del mercato. Successivamente, ci concentreremo sulle tecnologie utilizzate nel settore dello sviluppo mobile, esplorando le soluzioni native, ibride, cross-platform e le progressive web app.

Proseguiremo con una panoramica su Flutter, effettuando un'analisi comparata con altri framework e linguaggi di programmazione, per comprendere le ragioni della sua scelta per il caso di studio. Approfondiremo diversi aspetti di Flutter, inclusi i suoi principali widget, la tecnologia hot reload, l'allineamento degli widget e l'annidamento.

Descriveremo poi Firebase, confrontandolo con altre soluzioni e spiegando perché è stato scelto per la realizzazione dell'e-commerce.

Infine, discuteremo il caso di studio relativo all'e-commerce, per il quale sono state realizzate due versioni differenti: una multi-vendor e una single-vendor. Nella versione multi-vendor l'attenzione è stata rivolta sulle funzioni base per la creazione di un e-commerce per la rivendita di prodotti per animali, fornendo un'interfaccia semplice per i negozianti e i clienti. Nella versione single-vendor, è stato sviluppato uno store di libri cinesi, con un approfondimento sia della GUI che degli aspetti tecnologici. Sono state esaminate le potenzialità di Flutter, realizzando anche una piccola interfaccia social con post e una sezione di messaggistica con notifiche in tempo reale, per migliorare l'interazione tra i clienti, l'usabilità e l'accessibilità, con particolare attenzione al supporto multilinguistico.

Desidero ringraziare il professor Mirko Viroli per i consigli che ha saputo darmi e per la disponibilità concessami nella realizzazione di questa tesi. Ringrazio la mia ragazza Yang Fan e i miei cani Ciccio e Nana per essermi stati di ispirazione nella realizzazione di questa tesi. Ringrazio tutta la mia famiglia e i loro cani Mia, Whisky e Laila per il prezioso sostegno che mi hanno sempre dato. In fine e non per importanza, vorrei ringraziare i miei amici Dominik, Beatrice, Francesco e tutto il gruppo del ChiBurdel

Contents

List of Figures

List of Listings

Chapter 1

Introduzione

Flutter è un kit di sviluppo software (*UI toolkit*) **open-source** creato da Google, progettato per facilitare la realizzazione di applicazioni **cross-platform**. Il termine **"cross-platform"** (noto anche come **"multi-platform software"** o **"platform-independent software"**) si riferisce a software in grado di funzionare su più sistemi operativi o piattaforme con il minimo sforzo di configurazione [27].

Grazie a Flutter, è possibile scrivere un unico codice che consente di sviluppare applicazioni compatibili con **web, Fuchsia, Android, iOS, Linux, macOS** e **Windows**[28].

Flutter utilizza il linguaggio di programmazione **Dart**, un linguaggio orientato agli oggetti che si distingue per la sua semplicità e intuitività. Dart è stato progettato per essere facile da apprendere e utilizzare, con una sintassi chiara e moderna, e offre una performance elevata grazie alla compilazione nativa (AOT, Ahead Of Time) per le applicazioni mobile e alla compilazione just-in-time (JIT) per il ciclo di sviluppo.

Un aspetto chiave di Flutter è il suo approccio basato sui widget: in Flutter, *"everything is a widget"*, il che significa che ogni elemento dell'interfaccia utente viene trattato come un widget personalizzabile e riutilizzabile. Questo concetto, combinato con le caratteristiche di Dart, permette agli sviluppatori di creare interfacce utente reattive e fluide con meno codice e maggiore efficienza.

Nel contesto di questa tesi, verrà presentato un progetto di e-commerce sviluppato con Flutter, progettato per funzionare in modo uniforme su **web, iOS** e

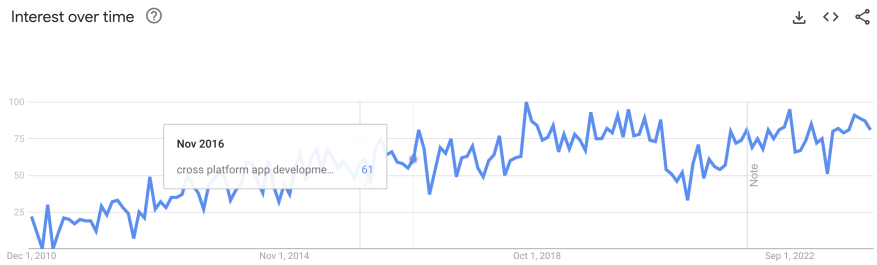


Figure 1.1: cross-platform trend dal 2010 al 2022 [21].

Android.

1.1 Contestualizzazione dello sviluppo mobile

L'evoluzione dello sviluppo mobile rappresenta una delle trasformazioni tecnologiche più significative degli ultimi decenni. Dall'avvento dei primi telefoni cellulari, che permettevano solo chiamate e messaggi di testo, si è passati rapidamente a dispositivi sempre più potenti, fino agli smartphone, capaci di supportare applicazioni complesse che hanno rivoluzionato il modo in cui le persone interagiscono con la tecnologia. La nascita di sistemi operativi come iOS e Android, insieme all'introduzione di app store, ha aperto la strada a un ecosistema di applicazioni mobili in continua crescita, influenzando settori che dell'intrattenimento, della salute, del commercio elettronico, dei social media e persino delle piccole medie imprese che grazie ad applicazioni e gestionali elettronici ne vengono facilitati i processi di lavoro.

Parallelamente, le tecnologie di sviluppo si sono evolute per rispondere alle esigenze di un mercato sempre più competitivo e diversificato. Se in origine lo sviluppo di app richiedeva competenze specifiche per ciascuna piattaforma (sviluppo nativo), negli ultimi anni abbiamo assistito alla crescente adozione di framework cross-platform. Questi strumenti, come Flutter, permettono di scrivere un singolo codice capace di funzionare su più piattaforme, riducendo tempi e costi di sviluppo, e rendendo più accessibile la creazione di applicazioni multi-dispositivo (come si mostra in Figura 1.1).

In questo contesto, l'emergere di tecnologie innovative ha reso le app mobili

sempre più centrali nella vita quotidiana degli utenti, spingendo le aziende a investire in soluzioni mobile-first per raggiungere un pubblico globale. Questo sviluppo ha gettato le basi per la trasformazione digitale che caratterizza la nostra società moderna.

1.2 Significato e impatto delle applicazioni mobili nella società moderna

[14]Le applicazioni mobili hanno profondamente trasformato il nostro modo di vivere, lavorare e comunicare. Originariamente concepite come semplici strumenti di utilità, le app si sono evolute in potenti piattaforme di socializzazione e intrattenimento, diventando elementi imprescindibili della nostra vita quotidiana. La diffusione capillare degli smartphone ha reso possibile l'accesso a una vasta gamma di servizi direttamente dal palmo della mano, facilitando attività che spaziano dall'acquisto di beni e servizi alla gestione della salute, dall digital banking fino alla comunicazione istantanea con persone in tutto il mondo.

Questa pervasività delle app ha influenzato significativamente il nostro stile di vita e il tempo che trascorriamo davanti agli schermi. Mediamente, ogni persona spende circa 3 ore e 47 minuti al mese utilizzando applicazioni mobili sui propri dispositivi, questi numeri incrementano esponenzialmente quando si parla dei giovani, figura 1.2. Questo dato riflette non solo l'importanza crescente delle app nella nostra routine quotidiana, ma anche la loro capacità di plasmare il modo in cui interagiamo con il mondo circostante.

L'impatto economico delle applicazioni mobili è altrettanto significativo. Il settore delle app ha creato nuovi modelli di business, come l'economia delle app e le piattaforme di e-commerce, che hanno ampliato le opportunità di mercato e generato nuove fonti di reddito per sviluppatori e imprese. Inoltre, l'ecosistema delle app ha contribuito alla crescita dell'occupazione, stimolando la domanda di competenze specializzate nello sviluppo di software e nella gestione di piattaforme digitali.

Le dimensioni del mercato globale dell'industria delle applicazioni mobili sono cresciute di 206,73 miliardi di dollari entro la fine del 2022. Si prevede che

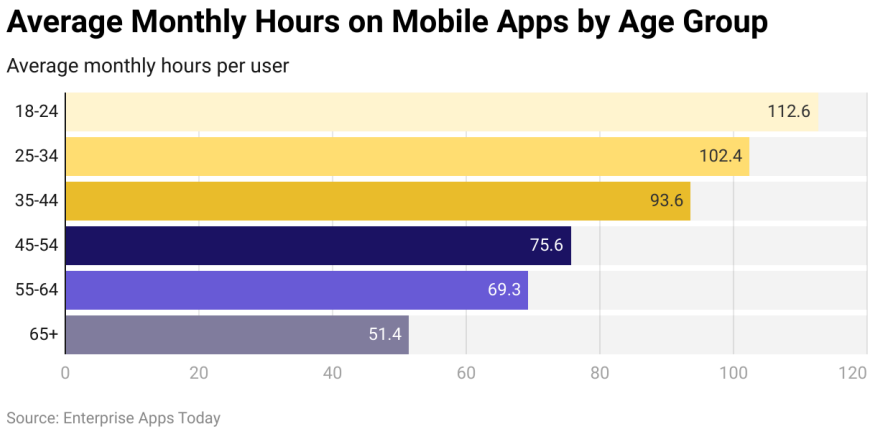


Figure 1.2: Ore mensili spese sulle applicazioni mobile [14].

l'industria delle applicazioni mobili raggiungerà un tasso di crescita annuale composto (CAGR) del 13,4%, arrivando a 565,4 miliardi di dollari entro la fine del 2030, figura 1.3.

Oltre all'aspetto economico, le applicazioni mobili hanno influenzato profondamente i comportamenti sociali. L'accesso immediato a informazioni e servizi ha cambiato le aspettative degli utenti, spingendo verso un mondo sempre più connesso e interattivo. Tuttavia, questa dipendenza crescente dalle app solleva anche questioni riguardanti la privacy, la sicurezza e la sostenibilità digitale.

Guardando al futuro, è chiaro che le applicazioni mobili continueranno a essere un motore di innovazione, spingendo i confini di ciò che è possibile con la tecnologia. La capacità di adattarsi rapidamente ai cambiamenti tecnologici e di rispondere alle nuove esigenze degli utenti sarà fondamentale per le aziende che vogliono rimanere competitive in un mercato in continua evoluzione.

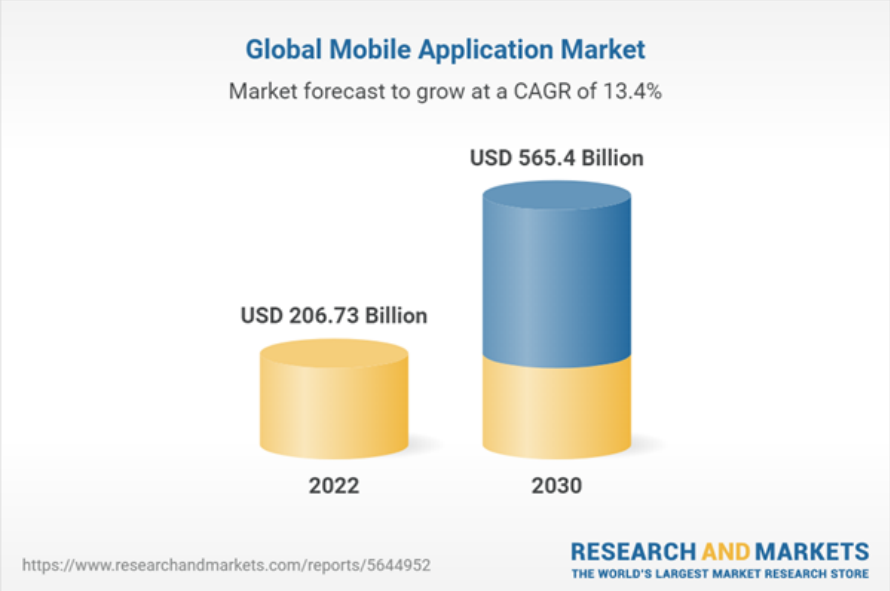


Figure 1.3: Stima di crescita del mercato globale delle applicazioni mobile [14].

Chapter 2

Le applicazioni mobile

Le applicazioni mobili sono diventate una parte essenziale della nostra vita quotidiana, utilizzate per una vasta gamma di attività come comunicare con gli amici, ordinare cibo, fare acquisti, pagare tasse, e molto altro. Questo fenomeno ha le sue radici in una serie di innovazioni tecnologiche che hanno rivoluzionato il mondo della comunicazione e dell'informatica.

Oggi, le applicazioni mobili non solo dominano il settore tecnologico, ma influenzano profondamente il nostro modo di vivere e lavorare. L'evoluzione di queste tecnologie continua a plasmare il futuro, rendendo sempre più necessaria la presenza di app mobili dedicate per le imprese che desiderano rimanere competitive in un mercato in rapida evoluzione [11].

2.1 Evoluzione delle applicazioni mobile

Tutto è iniziato nel 1973, quando Martin Cooper inventò il primo telefono cellulare portatile, aprendo la strada all'era della comunicazione mobile. Negli anni successivi, la fondazione di Apple nel 1976 da parte di Steve Jobs, Steve Wozniak e Ronald Wayne segnò l'inizio di una nuova era nell'informatica personale, che avrebbe avuto un impatto profondo anche sullo sviluppo mobile.

Nel 1983, il Motorola DynaTAC 8000X divenne il primo telefono cellulare disponibile commercialmente, segnando un altro passo fondamentale verso la diffusione della tecnologia mobile. La vera svolta, tuttavia, arrivò negli anni '90 con

il lancio delle reti digitali 2G e l'introduzione del primo telefono GSM, il Nokia 1011, nel 1992. Questi eventi hanno reso possibile una comunicazione mobile più avanzata e affidabile, aprendo la strada alla creazione dei primi smartphone.

Il primo smartphone per uso generale, l'IBM Simon, fu introdotto nel 1994 e rappresentò una pietra miliare, combinando funzionalità telefoniche con quelle di un assistente digitale. Negli anni successivi, dispositivi come il PalmPilot e il BlackBerry hanno contribuito a definire il concetto di dispositivo mobile multifunzione, ponendo le basi per l'ecosistema di app che conosciamo oggi.

Il vero punto di svolta arrivò nel 2007, quando Steve Jobs presentò l'iPhone, un dispositivo che non solo ridefinì il concetto di smartphone, ma rivoluzionò anche il modo in cui le applicazioni venivano sviluppate e distribuite. L'anno successivo, Apple lanciò l'App Store, una piattaforma che ha reso le applicazioni mobili accessibili a un pubblico globale, segnando l'inizio di un mercato in continua espansione.

Parallelamente, Android, acquisita da Google nel 2005, emerse come un altro attore chiave nel mercato mobile con il lancio del primo dispositivo Android nel 2008. La competizione tra iOS e Android ha accelerato l'innovazione, portando alla creazione di un'incredibile varietà di app che oggi utilizziamo quotidianamente [11].

Questa rapida evoluzione, guidata da innovazioni tecnologiche e dalla crescente domanda di connettività, ha portato allo sviluppo di un ecosistema di applicazioni mobile in continua espansione. Le tendenze attuali nel settore delle app, caratterizzate dalla crescente centralità dei social media, delle piattaforme di streaming e dei servizi on-demand, rappresentano la naturale evoluzione di questo percorso, evidenziando come le app mobili continuino a trasformare il nostro modo di interagire con la tecnologia e tra di noi.

2.2 Tendenze attuali nel settore delle app

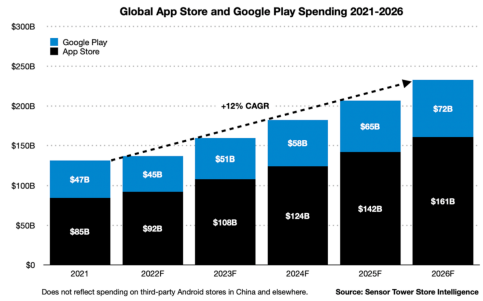
Negli ultimi anni, il settore delle applicazioni mobile ha visto emergere diverse tendenze che riflettono i cambiamenti nei comportamenti degli utenti e nelle tecnologie disponibili. Tra le categorie di applicazioni più popolari e utilizzate, spiccano i social network, che continuano a dominare il mercato globale delle app.

Le piattaforme di social media come TikTok, Instagram e Facebook sono costantemente tra le applicazioni più scaricate, dimostrando l'inarrestabile crescita e influenza di questi strumenti nella nostra vita quotidiana. Queste app non solo hanno trasformato il modo in cui comunichiamo e condividiamo contenuti, ma hanno anche creato nuove opportunità di marketing, intrattenimento e interazione sociale. In particolare, TikTok ha conosciuto un'enorme crescita, diventando un fenomeno globale grazie ai suoi contenuti brevi e coinvolgenti, attirando miliardi di utenti in tutto il mondo.

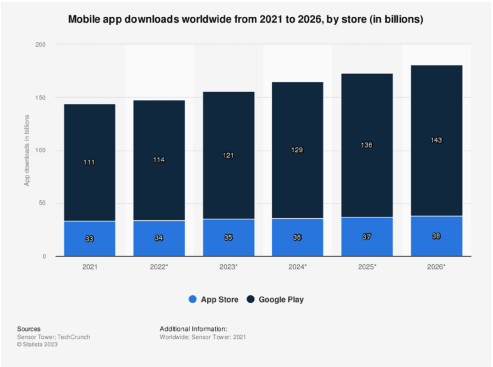
Oltre ai social network, altre categorie di app come le piattaforme di streaming musicale, e-commerce e navigazione hanno raggiunto un'ampia diffusione. Spotify, Amazon e Google Maps sono esempi di applicazioni che hanno rivoluzionato i rispettivi settori, offrendo servizi che si integrano perfettamente nella routine quotidiana degli utenti. Spotify, ad esempio, ha trasformato il modo in cui ascoltiamo musica, mentre Amazon ha reso lo shopping online più accessibile e conveniente che mai. Allo stesso modo, Google Maps ha reso la navigazione e la scoperta di nuovi luoghi un'esperienza intuitiva e indispensabile.

L'uso delle app di comunicazione, come WhatsApp, Messenger e WeChat, è particolarmente rilevante. Secondo le statistiche del 2022, le applicazioni social e di comunicazione rappresentano circa il 42% del tempo totale speso dagli utenti sui propri smartphone. Questo dato sottolinea come queste app non siano più semplici strumenti di comunicazione, ma veri e propri ecosistemi digitali in cui gli utenti trascorrono gran parte del loro tempo, condividendo esperienze, consumando contenuti e interagendo con le proprie comunità online. La centralità di queste applicazioni nel tempo libero e nella vita quotidiana ha consolidato la loro posizione dominante nel mercato delle app.

Un altro aspetto interessante da considerare è il confronto tra i principali store di applicazioni: il Google Play Store e l'Apple App Store. Nonostante il numero di applicazioni scaricate dal Google Play Store sia significativamente superiore a quello dell'App Store, i guadagni generati dalle applicazioni iOS sono nettamente maggiori rispetto a quelli delle app Android. Questo fenomeno è spesso attribuito al diverso profilo demografico degli utenti iOS, che tendono a spendere di più in app e acquisti in-app. Tale dinamica evidenzia come, nonostante Android domini il mercato in termini di quota di mercato e volumi di download, l'ecosistema iOS



(a) Mobile app spending



(b) Mobile app downloads

Figure 2.1: Analisi statistica dei dwnload e delle spese in app [5]

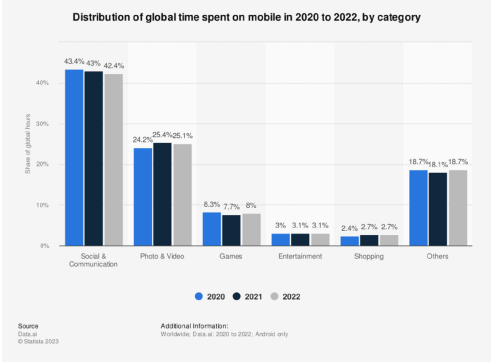
rimanga il leader indiscusso in termini di redditività.

Questa disparità nei ricavi tra le due piattaforme è un segnale importante per gli sviluppatori e le aziende, che spesso privilegiano l'ecosistema iOS per lanciare nuove app o aggiornamenti significativi, cercando di massimizzare i profitti in un mercato che, sebbene più piccolo in termini di numero di download, offre maggiori opportunità di guadagno. L'analisi di questi trend, supportata da grafici e dati statistici, evidenzia come il mercato delle applicazioni mobili sia in continua evoluzione, influenzato non solo dalle tecnologie emergenti, ma anche dalle preferenze e dai comportamenti degli utenti, figura 2.1.

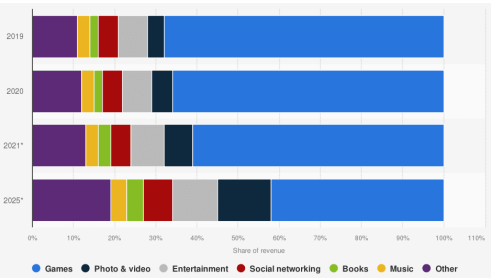
2.3 Categorie di app

Il panorama delle applicazioni mobili è estremamente diversificato, con diverse categorie che si distinguono non solo per il numero di download, ma anche per l'effettivo utilizzo e il tempo speso dagli utenti. Un'analisi più approfondita delle statistiche rivela tendenze interessanti e talvolta sorprendenti, figura 2.2.

Social Network e Comunicazione: Le applicazioni di social networking e comunicazione continuano a dominare il mercato delle app. Nel 2023, le tre app più scaricate al mondo sono state TikTok, Instagram e Facebook, riflettendo una forte preferenza per le piattaforme che favoriscono la condivisione e l'interazione sociale. Instagram, ad esempio, ha registrato ben 68,92 milioni di download, se-



(a) Mobile app spending



(b) Mobile app downloads

Figure 2.2: Analisi statistica delle categorie di app scaricate e di quelle utilizzate [5][26]

guita da Facebook con 38,52 milioni, dimostrando la continua rilevanza di queste piattaforme nell'ecosistema mobile globale. Le app di messaggistica come WhatsApp e il più leggero WhatsApp Business continuano anch'esse a registrare download significativi, con rispettivamente 29,57 milioni e 16,22 milioni di installazioni.

Giochi: Sebbene i giochi costituiscano la categoria con il maggior numero di download su entrambe le piattaforme, con 221.338 applicazioni disponibili solo su iOS (14.669 a pagamento e 206.669 gratuite), il tempo effettivamente speso su queste app rappresenta solo l'8% del totale del tempo passato su dispositivi mobili. Questo suggerisce che, sebbene i giochi siano frequentemente scaricati, molti di essi vengono usati per brevi periodi o vengono abbandonati dopo un uso iniziale.

Shopping e eCommerce: L'eCommerce ha visto una crescita significativa, con la maggior parte dei consumatori che ora preferiscono fare acquisti tramite app mobili rispetto ai siti web tradizionali. Amazon, ad esempio, è tra le app più popolari, contribuendo a una tendenza generale che vede oltre il 63% degli ordini di shopping online provenire da dispositivi mobili. Tuttavia, il tempo speso su queste app è relativamente breve, con sessioni medie che durano tra 4,6 e 5 minuti. Nonostante ciò, le app di shopping generano un coinvolgimento elevato, con il 71% del traffico dei siti di retail proveniente da dispositivi mobili, il che sottolinea l'importanza crescente del mobile commerce.

Bancario e Fintech: Le applicazioni finanziarie e di mobile banking stanno diventando sempre più popolari. Nel 2022, le app bancarie e fintech hanno visto

un aumento del 54% nei download, superando i 26 milioni a livello globale. Il 90% degli utenti ora preferisce utilizzare app mobili per visualizzare il saldo del conto, e quasi tutti i millennials (97%) fanno affidamento su queste app per le loro esigenze finanziarie quotidiane. Inoltre, il tempo medio per sessione su queste app è aumentato, riflettendo un crescente coinvolgimento degli utenti.

Viaggi e Trasporti: Anche il settore dei viaggi ha visto una crescente adozione delle app mobili. Nel 2022, oltre un miliardo di utenti ha utilizzato app per prenotazioni di viaggi, con una preferenza sempre maggiore per le soluzioni mobili rispetto alle tradizionali agenzie di viaggio. Le app di ride-sharing, come quelle per il noleggio di auto, hanno visto un incremento del 27% nelle nuove installazioni, con un aumento del 19,4% degli utenti attivi mensilmente.

Cibo e Ristoranti: Le app per ordinare cibo hanno registrato un impressionante numero di download, con Zomato che guida il mercato con oltre 55 milioni di installazioni nel 2022. Uber Eats segue con 47 milioni, dimostrando l'enorme domanda di servizi di consegna di cibo tramite app. Il mercato della consegna di cibo online è previsto crescere ulteriormente, con un volume di mercato stimato a 2 trilioni di dollari entro il 2027, segnando un trend di crescita annuale del 12,78%.

Disparità tra Download e Utilizzo: Un fenomeno interessante è il divario tra il numero di app scaricate e quelle effettivamente utilizzate. Tra i giovani, in particolare, questo divario è evidente: il 54% degli individui possiede dispositivi con più di tre schermate piene di app, ma solo una piccola parte di queste viene utilizzata quotidianamente. Questo comportamento riflette un trend di download impulsivi seguiti da un utilizzo sporadico o nullo, indicando che non tutte le app scaricate riescono a mantenere l'interesse degli utenti nel lungo termine, figura **??**.

In conclusione, l'analisi delle categorie di app non solo evidenzia la diversità di comportamenti tra le diverse fasce d'età e preferenze, ma sottolinea anche come il mercato delle applicazioni mobili continui a evolversi, con tendenze che variano significativamente a seconda della categoria. La comprensione di queste dinamiche è essenziale per le aziende che desiderano competere efficacemente in questo spazio in continua crescita.

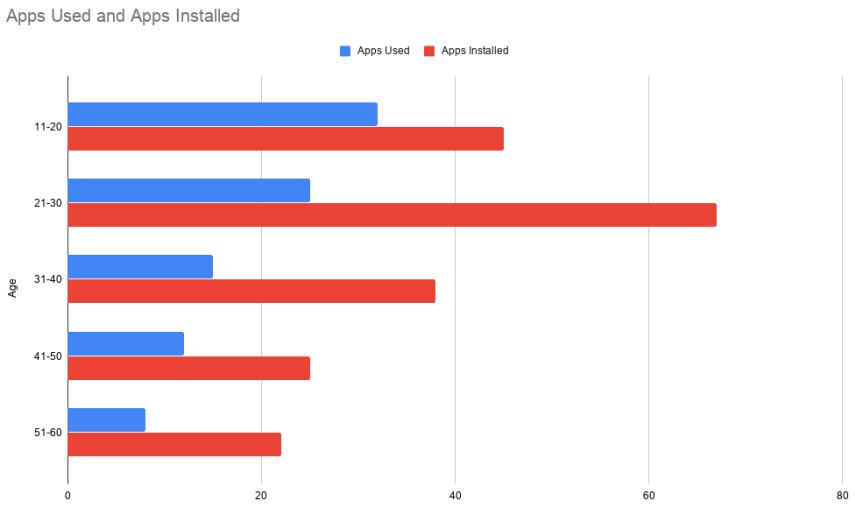


Figure 2.3: App utilizzate App scaricate [5].

2.3.1 Native

Le applicazioni native, figura 2.4, sono sviluppate specificamente per un sistema operativo (OS) particolare, sfruttando al massimo le capacità e le funzionalità del dispositivo. A differenza delle app web mobili, che sono essenzialmente siti web ottimizzati per i dispositivi mobili, le app native sono scritte in linguaggi di programmazione specifici per la piattaforma in cui verranno eseguite. Ad esempio, per sviluppare app native per Android, viene utilizzato il linguaggio **Java** o **Kotlin**, mentre per iOS si utilizzano **Objective-C** o **Swift**.

Un'app nativa si distingue per le **elevate prestazioni** e l'integrazione profonda con il sistema operativo. Questa tipologia di app è ideale per applicazioni che richiedono un *alto livello di personalizzazione* e che devono sfruttare componenti nativi del dispositivo, come le app di gioco, le app di realtà virtuale (VR) e le applicazioni con grafica estesa. Tuttavia, uno svantaggio significativo dello sviluppo nativo è che il codice scritto per una piattaforma non può essere riutilizzato su un'altra, rendendo necessaria la scrittura di codici separati per ogni OS.

Le app native si aggiornano attraverso i rispettivi store di applicazioni e sono accessibili tramite l'icona dell'app presente sul dispositivo. Per sviluppare app

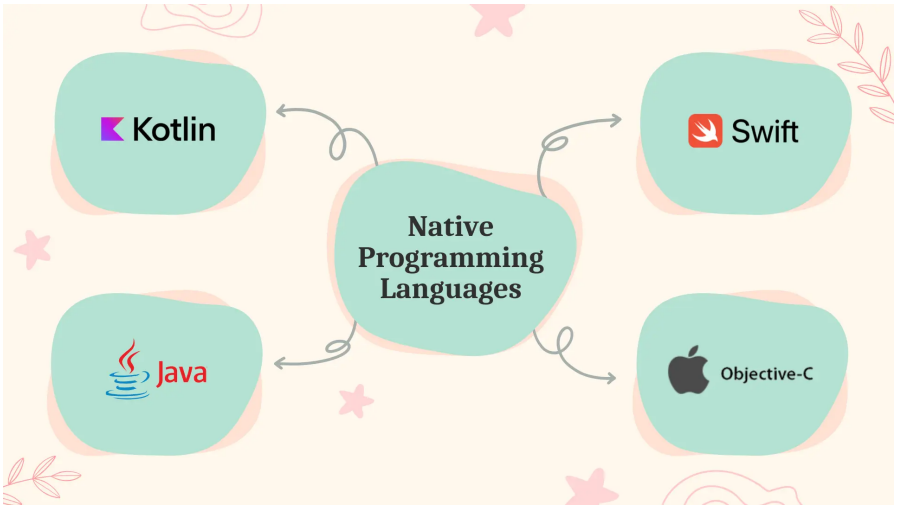


Figure 2.4: Linguaggi di programmazione nativi [20].

native, è cruciale scegliere il linguaggio di programmazione giusto, poiché esso influenzerà le **prestazioni**, la **facilità d'uso**, e l'**esperienza utente complessiva**.

2.3.1.1 iOS

Il sistema operativo **iOS**, sviluppato da **Apple**, è il fulcro dell'ecosistema Apple, che include dispositivi come iPhone, iPad, Apple Watch e Apple TV. Per sviluppare applicazioni iOS, i programmatori utilizzano principalmente due linguaggi di programmazione: **Objective-C** e **Swift**.

2.3.1.2 Objective-C

Objective-C è stato il primo linguaggio di programmazione supportato da Apple per lo sviluppo di applicazioni mobili. Questo linguaggio orientato agli oggetti deriva dalla sintassi del linguaggio **C**, combinata con le caratteristiche della programmazione orientata agli oggetti di *SmallTalk*. Sebbene sia maturo e offra un accesso completo alle API di Apple, Objective-C è considerato **meno intuitivo** rispetto ai linguaggi moderni, a causa della sua sintassi complessa e della difficoltà nella gestione del debugging.

- **Pro:** Maturo, con accesso alle API di Apple, ben adatto per progetti *legacy*.

- **Contro:** Curva di apprendimento ripida, sintassi meno moderna rispetto a Swift.
- **Esempio:** Utilizzato per applicazioni iOS *legacy* come **Airbnb** e **Dropbox**.

2.3.1.3 Swift

Swift, introdotto da Apple nel 2014, ha rapidamente rivoluzionato lo sviluppo delle app iOS. Questo linguaggio moderno è stato progettato per essere **sicuro**, **espressivo** e **performante**, facilitando la creazione di applicazioni sofisticate e intuitive per i diversi dispositivi Apple. Grazie alla sua **sintassi moderna** e alla **libreria standard**, Swift è ora il linguaggio preferito per lo sviluppo di nuove applicazioni iOS.

- **Pro:** Sintassi moderna, alte prestazioni, accesso completo alle API iOS.
- **Contro:** Limitato alle piattaforme Apple, mancanza di compatibilità retroattiva.
- **Esempio:** Dominante nello sviluppo di app iOS moderne, utilizzato da aziende come **Airbnb** e **Lyft**.

2.3.1.4 Android

Android, sviluppato da **Google**, è un sistema operativo open-source che alimenta una vasta gamma di dispositivi mobili prodotti da vari produttori, tra cui Samsung, Huawei e Google stessa. Per sviluppare app Android, i programmatori utilizzano principalmente due linguaggi di programmazione: **Java** e **Kotlin**.

2.3.1.5 Java

Java è stato il linguaggio di riferimento per lo sviluppo di applicazioni Android sin dal lancio ufficiale della piattaforma nel 2008. Questo linguaggio orientato agli oggetti è stato creato nel 1995 ed è conosciuto per la sua **indipendenza dalla piattaforma**, il **supporto di una vasta comunità** e le sue **prestazioni affidabili**. Molte delle librerie e dei codici esistenti sono scritti in Java, rendendolo una scelta pragmatica per molti sviluppatori.

- **Pro:** Indipendenza dalla piattaforma, forte supporto della comunità, prestazioni affidabili.
- **Contro:** Verbosità, consumo di memoria, compilazione più lenta.
- **Esempio:** Utilizzato per applicazioni Android come **WhatsApp**, **Twitter** e **LinkedIn**.

2.3.1.6 Kotlin

Kotlin, supportato ufficialmente da Google dal 2017, è un linguaggio di programmazione moderno che sta guadagnando rapidamente popolarità tra gli sviluppatori Android. Con la sua **syntassi concisa**, l'**interoperabilità con Java** e le **funzionalità di sicurezza avanzate**, Kotlin è diventato la scelta preferita per la programmazione Android. Questo linguaggio offre una **riduzione del codice boilerplate** e una **maggiore sicurezza** rispetto a Java, rendendolo ideale per la creazione di applicazioni Android efficienti e manutenibili.

- **Pro:** Sintassi concisa, interoperabilità con Java, codice più pulito.
- **Contro:** Comunità più piccola rispetto a Java, compilazione più lenta.
- **Esempio:** Adottato da aziende come **Pinterest** e **Trello** per le loro applicazioni Android.

2.3.2 App ibride

Le **app ibride** rappresentano una soluzione efficace per lo sviluppo multipiattaforma, permettendo di scrivere il codice una sola volta per poi eseguirlo su diversi sistemi operativi come **iOS** e **Android**. Questa modalità di sviluppo accelera i tempi di lancio sul mercato, riducendo significativamente il tempo necessario per creare e mantenere due versioni separate di un'app, una per ciascun sistema operativo.

A differenza delle app native, che richiedono un codice specifico per ogni piattaforma, le app ibride combinano le funzionalità delle applicazioni web con quelle native. Utilizzando tecnologie come **HTML5**, **CSS**, e **JavaScript**, queste app

sfruttano i principi del web per offrire esperienze utente che possono avvicinarsi a quelle delle app native. **HTML5** consente di strutturare contenuti con supporto per multimedia e capacità offline, mentre **CSS** aggiunge uno styling visivamente accattivante e design reattivo. **JavaScript**, noto per la sua versatilità, fornisce dinamismo e interattività alle app.

Un aspetto interessante delle app ibride è che esse utilizzano un unico codice sorgente che viene poi "incapsulato" in una struttura nativa attraverso l'uso di framework come **Ionic** e **Apache Cordova**. Questi framework permettono di accedere alle funzionalità native del dispositivo, come la fotocamera o il GPS, mentre si mantiene una base di codice condivisa.

Tra i principali vantaggi delle app ibride ci sono la rapidità di sviluppo e di lancio, la facilità di manutenzione e la riduzione dei costi. Grazie alla possibilità di sviluppare e mantenere una sola versione dell'app per più piattaforme, le startup e le aziende possono ottenere un significativo risparmio sia in termini di tempo che di budget. Inoltre, le app ibride possono offrire un'esperienza utente piuttosto buona, combinando i vantaggi delle app native e web, sebbene possano presentare alcune limitazioni rispetto alle app completamente native, specialmente in termini di prestazioni e compatibilità con tutte le funzionalità specifiche di ciascun sistema operativo.

Tuttavia, le app ibride hanno anche delle limitazioni. Un'importante restrizione è la mancanza di supporto offline completo, che può influire sull'accessibilità dell'app quando non è disponibile una connessione Internet. Inoltre, le app ibride potrebbero non gestire perfettamente alcune funzionalità specifiche di un sistema operativo, a causa delle differenze tra le piattaforme e le loro caratteristiche uniche.

In sintesi, mentre le app ibride offrono una soluzione conveniente e versatile per lo sviluppo multipiattaforma, è importante considerare le specifiche esigenze del progetto e le potenziali limitazioni per determinare se questa modalità di sviluppo è la più adatta [20] [22].

2.3.3 Progressive web app

Quando si discute dei principali linguaggi di programmazione utilizzati per lo sviluppo web e mobile, è fondamentale menzionare le **Progressive Web Apps**

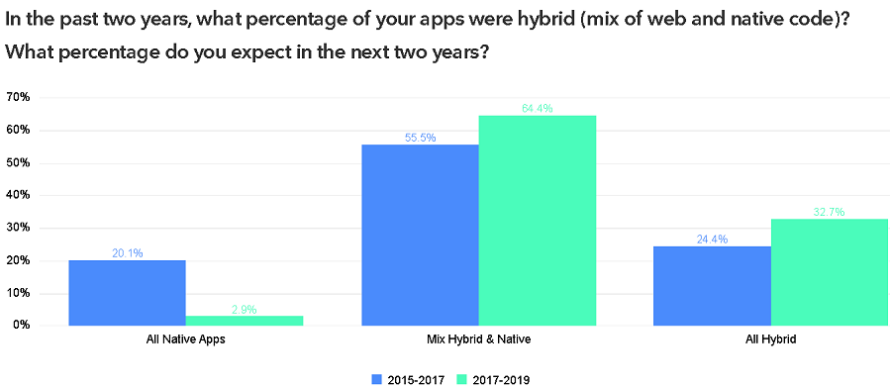


Figure 2.5: Analisi utilizzo linguaggi nativi, Hybridi o un mix di essi[1].

(PWA). Questi siti web sono progettati per offrire un'esperienza utente simile a quella delle applicazioni per smartphone, sia nell'aspetto che nelle funzionalità. La tecnologia PWA consente agli utenti di installare un sito web sui loro smartphone come se fosse un'app, combinando i benefici dei siti web e delle applicazioni native, figura 2.5.

Le PWA offrono numerosi vantaggi rispetto ai siti web tradizionali. In primo luogo, garantiscono **affidabilità**: le applicazioni si caricano istantaneamente, indipendentemente dalla qualità della rete. Inoltre, la **velocità** è notevolmente migliorata; la trasmissione dei dati attraverso la rete è rapida e l'interfaccia utente è fluida e altamente reattiva. Infine, le PWA favoriscono un maggiore **coinvolgimento** degli utenti, offrendo un'esperienza comoda che stimola l'uso continuo e il ritorno frequente.

Dal punto di vista di Google, le PWA si distinguono dalle applicazioni web tradizionali e dalle app native per la loro capacità di operare offline e di accedere a funzionalità native del dispositivo, come la fotocamera e il GPS.

Quando si tratta di sviluppare PWA, diversi linguaggi di programmazione possono essere utilizzati per ottenere risultati efficaci. **Ruby**, ad esempio, è un linguaggio general-purpose noto per la sua semplicità e ampiamente utilizzato per le applicazioni web. Anche se Ruby facilita la scrittura del codice, il processo di debug può essere complesso. **Python**, un altro linguaggio versatile e orientato agli oggetti, offre flessibilità sia per piccole che grandi implementazioni e viene usato comunemente per analisi dei dati e automazione dei compiti. Tuttavia, Python ha

una curva di apprendimento più ripida rispetto ad altri linguaggi.

CSS, o Cascading Style Sheets, è essenziale per descrivere la presentazione di codice scritto in linguaggi di markup come HTML, sebbene non sia sufficiente da solo per creare una PWA. **JavaScript** è spesso la scelta migliore per chi ha esperienza nello sviluppo web, in quanto può essere utilizzato insieme a HTML e CSS per costruire un'applicazione web completa. Con un basso ostacolo all'ingresso, è ideale per chi possiede conoscenze di base in programmazione.

Infine, **PHP**, un linguaggio di scripting general-purpose, è stato introdotto nel 1994 e può essere utilizzato nel backend delle applicazioni. Sebbene PHP non sia la scelta migliore per sviluppare una PWA completa, può essere utile in combinazione con HTML, CSS e JavaScript per il frontend.

In sintesi, le PWA rappresentano una soluzione potente e flessibile per offrire esperienze utente avanzate attraverso il web, sfruttando una varietà di tecnologie e linguaggi di programmazione per soddisfare le esigenze moderne [15] [8].

2.3.4 Applicazioni Cross-platform

Le applicazioni cross-platform vengono sviluppate utilizzando framework che permettono agli ingegneri di costruire applicazioni dall'aspetto nativo per più piattaforme, come Android e iOS, partendo da un'unica base di codice. Uno dei principali vantaggi di questo approccio rispetto allo sviluppo nativo risiede proprio nella possibilità di riutilizzare il codice, riducendo significativamente i tempi di sviluppo.

Per chiarire un dubbio comune tra i giovani imprenditori: lo sviluppo di applicazioni cross-platform non è sinonimo di sviluppo di applicazioni ibride. Mentre lo sviluppo ibrido combina elementi web e nativi, i framework cross-platform sono progettati per creare codice condivisibile e riutilizzabile, che può essere impiegato per costruire applicazioni per diversi sistemi operativi. Questo riduce i costi di sviluppo e lo sforzo richiesto, pur garantendo una produzione funzionale e conveniente.

Tra i framework più popolari per lo sviluppo cross-platform troviamo **React Native**, **Xamarin** e **Flutter**, che hanno reso queste applicazioni molto diffuse. Tuttavia, pur offrendo una soluzione pratica e versatile, le applicazioni

cross-platform possono presentare alcune limitazioni in termini di performance e personalizzazione rispetto alle applicazioni native.

Esempi noti di applicazioni cross-platform includono **Insightly**, **Bloomberg**, **Reflectly**, **Skype**, e **Slack**. Queste applicazioni dimostrano i vantaggi principali di questa tecnologia:

- **Sviluppo rapido e senza intoppi**: Grazie alla riusabilità del codice, gli sviluppatori e i proprietari di imprese possono godere di una maggiore produttività ed efficienza nel lungo termine.
- **Facile manutenzione del prodotto**: Con un'unica base di codice, è più semplice testare, implementare correzioni e aggiornamenti, assicurando applicazioni mobili di alta qualità.
- **Costi ridotti**: Il framework cross-platform permette di raggiungere un mercato più ampio a livello globale con spese iniziali ridotte, risultando particolarmente vantaggioso per le startup.
- **Riuso del codice**: Gli sviluppatori possono trasferire lo stesso codice tra diverse piattaforme senza doverlo riscrivere ogni volta.

Nonostante questi vantaggi, lo sviluppo cross-platform può presentare delle sfide:

- **Ciclo di sviluppo complicato**: La necessità di adattare l'applicazione a più piattaforme può richiedere un team di sviluppatori altamente qualificato, in grado di gestire le differenze tra i sistemi operativi e l'hardware.
- **Integrazioni difficili**: Integrare applicazioni cross-platform con ambienti locali può risultare complesso, specialmente quando si utilizzano approcci di programmazione come il callback-style.

Considera lo sviluppo di un'applicazione cross-platform quando hai bisogno di rilasciare un'app su più piattaforme con un budget, tempo e risorse limitati, specialmente se l'applicazione non è eccessivamente complessa e non richiede funzionalità che variano molto tra le piattaforme [22] [21].

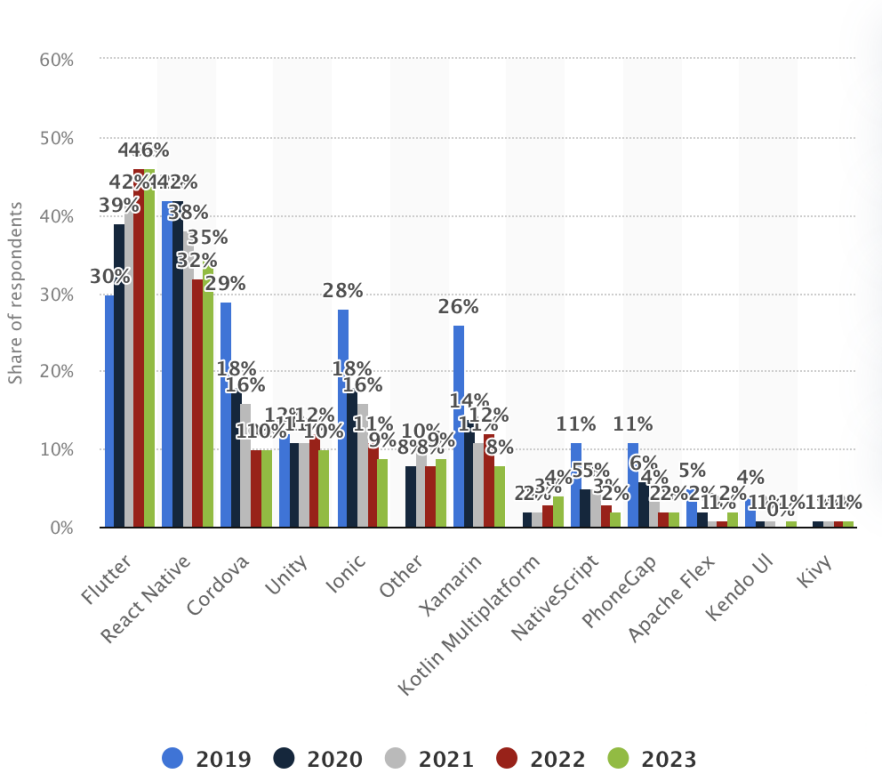


Figure 2.6: Linguaggi cross-platform più utilizzati [2].

2.4 Lo stato dell'arte

Come ci suggerisce Aglowid [3] negli ultimi anni, lo sviluppo di applicazioni mobile cross-platform è diventato sempre più popolare, soprattutto tra imprenditori, start-up e PMI. La crescente domanda di app mobili ha portato alla necessità di ridurre i tempi di sviluppo, favorendo così l'adozione di soluzioni cross-platform come React Native e Flutter, che stanno rapidamente guadagnando terreno. Secondo Statista, Flutter è il framework mobile più popolare del 2024, figura 2.6.

Il dibattito tra sviluppo di app mobile native e cross-platform continua a dividere il mondo della tecnologia. Tuttavia, con l'evoluzione dei sistemi, lo sviluppo cross-platform sembra essere il futuro. Vediamo alcune differenze chiave tra questi approcci:

Fattori	Sviluppo Native	Cross-Platform
Architettura	App diverse per piattaforme diverse	Un'unica app per più piattaforme
Costo	Sviluppo costoso	Costo relativamente basso
Riuso del codice	Singola piattaforma	Codice riutilizzabile
Accesso all'hardware	Accesso completo tramite SDK	Accesso limitato alle API dei dispositivi
UI/UX	Specifica per piattaforma e consistente	UI/UX unificata, ma con limitata consistenza
Performance	Prestazioni native senza compromessi	Alte prestazioni, ma con possibili lag e problemi di compatibilità
Pubblico di riferimento	Limitato a una piattaforma specifica	Pubblico più ampio su più piattaforme
Tempo di sviluppo	Tempi di sviluppo più lunghi	Tempi di sviluppo ridotti
Dimensione del team	Grande (risorse diverse per piattaforme diverse)	Piccolo (una risorsa per tutte le piattaforme)

Table 2.1: Confronto tra sviluppo Native e Cross-Platform

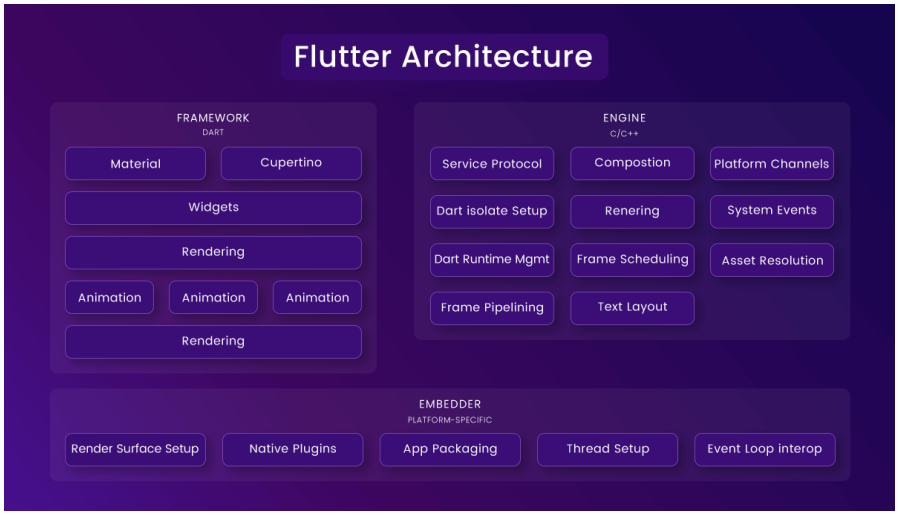


Figure 2.7: Architettura di Flutter [3].

2.4.1 Panoramica su Flutter

Flutter, un framework open-source sviluppato da Google nel 2017, consente lo sviluppo di applicazioni per Android, iOS, macOS, Windows, Linux e Web da un'unica base di codice, utilizzando il linguaggio di programmazione Dart. Flutter è molto apprezzato nella comunità cross-platform per la sua versatilità, figura 2.7.

Secondo il sondaggio annuale di StackOverflow, il 13,55% dei professionisti preferisce lavorare con Flutter. Tuttavia, Flutter e Dart sono relativamente giovani, il che li rende meno maturi e stabili rispetto ad altre tecnologie.

Caratteristiche di Flutter:

- Consente di lavorare con interfacce moderne grazie a una GPU portatile che rende potente l'UI.
- Ideale per sviluppare MVP (Minimum Viable Product) rapidamente e a

basso costo.

- La funzionalità "Hot Reload" permette ai developer di vedere le modifiche al codice in tempo reale.

Progetti famosi in Flutter: Google Ads, eBay, Xianyu di Alibaba.

Pro	**Contro**
Facile da apprendere	DART è un linguaggio meno popolare
Semplice da eseguire il debug	La funzionalità di Hot Reload funziona solo con DART
Completamente compilato	Non tutti i dispositivi sono supportati
Collezione completa di widget	Meno pacchetti di terze parti disponibili
Hot Reload permette modifiche in tempo reale	Non sempre adatto per progetti di grandi dimensioni
Comunità attiva e in crescita	

Table 2.2: Pro e Contro di Flutter

2.4.2 Rassegna delle tecnologie e dei framework correlati

Nel panorama dello sviluppo di applicazioni mobili, oltre a Flutter, esistono numerosi altri framework e tecnologie che hanno avuto un impatto significativo nel settore. Questi strumenti offrono soluzioni diverse per la creazione di applicazioni cross-platform e ognuno di essi presenta caratteristiche uniche che possono soddisfare esigenze specifiche di sviluppo.

In questa sezione, esamineremo alcuni dei framework più rilevanti che, come Flutter, sono progettati per facilitare lo sviluppo di applicazioni per più piattaforme. L'obiettivo è comprendere le peculiarità e i punti di forza di ciascuno di questi strumenti per determinare quale possa essere il più adatto a seconda del tipo di progetto e delle necessità del team di sviluppo.

I framework che verranno analizzati includono:

- React Native: sviluppato da Facebook, è noto per la sua capacità di creare applicazioni mobili con un'interfaccia utente altamente reattiva.
- Ionic: utilizza tecnologie web come HTML, CSS e JavaScript per costruire applicazioni mobili e progressive web apps.
- Xamarin: un framework acquisito da Microsoft che consente di sviluppare applicazioni native utilizzando il linguaggio C#.

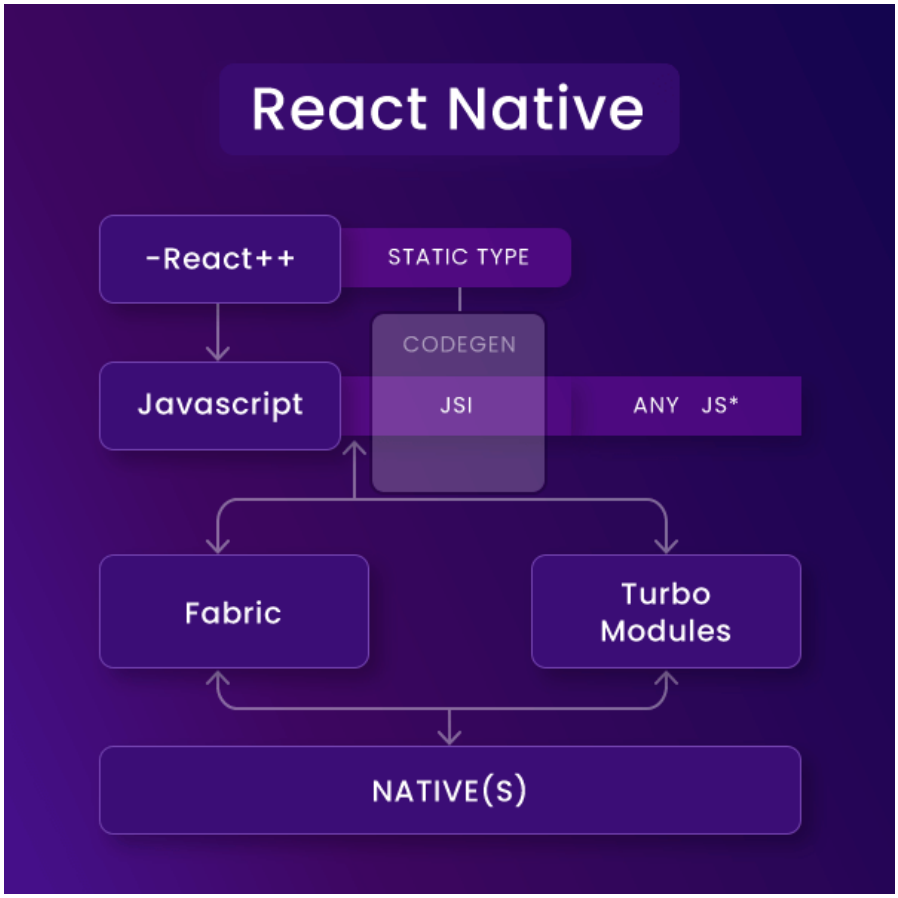


Figure 2.8: Architettura di React Native [3].

Ogni framework ha le proprie caratteristiche distintive, debolezze e punti di forza che andremo ad esaminare.

2.4.2.1 React Native

React Native, lanciato da Facebook nel 2015, utilizza JavaScript e React.js per costruire app cross-platform. È ideale per app semplici, ma può richiedere codice nativo per funzionalità più complesse, figura 2.8.

Caratteristiche di React Native:

- Basato su un codice riutilizzabile per diverse piattaforme.
- Grande compatibilità con plugin di terze parti.

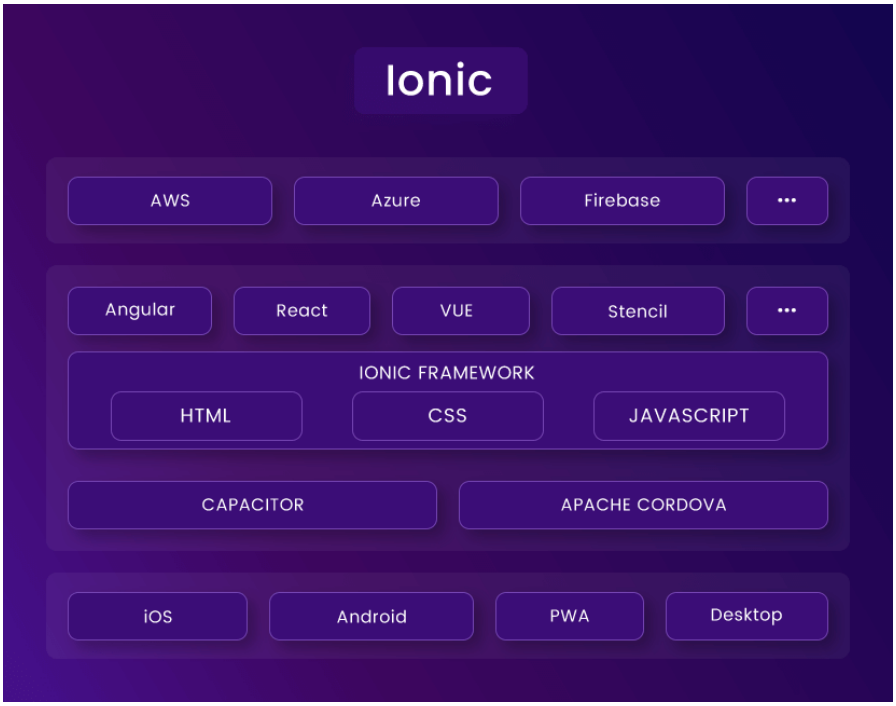


Figure 2.9: Architettura di Ionic [3].

- Focalizzato su UI altamente reattive.

Progetti famosi in React Native: Facebook, Bloomberg, Walmart.

Pro	**Contro**
Ottimo focus sull'interfaccia utente	Non completamente cross-platform
Interfaccia altamente reattiva	Non tutto il codice è riutilizzabile
Varietà di componenti e API integrati	Mancanza di coerenza negli aggiornamenti
Funzione Fast Refresh per vedere modifiche rapide	Ottimizzazione della velocità e della memoria non sempre efficace
Supporto da una comunità ampia	Dipendenza da terze parti

Table 2.3: Pro e Contro di React Native

2.4.2.2 Ionic

Ionic è un framework open-source che utilizza tecnologie web come HTML, CSS e JavaScript per creare app cross-platform, desktop e PWA (Progressive Web Apps). Supporta Angular, React e Vue, figura 2.9.

Caratteristiche di Ionic:

- Cross-platform.

Figure 2.10: Architettura di Xamarin [3].

- Compatibile con diversi framework e plugin Cordova.

Progetti famosi in Ionic: NHS, EA Games, Southwest Airlines.

Pro	Contro
Basato su tecnologie web (Angular, HTML, CSS, JavaScript)	Conoscenza di AngularJS quasi necessaria per app complesse
Compatibilità con React, Vue e Angular	Prestazioni inferiori per app complesse
Ampia gamma di strumenti, plugin e componenti UI	Navigazione complessa a causa di UI complicata
Comunità vivace e documentazione abbondante	Alta dipendenza dai plugin

Table 2.4: Pro e Contro di Ionic

2.4.2.3 Xamarin

Xamarin, acquisito da Microsoft nel 2016, utilizza C# per lo sviluppo di app cross-platform. Offre una forte integrazione con Visual Studio, facilitando la creazione di app native-like, figura 2.10.

Caratteristiche di Xamarin:

- Integrazione con librerie e API native.
- Supporta l'interfaccia utente specifica per piattaforma.

Progetti famosi in Xamarin: UPS, Alaska Airlines, Microsoft News.

Pro	**Contro**
Veramente cross-platform	Le app possono essere più grandi a seconda della complessità
UI specifica per piattaforma	Non raccomandato per app con UI/UX pesanti
Supporta il collegamento con librerie e API native	Accesso limitato a librerie open-source
Comunità forte	Le imprese devono acquistare una licenza Visual Studio

Table 2.5: Pro e Contro di Xamarin

Chapter 3

Il linguaggio Dart

Dart è un linguaggio ottimizzato per lo sviluppo di applicazioni veloci su qualsiasi piattaforma. Il suo obiettivo è offrire il linguaggio di programmazione più produttivo per lo sviluppo multipiattaforma, abbinato a una piattaforma di runtime flessibile per i framework delle applicazioni.

I linguaggi sono definiti dalla loro dotazione tecnica—le scelte fatte durante lo sviluppo che modellano le capacità e i punti di forza di un linguaggio. Dart è progettato per una dotazione tecnica particolarmente adatta allo sviluppo client, dando priorità sia all'esperienza di sviluppo (hot reload state in meno di un secondo) sia a esperienze di produzione di alta qualità su una vasta gamma di target di compilazione (web, mobile e desktop).

Dart costituisce anche la base di Flutter. Dart fornisce il linguaggio e i runtime che alimentano le app Flutter, ma supporta anche molte attività fondamentali per gli sviluppatori come la formattazione, l'analisi e il test del codice [13].

3.1 Introduzione a Dart come linguaggio di programmazione

Il linguaggio Dart è type safe; utilizza il controllo statico dei tipi per garantire che il valore di una variabile corrisponda sempre al tipo statico della variabile stessa. Questo è talvolta indicato come tipizzazione sicura. Sebbene i tipi siano obbligatori, le annotazioni di tipo sono opzionali grazie al type inference. Il sistema

di tipi di Dart è anche flessibile, consentendo l'uso di un tipo dynamic combinato con controlli runtime, che può essere utile durante la sperimentazione o per codice che deve essere particolarmente dinamico.

Dart ha una sicurezza dei null integrata e robusta. Questo significa che i valori non possono essere nulli a meno che non venga specificato. Con la sicurezza dei null robusta, Dart può proteggerti dalle null exceptions a runtime attraverso l'analisi statica del codice. A differenza di molti altri linguaggi null-safe, quando Dart determina che una variabile non può essere nulla, quella variabile non può mai essere nulla. Se ispezioni il tuo codice in esecuzione nel debugger, vedrai che la non-nullabilità è mantenuta a runtime; da qui il termine null-safe robusta.

Il seguente esempio di codice mostra diverse funzionalità del linguaggio Dart, incluse librerie, chiamate asincrone, tipi nullable e non-nullable, sintassi con frecce, generatori, stream e getter [13].

3.2 Principali meccanismi

Piattaforme Dart

Dart offre una tecnologia del compilatore versatile che permette di eseguire il codice su diverse piattaforme:

- **Piattaforma nativa**: Per le app destinate a dispositivi mobili e desktop, Dart include sia una *Virtual Machine (VM)* con compilazione *just-in-time (JIT)* che un compilatore *ahead-of-time (AOT)* per produrre codice macchina.

- **Piattaforma web**: Per le app destinate al web, Dart può compilare codice per scopi sia di sviluppo che di produzione. I compilatori web di Dart traducono il codice Dart in *JavaScript* o *WebAssembly*.

figura 3.1

Il framework **Flutter**, un popolare toolkit UI multipiattaforma basato su Dart, fornisce strumenti e librerie per costruire interfacce utente che funzionano su iOS, Android, macOS, Windows, Linux e web [13].

Dart Native (codice macchina JIT e AOT)

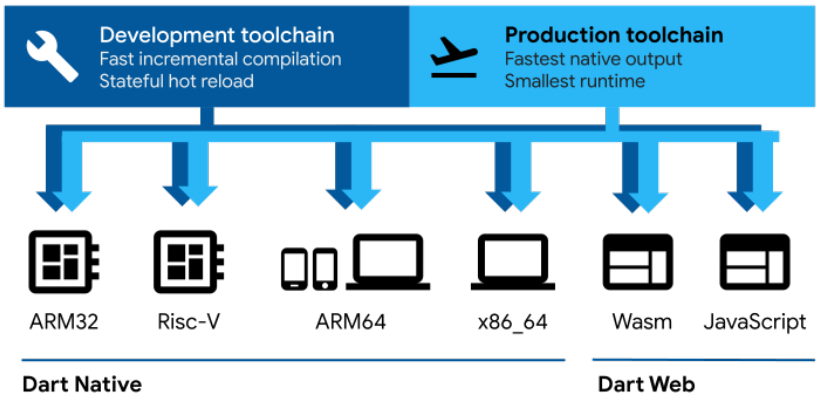


Figure 3.1: Piattaforme Dart [13].

Durante lo sviluppo, un ciclo di iterazione rapido è cruciale. La Dart VM supporta questa necessità grazie al compilatore *JIT* con ricompilazione incrementale, che abilita il *hot reload*. Inoltre, offre raccolte di metriche in tempo reale e un ricco supporto per il debug attraverso *DevTools*.

Quando un'app è pronta per la distribuzione in produzione—sia che venga pubblicata su un app store o distribuita su un backend di produzione—il compilatore *AOT* di Dart può generare codice macchina nativo per architetture ARM o x64. Il codice compilato in AOT garantisce tempi di avvio brevi e consistenti.

Il codice AOT viene eseguito all'interno di un runtime Dart efficiente, che applica il *sound Dart type system* e gestisce la memoria tramite un'allocazione rapida degli oggetti e un *garbage collector* generazionale.

Dart Web (JavaScript dev & prod e WebAssembly)

Dart Web permette di eseguire codice Dart su piattaforme web utilizzando JavaScript. Puoi compilare codice Dart in JavaScript, che viene poi eseguito all'interno di un browser (ad esempio, V8 in Chrome), oppure in WebAssembly.

Dart Web offre tre modalità di compilazione:

- **Compilatore di sviluppo JavaScript incrementale**: Permette un ciclo di sviluppo rapido.
- **Compilatore di produzione JavaScript ottimizzante**: Compila codice

Dart in JavaScript ottimizzato per la produzione, riducendo il codice attraverso tecniche come l'eliminazione del codice morto.

- **Compilatore di produzione WebAssembly ottimizzante (WasmGC)**: Compila codice Dart in WebAssembly GC, producendo un codice molto veloce e distribuibile.

Runtime di Dart

Indipendentemente dalla piattaforma o dal metodo di compilazione utilizzato, l'esecuzione del codice richiede un *runtime Dart*, responsabile di compiti critici come:

- **Gestione della memoria**: Dart utilizza un modello di memoria gestita, con la memoria non utilizzata che viene recuperata da un *garbage collector (GC)*.
- **Applicazione del sistema di tipi Dart**: Sebbene la maggior parte dei controlli di tipo in Dart avvengano a livello statico (durante la compilazione), alcuni controlli vengono eseguiti a runtime, come nel caso degli operatori di controllo e di cast.
- **Gestione degli isolates**: Il runtime gestisce l'isolate principale, dove normalmente viene eseguito il codice, e qualsiasi altro isolate creato dall'applicazione.

Sulle piattaforme native, il runtime Dart è automaticamente incluso all'interno degli eseguibili autosufficienti ed è parte della Dart VM fornita dal comando `dart run`.

3.3 Confronto fra Dart e altri linguaggi

In questa sezione, analizziamo e confrontiamo Dart con tre linguaggi di programmazione: JavaScript, Kotlin e Java. Ogni sottosezione esplorerà le principali differenze e somiglianze, evidenziando le peculiarità e le aree in cui Dart si distingue.

3.3.1 Dart vs JavaScript

In questa sezione, confrontiamo Dart e JavaScript, due linguaggi di programmazione che competono nel contesto dello sviluppo di applicazioni mobili cross-platform.

Dart è stato sviluppato da Google e utilizzato principalmente per costruire applicazioni web, server e mobili. Sebbene Dart esista dal 2011, è diventato più noto con il lancio di Flutter nel 2018, che ha contribuito ad accrescerne la popolarità nel campo dello sviluppo di applicazioni mobili cross-platform. Dart viene compilato sia in codice nativo che in JavaScript, grazie a un ambiente di esecuzione che include una macchina virtuale Dart (Dart VM) e un gestore di pacchetti chiamato Pub.

JavaScript, d'altra parte, è emerso come il linguaggio principale per il rendering delle pagine web ed è evoluto per supportare anche lo sviluppo server-side e mobile. Con l'introduzione di Node.js, JavaScript è diventato una scelta preminente per lo sviluppo sia frontend che backend. Framework come React e React Native hanno ulteriormente consolidato la sua posizione nel panorama dello sviluppo mobile.

Facilità d'uso: JavaScript è un linguaggio maturo e stabilito, con una vasta gamma di framework e librerie disponibili, il che facilita l'accelerazione dello sviluppo. Dart, essendo relativamente nuovo, ha una curva di apprendimento più ripida per i programmatori al di fuori di Google, sebbene la sua sintassi simile a Java possa agevolare i programmatori con esperienza in linguaggi orientati agli oggetti.

Popolarità: JavaScript è onnipresente e rimane uno dei linguaggi più popolari per lo sviluppo di applicazioni web e mobili. Dart, sebbene guadagni terreno grazie a Flutter, non ha ancora raggiunto il livello di diffusione di JavaScript.

Produttività: JavaScript beneficia di una vasta gamma di framework e librerie, che possono accelerare lo sviluppo ma anche comportare un continuo aggiornamento delle competenze. Dart, sebbene ben documentato, ha una comunità più piccola e risorse limitate, il che può rallentare la risoluzione dei problemi.

Curva di apprendimento: JavaScript è generalmente considerato più accessibile grazie alla sua lunga storia e alla disponibilità di risorse di apprendimento.

Dart può essere più difficile da apprendere per i principianti, ma la documentazione di Google e la sintassi simile a Java possono facilitare l'acquisizione da parte di programmatori esperti in altri linguaggi orientati agli oggetti.

Velocità: JavaScript è un linguaggio interpretato e può essere veloce per certe operazioni. Tuttavia, Dart ha dimostrato di essere più veloce in alcuni benchmark rispetto a JavaScript. Dart supporta la compilazione Ahead-of-Time (AOT) e Just-in-Time (JIT), il che può migliorare le prestazioni sia durante lo sviluppo che in fase di rilascio.

Frontend vs Backend: JavaScript è ampiamente utilizzato sia per lo sviluppo frontend che backend, mentre Dart è attualmente utilizzato principalmente per lo sviluppo frontend tramite Flutter, con una presenza meno significativa nel backend.

Sicurezza dei tipi: JavaScript, essendo un linguaggio dinamico, non offre sicurezza dei tipi e gli errori di programmazione possono emergere solo durante l'esecuzione. Dart, con la sua tipizzazione statica, è più sicuro in termini di controllo degli errori durante la compilazione.

Web vs Mobile: JavaScript ha dominato sia lo sviluppo web che mobile grazie ai suoi framework e librerie. Dart, in contrasto, è emerso principalmente come linguaggio per lo sviluppo mobile cross-platform con Flutter, e la sua adozione nel contesto web è meno comune.

Supporto degli editor/IDE: JavaScript è supportato da molti editor e IDE, tra cui strumenti leggeri come VIM e più complessi come WebStorm. Dart è supportato principalmente da IntelliJ IDEA e Android Studio, che offrono plugin specifici per Dart e Flutter.

Uso commerciale: JavaScript è ampiamente utilizzato da grandi aziende come Facebook, Instagram e eBay. Dart, pur essendo utilizzato internamente da Google e adottato da aziende come Alibaba, non ha ancora raggiunto la stessa diffusione commerciale di JavaScript.

In conclusione, sia Dart che JavaScript offrono vantaggi distintivi per lo sviluppo di applicazioni mobili cross-platform. Mentre JavaScript è consolidato e largamente adottato, Dart sta guadagnando popolarità grazie a Flutter e potrebbe rappresentare un'alternativa valida per alcuni scenari di sviluppo [9].

3.3.2 Dart vs Kotlin

Dart e Kotlin sono due linguaggi di programmazione moderni che hanno guadagnato popolarità negli ultimi anni. Dart è principalmente associato a Flutter per lo sviluppo cross-platform, mentre Kotlin è diventato il linguaggio preferito per lo sviluppo Android grazie alla sua modernità e interoperabilità con Java.

3.3.2.1 Panoramica dei Linguaggi

- **Dart**: Creato da Lars Bak e Kasper Lund, è stato rilasciato nel novembre 2013. Ottimizzato per lo sviluppo cross-platform con la versione 2.0, è utilizzato in applicazioni di grandi dimensioni come AdWords.

- **Kotlin**: Sviluppato da Andrey Breslav e supportato da JetBrains, è stato rilasciato a febbraio 2016. È noto per la sua espressività, concisione e interoperabilità con Java.

3.3.2.2 Sistema di Tipi

- **Kotlin**: Tipizzazione statica, sicurezza dei nulli predefinita, senza conversioni implicite non sicure. Supporta il tipo dinamico in Kotlin/JS.

- **Dart**: Tipizzazione statica con sicurezza dei nulli su base solida. Supporta il tipo ‘dynamic‘ per flessibilità a runtime.

3.3.2.3 Sintassi

- **Hello World**:

  – Dart:

Listing 3.1: Hello World in Dart

```dart
void main() {
  print('Hello world!');
}
```

  – Kotlin:

Listing 3.2: Hello World in Kotlin

```kotlin
fun main() {
  println("Hello world!")
}
```

- **Variabili e Costanti**:
  - Dart:

Listing 3.3: Variabili e costanti in Dart

```dart
var name = 'Dart';
final pi = 3.14;
const gravity = 9.8;
```

  - Kotlin:

Listing 3.4: Variabili e costanti in Kotlin

```kotlin
val name: String = "Kotlin" // immutabile
var age: Int = 25 // mutabile
```

- **Stringhe**:
  - Dart:

Listing 3.5: Stringhe in Dart

```dart
String name = 'Dart';
```

  - Kotlin:

Listing 3.6: Stringhe in Kotlin

```kotlin
val name: String = "Kotlin";
```

3.3.2.4 Sicurezza dei Nulli

- **Kotlin**: Gestione predefinita dei nulli con tipi nullable ('String?').
- **Dart**: Sicurezza dei nulli su base solida, 'String?' per tipi nullable.

3.3.2.5 Operazioni Asincrone

- **Kotlin**: Utilizza coroutine per la programmazione asincrona.

Listing 3.7: Esempio di coroutine in Kotlin

```kotlin
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
}
```

- **Dart**: Utilizza ‘Future‘ e ‘async/await‘.

Listing 3.8: Esempio di async/await in Dart

```dart
Future<void> main() async {
  print('Hello,');
  await Future.delayed(Duration(seconds: 1));
  print('World!');
}
```

3.3.2.6 Supporto Cross-Platform

- **Dart**: Eccellente supporto cross-platform con Flutter per mobile, web e desktop.
- **Kotlin**: Kotlin Multiplatform Mobile (KMM) per condividere la logica aziendale tra iOS e Android, con UI specifica per ogni piattaforma.

3.3.2.7 Strumenti

- **CLI**:
  - Kotlin: ‘kotlinc‘, ‘java -jar‘.
  - Dart: ‘dart‘.
- **IDE**:

– Entrambi i linguaggi sono supportati da Android Studio, IntelliJ IDEA e Visual Studio Code.

3.3.2.8 CI/CD

- **Kotlin**: Supporto CI/CD con Jenkins, CircleCI, Travis CI e Codemagic.
- **Dart**: Codemagic fornisce supporto specializzato per le app Flutter e Dart.

3.3.2.9 Conclusioni

Dart e Kotlin offrono caratteristiche forti per i loro ambiti specifici: Dart è eccellente per lo sviluppo cross-platform con Flutter, mentre Kotlin è una scelta potente per lo sviluppo Android e soluzioni multiplatform con KMM. La scelta tra i due dipende dalle esigenze specifiche del progetto e dall'ecosistema esistente [10].

3.3.3 Dart vs Java

Dart: Un Velocista in Flutter

Dart è noto per la sua funzionalità di hot-reload, che accelera notevolmente il processo di sviluppo. Utilizza un compilatore Just-In-Time (JIT) durante lo sviluppo per supportare l'hot-reload e un compilatore Ahead-Of-Time (AOT) per la produzione finale, garantendo codice ottimizzato. In termini di performance, Dart è veloce durante lo sviluppo e competitivo nella fase di distribuzione.

Java: Il Performer Costante

Java, con la sua Virtual Machine (JVM), offre prestazioni consistenti e stabili attraverso diverse piattaforme. Sebbene Java possa non essere il più veloce nel tempo di avvio, la sua ottimizzazione JVM e la gestione della memoria contribuiscono a una performance affidabile e duratura.

3.3.3.1 Scalabilità

Dart: Atleta dell'Equilibrio

Dart supporta una scalabilità forte grazie alla sua architettura modulare e al sistema di pacchetti. Tuttavia, essendo ancora in fase di maturazione, potrebbe mostrare alcune limitazioni in progetti estremamente grandi.

Java: Il Gigante della Scalabilità

Java è ben consolidato nella gestione di grandi progetti grazie alla sua robustezza e alla vasta gamma di framework e strumenti, come Spring, che supportano la scalabilità e il multi-threading.

3.3.3.2 Sintassi

Sintassi Dart

- **Tipizzazione Opzionale**: Dart consente l'annotazione dei tipi, ma ha un sistema di inferenza dei tipi robusto.

Listing 3.9: Tipizzazione Opzionale in Dart

```
int myNumber = 42;
var myString = 'Hello, Dart!';
```

- **Sintassi Concisa**: Dart ha una sintassi moderna e compatta.

Listing 3.10: Sintassi Concisa in Dart

```
int add(int a, int b) => a + b;
class Person {
  String name;
  int age;
  Person(this.name, this.age);
}
```

- **Programmazione Asincrona**: Supporta nativamente async e await.

Listing 3.11: Programmazione Asincrona in Dart

```
Future<void> fetchData() async {
  var data = await fetchDataFromNetwork();
  print(data);
}
```

Sintassi Java

- **Tipizzazione Forte**: Richiede dichiarazioni esplicite dei tipi.

Listing 3.12: Tipizzazione Forte in Java

```
int myNumber = 42;
```

```
2     public int add(int a, int b) {
3       return a + b;
4     }
```

- **Approccio Orientato agli Oggetti**: Tutto è un oggetto e risiede all'interno di classi.

Listing 3.13: Approccio Orientato agli Oggetti in Java

```
1     public class Person {
2       private String name;
3       private int age;
4       public Person(String name, int age) {
5         this.name = name;
6         this.age = age;
7       }
8     }
```

- **Programmazione Asincrona**: Richiede librerie esterne come Future o RxJava.

3.3.3.3 Sicurezza

Sicurezza Dart

- **Sicurezza dei Tipi**: Riduce gli errori di tipo a runtime.
- **Gestione dei Pacchetti**: Pub consente una gestione sicura delle dipendenze.
- **Ambiente di Esecuzione Sicuro**: Isola e gestisce l'esecuzione del codice.

Sicurezza Java

- **Security Manager**: Controlla l'accesso alle risorse di sistema.
- **Esecuzione in Sandbox**: Isola il codice per prevenire azioni pericolose.
- **Gestione della Memoria**: La garbage collection riduce i rischi di vulnerabilità della memoria.

Table 3.1: Confronto tra Dart e Java

Caratteristica	**Dart**	**Java**
Introduzione	2011 da Google	1995 da Sun Microsystems
Scopo	Applicazioni web e mobile	Applicazioni Android, web, enterprise
Sintassi	Moderna e concisa	Verbosa ma strutturata
Frameworks	Flutter	Spring, Android SDK
Tipizzazione	Opzionale con inferenza dei tipi	Statica e rigorosa
Performance	Veloce con hot-reload	Stabile e affidabile
Indipendenza dalla Piattaforma	Compilato per esecuzione cross-platform	Bytecode eseguito su JVM
Curva di Apprendimento	Facile per chi conosce C, C#	Ripida, a causa della sintassi complessa
Strumenti e Librerie	Limitati rispetto a Java	Ampia gamma di strumenti e librerie
Comunità	Piccola ma in crescita	Forte e matura

3.3.3.4 Dart vs Java: Confronto

3.3.3.5 Casi d'Uso

Casi d'Uso di Dart

- **Applicazioni Web**: Eccelle con AngularDart.
- **Sviluppo Mobile con Flutter**: Supporta lo sviluppo cross-platform.
- **Sviluppo Server-Side**: Utilizzabile con Aqueduct per backend scalabili.

Casi d'Uso di Java

- **Applicazioni Enterprise**: Robusto per applicazioni di grande scala.
- **Applicazioni Web**: Utilizza Servlets e JSP.
- **Sviluppo Android**: Compatibile con l'ambiente Android e le API.

3.3.3.6 Conclusione

Sia Dart che Java hanno i loro punti di forza e applicazioni specifiche. Dart è ideale per lo sviluppo rapido e cross-platform, mentre Java eccelle in applicazioni enterprise e sviluppo Android. La scelta tra i due dipende dalle esigenze specifiche del progetto, come performance, scalabilità, sintassi e considerazioni sulla sicurezza [25].

3.4 Sintassi di base e caratteristiche distintive

In Dart, **tutto ciò che può essere assegnato a una variabile è un oggetto**, e ogni oggetto è un'istanza di una classe. Questo principio si applica a numeri, funzioni e anche a `null`. Con l'eccezione di `null` (se si abilita la *sound null safety*), tutti gli oggetti ereditano dalla classe `Object`.

Nota sulla versione: La *null safety* è stata introdotta a partire da Dart 2.12. Per utilizzare questa funzionalità, è necessario usare almeno la versione 2.12 del linguaggio.

Sebbene Dart sia **fortemente tipizzato**, le annotazioni di tipo sono opzionali, poiché Dart può dedurre i tipi. Ad esempio, nell'istruzione `var number = 101`, il tipo di `number` viene dedotto come `int`.

Null Safety: Se abiliti la *null safety*, le variabili non possono contenere `null`, a meno che tu non lo permetta esplicitamente. Per rendere una variabile *nullable*, puoi aggiungere un punto interrogativo (`?`) alla fine del tipo. Ad esempio, una variabile di tipo `int?` può contenere un intero oppure `null`. Se sei sicuro che un'espressione non restituirà mai `null` ma Dart non è d'accordo, puoi usare l'operatore `!` per affermare che il valore non è `null` (in questo caso, se fosse `null`, verrebbe lanciata un'eccezione). Un esempio: `int x = nullableButNotNullInt!;`.

Quando vuoi indicare esplicitamente che è permesso qualsiasi tipo, puoi utilizzare `Object?` (se hai abilitato la *null safety*), `Object`, oppure—se hai necessità di posticipare il controllo del tipo a runtime—il tipo speciale `dynamic`.

Dart supporta **tipi generici**, come `List<int>` (una lista di interi) o `List<Object>` (una lista di oggetti di qualsiasi tipo).

Il linguaggio supporta anche **funzioni a livello globale**, come la funzione principale `main()`, oltre a funzioni legate a una classe o a un oggetto (rispettivamente metodi statici e di istanza). Puoi anche definire **funzioni annidate o locali** all'interno di altre funzioni.

Analogamente, Dart supporta **variabili a livello globale** e variabili legate a una classe o a un oggetto (variabili statiche e di istanza). Le variabili di istanza sono talvolta chiamate *campi* o *proprietà*.

Visibilità degli identificatori: A differenza di Java, Dart non utilizza le

parole chiave `public`, `protected`, e `private`. Se un identificatore inizia con un trattino basso (_), è privato alla sua libreria.

Gli identificatori possono iniziare con una lettera o con un trattino basso (_), seguiti da qualsiasi combinazione di questi caratteri, più le cifre.

Dart distingue tra **espressioni** e **istruzioni**. Le espressioni hanno un valore a runtime, mentre le istruzioni non ne hanno. Ad esempio, l'operatore ternario, è un espressione condizionale del tipo `condition ? expr1 : expr2` restituisce il valore di `expr1` o `expr2`, mentre un'istruzione `if-else` non restituisce un valore. Spesso, un'istruzione contiene una o più espressioni, ma un'espressione non può contenere direttamente un'istruzione.

Gli strumenti di Dart possono segnalare due tipi di problemi: **warning** ed **errori**. I *warning* indicano che il tuo codice potrebbe non funzionare come previsto, ma non impediscono l'esecuzione del programma. Gli *errori*, invece, possono essere di due tipi: **errori di compilazione**, che impediscono l'esecuzione del codice, ed **errori di runtime**, che si manifestano come eccezioni sollevate durante l'esecuzione del programma [12].

3.5 Gestione degli stati in Dart

La gestione degli stati (*state management*) è un aspetto fondamentale nello sviluppo di applicazioni. Essa rappresenta la tecnica di gestione e monitoraggio dello stato di un'applicazione, che può includere interazioni dell'utente, dati e modifiche dell'interfaccia utente (*UI*). In questo capitolo, ci concentreremo sulla gestione degli stati in Dart e Flutter, fornendo una guida approfondita alla comprensione e all'implementazione di queste tecniche nelle applicazioni.

La gestione degli stati in Dart e Flutter riguarda principalmente la gestione dei dati che l'applicazione deve visualizzare e come essa risponde agli input dell'utente. È essenziale comprendere questo concetto per padroneggiare la gestione degli stati, poiché esso rappresenta il "cervello" dell'applicazione, controllando come i dati fluiscono e cambiano nel tempo.

Esistono diverse tecniche per gestire lo stato in Dart e Flutter, tra cui:

- **Provider**: Una combinazione tra *dependency injection* (DI) e gestione degli

stati, progettata con widget per i widget. È un modo potente e flessibile per gestire lo stato in Flutter.

- **Riverpod**: Un'alternativa più flessibile e sicura rispetto a *Provider*, che offre un approccio innovativo alla gestione degli stati.

- **Bloc**: Sigla di *Business Logic Component*, è una libreria che permette una gestione predicibile degli stati. Bloc separa la logica di business dall'interfaccia utente, rendendo il codice più pulito e facilmente testabile.

- **Redux**: Un contenitore di stato predicibile per applicazioni Dart e Flutter, basato sugli stessi principi della libreria Redux in JavaScript.

Ciascuna di queste tecniche presenta vantaggi e svantaggi; la scelta dipende dalle esigenze specifiche e dalla complessità dell'applicazione. È importante comprendere ogni tecnica per poter scegliere quella più adatta al proprio progetto.

Un passaggio cruciale nella gestione degli stati è l'ottimizzazione della tecnica scelta. Questo può includere l'uso di stream per aggiornamenti continui dello stato, l'utilizzo di oggetti stato immutabili per prevenire modifiche indesiderate, e l'adozione del *builder pattern* per ricostruire solo i widget che necessitano di aggiornamento. L'ottimizzazione consiste nel rendere la gestione degli stati il più efficiente ed efficace possibile.

3.5.1 Creazione di un *Stateful Widget*

In Flutter, un widget può essere *stateful* o *stateless*. Un widget *stateful* è dinamico: ad esempio, può cambiare il suo aspetto in risposta a eventi scatenati dalle interazioni dell'utente o quando riceve dati. Per creare un *stateful widget*, è necessario implementare due classi: una sottoclasse di *StatefulWidget* che definisce il widget, e una sottoclasse di *State* che contiene lo stato mutabile del widget e il metodo *build()*.

Il processo di creazione di un *stateful widget* prevede i seguenti passaggi:

1. **Preparazione dell'ambiente**: Configurare l'ambiente di sviluppo e creare una nuova applicazione Flutter.

2. **Gestione dello stato**: Decidere quale oggetto gestirà lo stato del widget. In molti casi, il widget stesso gestirà il proprio stato.

3. **Sottoclasse di *StatefulWidget***: Creare una classe che estende *StatefulWidget* e sovrascrive il metodo *createState()* per creare un oggetto *State*.

4. **Sottoclasse di *State***: Definire una classe che estende *State*, contenente il metodo *build()* che descrive come costruire l'interfaccia utente del widget. Questa classe gestisce anche i dati mutabili del widget.

L'integrazione del *stateful widget* nell'albero dei widget consente di gestire e aggiornare dinamicamente l'interfaccia utente in risposta alle interazioni dell'utente [16][4].

3.5.2 Gestione dello Stato

Esistono diversi approcci alla gestione dello stato in Flutter, e la scelta dipende dall'architettura dell'applicazione e dal comportamento desiderato del widget. I principali approcci includono:

- **Il widget gestisce il proprio stato**: Questo approccio è ideale quando il comportamento del widget è isolato e non influenza altri elementi dell'interfaccia.

- **Il genitore gestisce lo stato del widget**: In questo caso, è il widget genitore che controlla lo stato e aggiorna il widget figlio di conseguenza.

- **Approccio misto**: Una combinazione dei due approcci precedenti, dove alcune parti dello stato sono gestite dal widget stesso, mentre altre sono gestite dal genitore.

La gestione ottimale dello stato consente di mantenere l'interfaccia utente reattiva ed efficiente, migliorando l'esperienza utente e la manutenibilità del codice.

3.5.3 Conclusioni

Padroneggiare la gestione degli stati in Dart e Flutter è una competenza fondamentale per ogni sviluppatore Flutter. Essa permette di creare applicazioni più efficienti, scalabili e manutenibili. Sebbene possa risultare complessa inizialmente, con pratica e comprensione, si può acquisire padronanza di questa tecnica. La chiave per padroneggiare la gestione degli stati consiste nel comprendere il concetto, scegliere la tecnica più adatta, implementarla correttamente, testarla accuratamente e ottimizzarla per ottenere le migliori prestazioni.

Chapter 4

Framework Flutter

Come ci suggerisce la documentazione ufficiale [17], Flutter è un framework open-source sviluppato da Google, concepito per la realizzazione di applicazioni multi-piattaforma belle e performanti, compilate nativamente a partire da un singolo codice sorgente. Questo framework offre una soluzione completa per lo sviluppo di applicazioni per dispositivi mobili, web, desktop e embedded. Grazie a Flutter, gli sviluppatori possono scrivere una sola volta il codice e distribuirlo su più piattaforme, garantendo al contempo prestazioni elevate e un'esperienza utente di alta qualità.

Le caratteristiche distintive di Flutter, come la compilazione nativa del codice, il sistema di *Hot Reload* per aggiornamenti rapidi durante lo sviluppo e la flessibilità nella personalizzazione dell'interfaccia utente, lo rendono uno dei framework più potenti e versatili attualmente disponibili.

4.1 Descrizione di Flutter come framework di sviluppo cross-platform

Flutter si distingue come un framework completo per lo sviluppo *cross-platform*, consentendo agli sviluppatori di creare applicazioni per più piattaforme con un unico codice sorgente scritto in Dart. Tra le piattaforme supportate troviamo dispositivi mobili (iOS e Android), il web, desktop (Windows, macOS, e Linux) e dispositivi embedded.

Uno dei principali vantaggi di Flutter risiede nella **compilazione nativa**, che traduce il codice Dart in codice macchina ARM o Intel, oltre che in JavaScript per le applicazioni web. Questo permette alle app sviluppate con Flutter di essere estremamente performanti su qualsiasi dispositivo, con il pieno supporto per il rendering grafico accelerato hardware.

Il motore di rendering di Flutter permette di **controllare ogni pixel** dello schermo, consentendo la creazione di interfacce utente completamente personalizzabili e adatte a qualsiasi dispositivo, senza dover scendere a compromessi in termini di qualità o prestazioni. La libreria di widget inclusa nel framework adatta automaticamente le interfacce grafiche alle diverse dimensioni dello schermo, garantendo un'esperienza visiva uniforme e fluida su tutte le piattaforme.

Flutter integra al suo interno due librerie grafiche: **Material Design** per le applicazioni Android e **Cupertino** per le applicazioni iOS, rendendo semplice creare applicazioni che rispettano i paradigmi grafici di entrambe le piattaforme, ma senza duplicare il codice. Grazie al concetto di *single codebase*, lo stesso codice può essere utilizzato per sviluppare applicazioni che verranno eseguite nativamente sia su iOS che su Android, riducendo drasticamente i tempi e i costi di sviluppo.

Inoltre, Flutter supporta lo sviluppo per il **web**, permettendo di portare le stesse applicazioni create per mobile anche nei browser web. Questo apre nuove possibilità per il *cross-platform*, in quanto gli utenti possono interagire con l'applicazione direttamente dal browser, senza dover scaricare alcuna app. Il codice Dart può essere compilato in JavaScript, rendendo l'app eseguibile su tutte le moderne piattaforme web.

Per il **desktop**, Flutter fornisce il supporto per la creazione di applicazioni native per Windows, macOS e Linux. Queste applicazioni sono compilate nativamente per ciascuna piattaforma, garantendo prestazioni simili a quelle ottenibili con applicazioni sviluppate specificamente per il desktop, senza dipendere dai motori di rendering dei browser.

4.1.1 Compilazione per più piattaforme

Flutter permette di sviluppare e testare rapidamente su più piattaforme, grazie alla flessibilità offerta dal toolkit di sviluppo. Per esempio, per creare e lanciare

un'applicazione su differenti target, è sufficiente usare pochi comandi:

Listing 4.1: Comandi per creare e lanciare un'applicazione Flutter su diverse piattaforme

```
# Creazione di una nuova applicazione Flutter
flutter create nome_app

# Compilazione per dispositivi mobili
flutter run -d android
flutter run -d ios

# Compilazione per il web (Chrome)
flutter run -d chrome

# Compilazione per desktop (Windows, macOS, Linux)
flutter run -d windows
flutter run -d macos
flutter run -d linux
```

La compilazione nativa permette di ottenere eseguibili ottimizzati per ciascun ambiente, riducendo i tempi di esecuzione e migliorando l'esperienza utente finale.

4.2 Vantaggi nell'utilizzo di Flutter per lo sviluppo mobile

Utilizzare Flutter per lo sviluppo di applicazioni mobile porta numerosi vantaggi rispetto a soluzioni tradizionali:

4.2.1 Compilazione nativa e prestazioni elevate

Flutter compila direttamente in codice macchina ARM o Intel, garantendo un'ottima reattività e prestazioni elevate su dispositivi iOS e Android. Questo approccio riduce la dipendenza da intermediari come WebView, che tendono a rallentare l'esecuzione. Ecco come avviene la compilazione nativa:

Listing 4.2: Compilazione nativa per dispositivi mobili

```
# Compilazione in modalit  di rilascio per Android
flutter build apk --release

# Compilazione in modalit  di rilascio per iOS
```

```
flutter build ios --release
```

Il codice viene compilato in modo efficiente per offrire tempi di caricamento ridotti e un utilizzo ottimizzato delle risorse del dispositivo, garantendo esperienze simili a quelle ottenibili con applicazioni native.

4.2.2 Codice Unico per iOS e Android

Uno dei più grandi benefici è la possibilità di utilizzare un singolo codice sorgente per entrambe le piattaforme principali: iOS e Android. Questo consente di risparmiare risorse significative in termini di tempo e costi, riducendo la necessità di mantenere due team di sviluppo separati e di lavorare su due versioni parallele dell'applicazione. Con Flutter, le applicazioni possono essere rilasciate simultaneamente su entrambe le piattaforme, mantenendo la parità delle funzionalità.

Listing 4.3: Rilascio simultaneo su Android e iOS

```
# Creare la build per entrambe le piattaforme
flutter build apk --release
flutter build ios --release
```

4.2.3 Prestazioni Elevate

Le app sviluppate con Flutter sono **compilate nativamente**, il che significa che offrono prestazioni superiori rispetto ad altre soluzioni *cross-platform* che dipendono da WebView o altri strati di astrazione. Flutter compila il codice Dart direttamente in codice macchina ARM o Intel per le piattaforme mobili, garantendo un'esecuzione fluida e reattiva.

4.2.4 Hot Reload

Durante il processo di sviluppo, Flutter offre la funzionalità di **Hot Reload**, che permette di modificare il codice e visualizzare i cambiamenti quasi istantaneamente, senza perdere lo stato dell'applicazione. Questa caratteristica aumenta notevolmente la produttività degli sviluppatori, permettendo di iterare e testare nuove funzionalità con rapidità.

4.2.5 Design Consistente su Tutte le Piattaforme

Flutter include widget per Material Design e Cupertino, che consentono di creare interfacce grafiche che rispettano i criteri di design di Android e iOS. Questo garantisce che le applicazioni sviluppate siano conformi agli standard grafici delle diverse piattaforme, offrendo al contempo la flessibilità di personalizzare ogni aspetto del design.

4.2.6 Ecosistema di Plugin

Flutter dispone di un ampio ecosistema di plugin disponibili su *pub.dev*, che facilita l'integrazione di funzionalità native nelle applicazioni, come l'accesso alla fotocamera, alla geolocalizzazione, al sistema di notifiche e molto altro. Questo elimina la necessità di scrivere codice specifico per ciascuna piattaforma.

4.2.7 Accesso Anticipato al Mercato

Grazie al *single codebase*, gli sviluppatori possono raggiungere rapidamente un ampio pubblico, distribuendo la loro applicazione su più piattaforme in tempi ridotti. Questo è particolarmente vantaggioso per le startup e per i team di sviluppo più piccoli, che possono lanciare il loro prodotto su Android e iOS contemporaneamente.

4.2.8 Manutenzione Semplificata

Mantenere un'unica base di codice significa che gli aggiornamenti, la correzione di bug e l'introduzione di nuove funzionalità devono essere eseguiti una sola volta, semplificando notevolmente il ciclo di sviluppo e manutenzione. Ciò riduce il rischio di introdurre incongruenze tra le versioni per diverse piattaforme.

In sintesi, Flutter offre una soluzione completa per lo sviluppo di applicazioni mobili *cross-platform*, garantendo velocità, prestazioni elevate, facilità di manutenzione e una forte flessibilità nella creazione di interfacce utente personalizzate.

4.3 Concetto di widget e la sua importanza nella creazione dell'interfaccia utente

I widget sono i mattoni fondamentali nella creazione di interfacce utente in Flutter. Ogni componente dell'interfaccia utente, che sia un pulsante, una casella di testo o un'immagine, è un widget. Flutter fornisce una vasta gamma di widget che possono essere combinati e personalizzati per costruire applicazioni con design ricchi e interattivi.

La costruzione dell'interfaccia utente in Flutter avviene tramite la composizione di widget semplici per costruire widget più complessi. Questo permette agli sviluppatori di creare interfacce modulari e riutilizzabili, facilitando lo sviluppo di applicazioni moderne e interattive. Inoltre, i widget non servono solo per rappresentare elementi visivi, ma anche per gestire il comportamento degli oggetti come il posizionamento, l'allineamento e la disposizione degli elementi.

Un esempio tipico è il widget `Container`, che permette di personalizzare il suo contenuto aggiungendo padding, margini, bordi e colori di sfondo. Anche i widget di layout, come `Row` e `Column`, sono strumenti fondamentali per organizzare e disporre i widget figli all'interno dell'interfaccia.

4.3.1 Allineamento dei Widget

Per controllare l'allineamento dei widget all'interno di una `Row` o `Column`, si utilizzano le proprietà `mainAxisAlignment` e `crossAxisAlignment`. Per una `Row`, l'asse principale è orizzontale e l'asse trasversale è verticale, mentre per una `Column` l'asse principale è verticale e l'asse trasversale è orizzontale.

Le enumerazioni `MainAxisAlignment` e `CrossAxisAlignment` offrono una varietà di costanti per controllare l'allineamento.

4.3.2 Dimensionamento dei Widget

Figure 4.24.1, quando il layout è troppo grande per adattarsi al dispositivo, appare un pattern a strisce giallo e nero lungo il bordo interessato. I widget possono essere dimensionati per adattarsi all'interno di una `Row` o `Column` utilizzando il widget

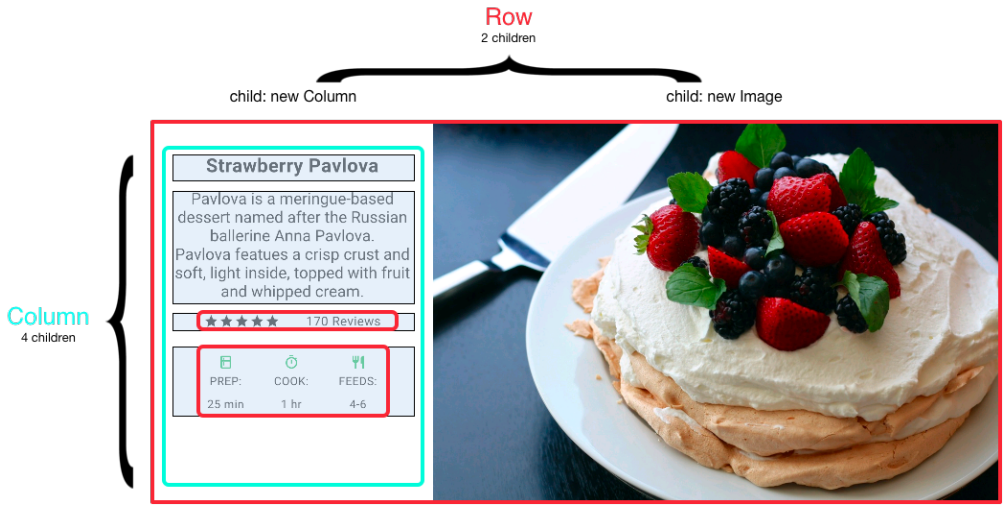


Figure 4.1: Esempio di allineamento per `Row` e `Column` [17]

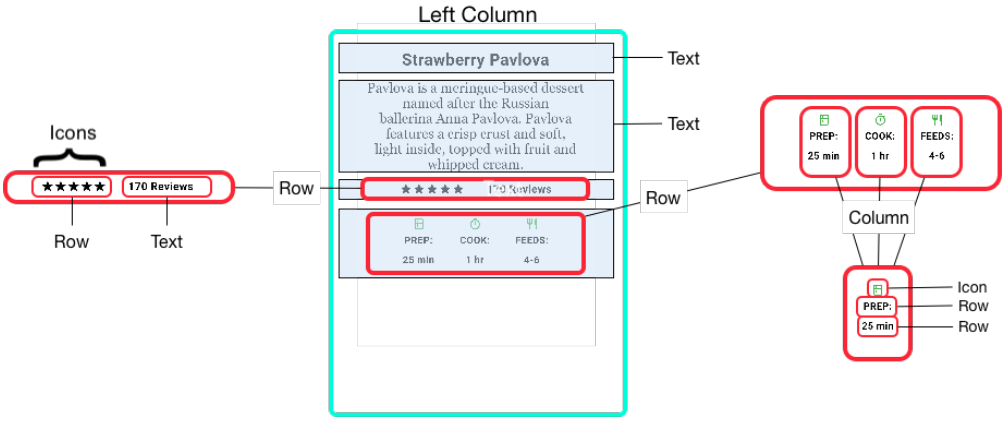


Figure 4.2: Esempio di allineamento per `Row` e `Column` più in dettaglio [17]

`Expanded`.

Esempio di una `Row` con immagini larghe che vengono ridimensionate per adattarsi all'interno del layout:

Listing 4.4: Utilizzo di Row con immagini in Flutter

```
Row(
  crossAxisAlignment: CrossAxisAlignment.center,
  children: [
    Expanded(
      child: Image.asset('images/pic1.jpg'),
    ),
    Expanded(
      child: Image.asset('images/pic2.jpg'),
    ),
    Expanded(
      child: Image.asset('images/pic3.jpg'),
    ),
  ],
);
```

Se si desidera che un widget occupi il doppio dello spazio rispetto ai suoi fratelli, si utilizza la proprietà `flex` del widget `Expanded`.

Listing 4.5: Utilizzo di flex all'interno di Expanded

```
Row(
  crossAxisAlignment: CrossAxisAlignment.center,
  children: [
    Expanded(
      child: Image.asset('images/pic1.jpg'),
    ),
    Expanded(
      flex: 2,
      child: Image.asset('images/pic2.jpg'),
    ),
    Expanded(
      child: Image.asset('images/pic3.jpg'),
    ),
  ],
);
```

4.3.3 Imballaggio dei Widget

Per impostazione predefinita, una `Row` o `Column` occupa tutto lo spazio lungo il proprio asse principale. Se si desidera imballare i figli insieme, è possibile impostare

la proprietà `mainAxisSize` su `MainAxisSize.min`.

Listing 4.6: Utilizzo di `MainAxisSize`: `MainAxisSize.min` all'interno di `Row`

```
Row(
  mainAxisSize: MainAxisSize.min,
  children: [
    Icon(Icons.star, color: Colors.green[500]),
    Icon(Icons.star, color: Colors.green[500]),
    Icon(Icons.star, color: Colors.green[500]),
    const Icon(Icons.star, color: Colors.black),
    const Icon(Icons.star, color: Colors.black),
  ],
);
```

4.3.4 Annidamento di `Row` e `Column`

Il framework di layout consente di annidare `Row` e `Column` all'interno di altre `Row` e `Column` secondo necessità. Questo consente di creare layout complessi come quello dell'esempio Pavlova.

4.3.4.1 Concetto di Constraint

Il concetto di *Constraint* è fondamentale per capire come funzionano i layout in Flutter. A differenza di altri ambienti di sviluppo, Flutter adotta un modello di layout basato su vincoli che scendono lungo l'albero dei widget e dimensioni che risalgono. I vincoli vengono passati dai widget genitori ai figli e includono informazioni sulle dimensioni massime e minime che un widget può assumere.

4.3.5 Flusso del layout: Vincoli, Dimensioni e Posizione

Il flusso del layout in Flutter segue una logica definita:

- **I vincoli scendono:** il genitore impone vincoli ai figli.
- **Le dimensioni salgono:** il figlio sceglie la propria dimensione all'interno dei vincoli.
- **Il genitore imposta la posizione:** una volta che i figli hanno determinato la loro dimensione, il genitore decide la loro posizione.

4.3.5.1 Limitazioni del Sistema di Layout

Il sistema di layout di Flutter, essendo progettato per essere efficiente in una singola passata, presenta alcune limitazioni:

- Un widget può determinare la propria dimensione solo entro i vincoli imposti dal genitore. Questo significa che generalmente un widget non può avere qualsiasi dimensione desideri.
- Un widget non può conoscere o decidere la propria posizione sullo schermo; è il genitore che decide la posizione del widget.
- È impossibile definire con precisione la dimensione e la posizione di un widget senza considerare l'intero albero dei widget.
- Se un figlio desidera una dimensione diversa da quella consentita dal genitore e il genitore non ha abbastanza informazioni per allinearlo, la dimensione desiderata potrebbe essere ignorata.

4.3.6 Esempi Pratici di Vincoli e Layout

Per illustrare come funzionano i vincoli e le dimensioni nei layout di Flutter, consideriamo alcuni esempi:

4.3.6.1 Esempio 1: `Container` senza dimensioni specificate

Listing 4.7: Esempio 1: `Container` senza dimensioni specificate

```
1 Container(color: Colors.red)
```

In questo caso, lo schermo forza il `Container` ad avere le stesse dimensioni dello schermo stesso. Il `Container` riempie quindi l'intero schermo e viene dipinto di rosso.

4.3.6.2 Esempio 2: `Container` con dimensioni specificate

Listing 4.8: Esempio 2: `Container` con dimensioni specificate

```
1 Container(width: 100, height: 100, color: Colors.red)
```

Anche se il `Container` desidera essere largo 100 pixel e alto 100 pixel, lo schermo lo forza ad assumere le dimensioni dello schermo stesso. Di conseguenza, il `Container` riempie l'intero schermo.

4.3.6.3 Esempio 3: Utilizzo di `Center`

Listing 4.9: Esempio 3: `Center` con `Container` di dimensioni specificate

```
Center(
  child: Container(width: 100, height: 100, color: Colors.red),
)
```

Qui, il widget `Center` permette al `Container` di avere qualsiasi dimensione desideri, purché non superi le dimensioni dello schermo. Il `Container` può quindi essere di 100x100 pixel come specificato.

4.3.6.4 Vincoli Stretti, Vincoli Lassi e Vincoli Illimitati

Vincoli Stretti (*Tight Constraints*) Un vincolo è considerato **stretto** quando impone una dimensione esatta al widget figlio. Questo accade quando il valore minimo e massimo per larghezza e altezza sono uguali. Ad esempio:

Listing 4.10: Vincoli Stretti

```
BoxConstraints.tight(Size size)
```

Questo costruttore crea un vincolo in cui `minWidth = maxWidth = size.width` e `minHeight = maxHeight = size.height`.

Vincoli Lassi (*Loose Constraints*) Un vincolo è **lasso** quando consente al widget figlio di essere di qualsiasi dimensione entro un intervallo specificato, tipicamente con un minimo di zero e un massimo definito.

Vincoli Illimitati (*Unbounded Constraints*) I vincoli illimitati si verificano quando il valore massimo per la larghezza o l'altezza è infinito (`double.infinity`). In tali casi, i widget che cercano di essere il più grandi possibile possono causare errori, poiché non è possibile renderizzare dimensioni infinite.

4.3.6.5 Widget `Row` e `Column` con Vincoli

I widget `Row` e `Column` si comportano diversamente a seconda dei vincoli ricevuti:

- Con vincoli limitati, cercano di occupare tutto lo spazio disponibile nel loro asse principale.
- Con vincoli illimitati, dimensionano i loro figli in base al contenuto, e i figli con `Expanded` o `Flexible` possono causare errori.

4.3.7 Widget Specifici e il Loro Comportamento con i Vincoli

4.3.7.1 `FittedBox`

Il `FittedBox` è un widget che scala e adatta il suo figlio per riempire lo spazio disponibile, rispettando le proporzioni. È utile quando si desidera che un widget si adatti esattamente alle dimensioni del suo contenitore.

Listing 4.11: Esempio di `FittedBox`

```
FittedBox(
  child: Text('Esempio di testo.'),
)
```

In questo esempio, il `FittedBox` ridimensiona il testo in modo che riempia lo spazio disponibile.

4.3.7.2 `Expanded` e `Flexible`

All'interno di widget come `Row` e `Column`, `Expanded` e `Flexible` sono utilizzati per controllare come i widget figli occupano lo spazio disponibile.

- `Expanded`: forza il figlio a riempire lo spazio disponibile nel widget genitore.
- `Flexible`: consente al figlio di riempire lo spazio disponibile, ma il figlio può anche avere dimensioni inferiori se lo desidera.

4.3.7.3 Esempio con `Row` e `Expanded`

Listing 4.12: Esempio di `Row` con `Expanded`

```
Row(
  children: [
    Expanded(
      child: Container(
        color: Colors.red,
        child: Text(
          'Testo lungo che non entra in una sola linea.',
          style: TextStyle(fontSize: 20),
        ),
      ),
    ),
    Expanded(
      child: Container(
        color: Colors.green,
        child: Text(
          'Ciao!',
          style: TextStyle(fontSize: 20),
        ),
      ),
    ),
  ],
)
```

In questo esempio, entrambi i `Container` vengono espansi per occupare lo spazio disponibile in modo proporzionale.

4.3.8 Esempi di Layout in Flutter

Per comprendere meglio i concetti di vincoli e dimensioni, è utile considerare vari esempi di layout, come l'utilizzo di `Container`, `Center`, `Align`, e altro. Questi esempi dimostrano come i widget si adattano ai vincoli imposti dai genitori.

4.3.9 State Widget

Oltre agli widget per definire il layout esistono altri tipi fondamentali di widget fondamentali per definire lo stato dell'interfaccia utente,che rappresenta una parte dell'interfaccia grafica. Questi widget sono classificati in due tipi principali: *stateless* e *stateful*.

4.3.9.1 Differenze tra widget stateless e stateful

I *stateless widget* sono widget che non cambiano il loro stato una volta costruiti, mentre i *stateful widget* possono modificare il loro stato in risposta all'interazione dell'utente o ad altri eventi. Alcuni esempi includono `Icon`, `IconButton`, e `Text` per i *stateless*, mentre `Checkbox`, `Radio`, `Slider`, e `TextField` per i *stateful*.

4.3.9.2 Importanza dei widget nell'interfaccia utente

I widget svolgono un ruolo cruciale nella costruzione di interfacce modulari, scalabili e interattive. Permettono agli sviluppatori di gestire il comportamento e l'aspetto dell'applicazione in modo separato, garantendo riusabilità e mantenibilità del codice.

4.4 Widget principali

Flutter offre una ricca libreria di widget di layout. Ecco alcuni dei più comuni:

4.4.1 Container

Il widget `Container` è spesso utilizzato per aggiungere padding, margini, bordi e colore di sfondo ai widget. Contiene un singolo widget figlio, che può essere un `Row`, `Column` o anche la radice di un albero di widget.

Listing 4.13: Utilizzo di Container

```
Widget _buildImageColumn() {
  return Container(
    decoration: const BoxDecoration(
      color: Colors.black26,
    ),
    child: Column(
      children: [
        _buildImageRow(1),
        _buildImageRow(3),
      ],
    ),
  );
}
```

Nel caso si fosse interessati esclusivamente ad aggiungere padding e margini è più consigliato l'uso di `SizedBox`.

4.4.2 GridView

Il widget `GridView` dispone i widget in una griglia a due dimensioni. Quando i contenuti superano il box di rendering, `GridView` fornisce automaticamente la possibilità di scorrere.

Listing 4.14: Utilizzo di GridView

```
Widget _buildGrid() => GridView.extent(
    maxCrossAxisExtent: 150,
    padding: const EdgeInsets.all(4),
    mainAxisSpacing: 4,
    crossAxisSpacing: 4,
    children: _buildGridTileList(30));
```

4.4.3 ListView

Il widget `ListView` è simile a una colonna e fornisce automaticamente lo scorrimento quando i suoi contenuti sono troppo lunghi.

Listing 4.15: Utilizzo di ListView

```
Widget _buildList() {
  return ListView(
    children: [
      _tile('CineArts at the Empire', '85 W Portal Ave', Icons.theaters),
      _tile('The Castro Theater', '429 Castro St', Icons.theaters),
      // altre righe
    ],
  );
}
```

4.4.4 Stack

Il widget `Stack` è utilizzato per sovrapporre un widget su un altro. Può essere usato per creare effetti visivi complessi.

Listing 4.16: Utilizzo di Stack

```
Widget _buildStack() {
  return Stack(
    alignment: const Alignment(0.6, 0.6),
    children: [
      const CircleAvatar(
        backgroundImage: AssetImage('images/pic.jpg'),
        radius: 100,
      ),
      Container(
        decoration: const BoxDecoration(
          color: Colors.black45,
        ),
        child: const Text(
          'Mia B',
          style: TextStyle(
            fontSize: 20,
            fontWeight: FontWeight.bold,
            color: Colors.white,
          ),
        ),
      ),
    ],
  );
}
```

4.4.5 Card

Il widget `Card` contiene informazioni correlate in una scatola con angoli arrotondati e un'ombra.

Listing 4.17: Utilizzo di Card

```
Widget _buildCard() {
  return SizedBox(
    height: 210,
    child: Card(
      child: Column(
        children: [
          ListTile(
            title: const Text(
              '1625 Main Street',
              style: TextStyle(fontWeight: FontWeight.w500),
            ),
            subtitle: const Text('My City, CA 99984'),
            leading: Icon(
              Icons.restaurant_menu,
              color: Colors.blue[500],
```

```
            ),
          ),
          const Divider(),
          // altre ListTile
        ],
      ),
    ),
  );
}
```

4.4.6 ListTile

Il widget `ListTile` è utilizzato per creare una riga contenente fino a 3 linee di testo e icone opzionali.

Listing 4.18: Utilizzo di ListTitle

```
ListTile _tile(String title, String subtitle, IconData icon) {
  return ListTile(
    title: Text(title,
        style: const TextStyle(
          fontWeight: FontWeight.w500,
          fontSize: 20,
        )),
    subtitle: Text(subtitle),
    leading: Icon(
      icon,
      color: Colors.blue[500],
    ),
  );
}
```

4.4.7 Widget di Layout Avanzati

4.4.7.1 `LimitedBox`

Il widget `LimitedBox` impone vincoli solo se i vincoli ricevuti dal genitore sono illimitati. È utile quando si utilizza un widget in un contesto con vincoli illimitati, come all'interno di uno `UnconstrainedBox`.

Listing 4.19: Esempio di `LimitedBox` con `UnconstrainedBox`

```
UnconstrainedBox(
  child: LimitedBox(
```

```
    maxWidth: 100,
    child: Container(color: Colors.red),
  ),
)
```

In questo caso, se `UnconstrainedBox` fornisce vincoli illimitati, `LimitedBox` applicherà un `maxWidth` di 100 al suo figlio.

4.4.7.2 `OverflowBox`

Il widget `OverflowBox` consente al figlio di eccedere i vincoli del genitore senza generare errori di overflow.

Listing 4.20: Esempio di `OverflowBox`

```
OverflowBox(
  minWidth: 0,
  minHeight: 0,
  maxWidth: double.infinity,
  maxHeight: double.infinity,
  child: Container(color: Colors.red, width: 4000, height: 50),
)
```

In questo esempio, il `Container` può avere una larghezza maggiore rispetto allo spazio disponibile senza causare errori.

4.4.8 Widget interattivi e non interattivi

Un widget può essere interattivo o non interattivo. I widget interattivi, come i pulsanti e i campi di testo, reagiscono agli input dell'utente, mentre i widget non interattivi, come le immagini o i testi statici, non cambiano in base all'interazione.

4.4.9 Widget interattivi più utilizzati

Tra i widget interattivi più comuni troviamo:

- **IconButton**: Utilizzato per creare pulsanti con icone che rispondono a un evento di pressione.
- **TextField**: Un campo di input testo, che permette agli utenti di inserire dati.

- **Slider**: Permette di selezionare un valore da una gamma predefinita.
- **Checkbox**: Un controllo che permette di selezionare o deselezionare un'opzione.

4.4.10 Creazione di un *stateful widget*

Il processo di creazione di un *stateful widget* richiede due classi principali:

- La sottoclasse di `StatefulWidget` che definisce il widget.
- La sottoclasse di `State` che gestisce lo stato del widget e la sua logica di aggiornamento.

Di seguito è riportato un esempio di un widget personalizzato che gestisce una stella e un contatore di preferiti:

Listing 4.21: Utilizzo di `StatefulWidget`

```
class FavoriteWidget extends StatefulWidget {
  @override
  _FavoriteWidgetState createState() => _FavoriteWidgetState();
}

class _FavoriteWidgetState extends State<FavoriteWidget> {
  bool _isFavorited = true;
  int _favoriteCount = 41;

  @override
  Widget build(BuildContext context) {
    return Row(
      children: [
        IconButton(
          icon: (_isFavorited ? Icon(Icons.star) : Icon(Icons.star_border)),
          color: Colors.red[500],
          onPressed: _toggleFavorite,
        ),
        Text('$_favoriteCount'),
      ],
    );
  }

  void _toggleFavorite() {
    setState(() {
      if (_isFavorited) {
        _favoriteCount -= 1;
```

```
28          _isFavorited = false;
29        } else {
30          _favoriteCount += 1;
31          _isFavorited = true;
32        }
33      });
34    }
35  }
```

4.4.11 Gestione dello stato nei widget

La gestione dello stato è una parte cruciale nello sviluppo di widget interattivi. Esistono vari approcci per la gestione dello stato:

- **Gestione autonoma**: Il widget gestisce il proprio stato internamente.
- **Gestione da parte del genitore**: Il genitore del widget controlla il suo stato e gli passa le informazioni.
- **Approccio ibrido**: Una combinazione in cui alcune parti dello stato sono gestite dal widget e altre dal genitore.

Chapter 5

Firebase come Database per l'E-commerce

In un mondo digitale sempre più competitivo e orientato al mobile, la scelta di una piattaforma affidabile per lo sviluppo di applicazioni e la gestione dei dati è cruciale per il successo di qualsiasi progetto di e-commerce. Firebase, piattaforma di sviluppo mobile di Google, rappresenta una soluzione completa che offre strumenti integrati per lo sviluppo, la gestione e l'ottimizzazione delle applicazioni, con un focus particolare sulla sincronizzazione in tempo reale, la sicurezza e l'integrazione con funzionalità di Intelligenza Artificiale (AI).

Firebase non è solo un database, ma un ecosistema che permette di costruire, gestire e migliorare applicazioni mobile e web in modo efficiente e scalabile. Dal database in tempo reale fino all'integrazione con Google Cloud, Firebase si è affermato come una delle soluzioni più utilizzate e apprezzate dagli sviluppatori di tutto il mondo. In questo capitolo verranno esplorati i vari aspetti che rendono Firebase una scelta ideale per un progetto di e-commerce, analizzando i vantaggi e le caratteristiche specifiche che questa piattaforma offre.

Questo capitolo non solo esplorerà gli strumenti di Firebase per la gestione del database, ma approfondirà anche il suo ecosistema, con un'attenzione particolare alle soluzioni che rendono possibile la costruzione di applicazioni di alta qualità e scalabili. Si analizzeranno inoltre le motivazioni che portano alla scelta di Firebase come soluzione ideale per la gestione dei dati in un progetto di e-commerce in

Flutter, mettendo in luce i vantaggi pratici, l'integrazione con altre tecnologie e le sue funzionalità avanzate come la gestione in tempo reale e l'integrazione con AI.

5.1 Introduzione a Firebase come piattaforma di sviluppo mobile di Google

Firebase è una piattaforma di sviluppo completa offerta da Google, creata per facilitare la realizzazione di applicazioni mobili e web. Con Firebase, gli sviluppatori possono costruire app moderne e scalabili, beneficiando dell'infrastruttura di Google Cloud, che garantisce prestazioni elevate, sicurezza e affidabilità a livello globale.

Firebase fornisce una suite di strumenti progettati per ogni fase del ciclo di vita di sviluppo dell'applicazione, partendo dalla fase di creazione (*build*) fino alla distribuzione e al monitoraggio (*run*) delle applicazioni. Grazie a strumenti avanzati di analisi, monitoraggio delle prestazioni e gestione delle versioni, Firebase offre una soluzione ideale per le app che devono crescere e adattarsi in ambienti dinamici e competitivi come quello dell'e-commerce.

Tra le sue funzionalità principali ci sono:

- **Real-time Database e Firestore**: database NoSQL che supportano la sincronizzazione in tempo reale, consentendo alle app di riflettere immediatamente i cambiamenti sui dati.
- **Cloud Functions**: funzioni serverless che permettono di eseguire codice lato server in risposta a eventi senza dover gestire un'infrastruttura.
- **Firebase Hosting**: una soluzione di hosting per app web statiche e dinamiche.
- **Authentication**: un sistema di autenticazione completo e sicuro che supporta provider di terze parti come Google, Facebook e Twitter.
- **Firebase Machine Learning**: un set di strumenti che facilita l'integrazione di modelli di machine learning nelle app.

Firebase è stato pensato per aiutare gli sviluppatori a creare app che non solo funzionino bene, ma che siano anche sicure, scalabili e in grado di crescere insieme al business.

5.1.1 Integrazione con Google Cloud e intelligenza artificiale

Uno dei principali punti di forza di Firebase è l'integrazione nativa con Google Cloud e, più recentemente, con le tecnologie di intelligenza artificiale, come la suite AI Gemini. Questo permette di costruire esperienze AI-powered direttamente nelle app con strumenti predefiniti e facilmente implementabili.

Le API di *Vertex AI*, ad esempio, consentono agli sviluppatori di includere capacità avanzate come la generazione di testo, immagini, video e audio all'interno delle applicazioni, accelerando così l'introduzione di nuove funzionalità interattive basate sull'intelligenza artificiale.

5.2 Motivazioni dietro la scelta di Firebase come soluzione per la gestione del database

Firebase è una piattaforma di sviluppo mobile e web offerta da Google che fornisce una serie di servizi backend per supportare lo sviluppo di applicazioni moderne. Come suggerito da [7][6], le motivazioni per scegliere Firebase come soluzione per la gestione del database includono:

5.2.1 Scalabilità e Affidabilità

Firebase è progettato per scalare automaticamente in risposta al carico di lavoro, permettendo alle applicazioni di gestire variazioni di traffico senza la necessità di intervento manuale. Utilizzando l'infrastruttura di Google Cloud, Firebase garantisce che le applicazioni possano crescere senza compromessi in termini di prestazioni e disponibilità. La scalabilità è particolarmente importante per le applicazioni e-commerce che possono sperimentare picchi di traffico durante le vendite speciali o le festività.

5.2.2 Real-Time Data Synchronization

Uno degli aspetti distintivi di Firebase è il suo Realtime Database, un database NoSQL che memorizza i dati come JSON e li sincronizza in tempo reale su tutti i client connessi. Questa capacità è cruciale per le applicazioni e-commerce, poiché consente aggiornamenti immediati delle informazioni sui prodotti, delle disponibilità e degli ordini. Ad esempio, se un prodotto viene acquistato, tutte le istanze dell'applicazione vengono aggiornate istantaneamente per riflettere la nuova disponibilità del prodotto.

5.2.3 Facilità di Implementazione

Firebase offre una serie di SDK e strumenti di integrazione che semplificano la configurazione e la gestione del backend. Gli sviluppatori possono concentrarsi sulla logica dell'applicazione e sull'interfaccia utente piuttosto che sulla gestione complessa dell'infrastruttura server. Questa facilità di implementazione è particolarmente vantaggiosa per le startup e le piccole imprese che necessitano di soluzioni rapide ed economiche.

5.2.4 Integrazione con Google Cloud

Firebase si integra perfettamente con altri servizi di Google Cloud, come Cloud Firestore e Cloud Functions. Questa integrazione consente agli sviluppatori di utilizzare una suite completa di strumenti per estendere le capacità delle loro applicazioni, migliorando le performance e ottimizzando le operazioni. Ad esempio, l'integrazione con Google Analytics fornisce insight dettagliati sul comportamento degli utenti e sull'efficacia delle campagne di marketing, figura 5.1.

5.3 Analisi comparativa di Firebase con altri possibili database o soluzioni di back-end

In questa sezione verrà effettuata una comparazione di Firebase con altre soluzioni di back-end popolari, come AWS, Heroku, Azure, Parse, Hoodie, Back4App e

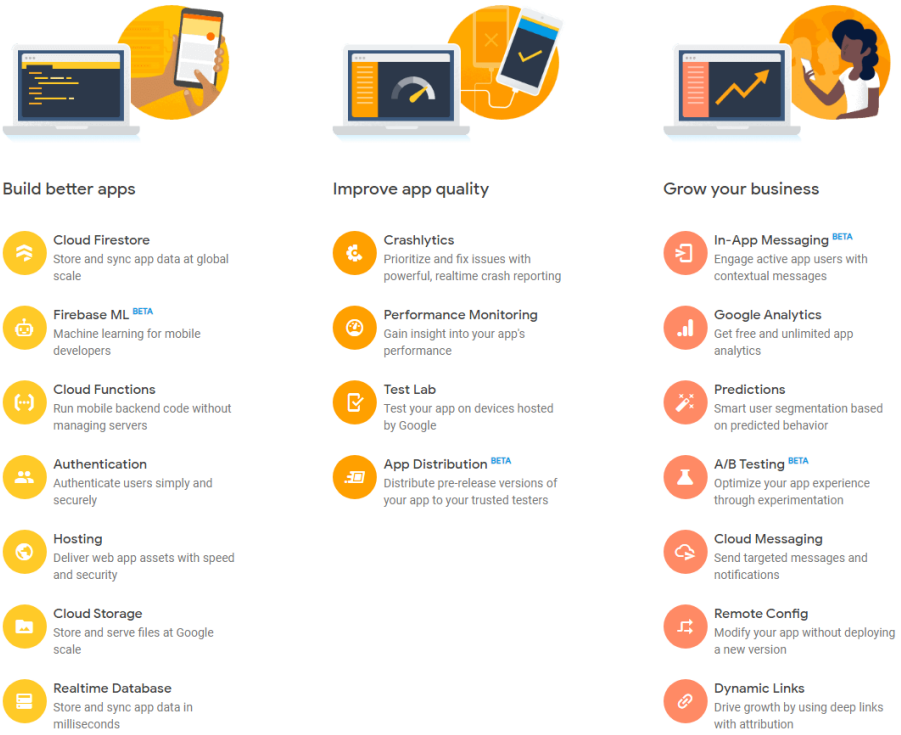


Figure 5.1: Potenzialità di Firebase [24].

CloudKit. L'obiettivo è fornire un quadro generale che evidenzi le caratteristiche principali, i vantaggi e gli svantaggi di ogni piattaforma, per consentire una scelta informata per lo sviluppo di applicazioni e-commerce.

Firebase offre funzionalità come analisi, database, comunicazione, notifiche di crash, e altro ancora. Ti consente di concentrarti sui tuoi clienti. Firebase è progettato e scalato sulle risorse di Google, anche per le applicazioni più grandi. I prodotti Firebase funzionano bene sia con Android che con iOS. Firebase scambia informazioni e approfondimenti, permettendo una migliore integrazione tra i suoi servizi [24].

5.3.1 Firebase vs AWS

Firebase e AWS (Amazon Web Services) sono due soluzioni che eliminano la necessità di gestione del server, adottando un modello *serverless*. Entrambe le piattaforme consentono agli sviluppatori di focalizzarsi sulla logica dell'applicazione,

senza preoccuparsi dell'infrastruttura sottostante.

Differenze principali:

- **Velocità e costo**: Firebase tende ad essere più veloce e meno costoso di AWS, in particolare per progetti di piccole e medie dimensioni.
- **Open-source**: AWS offre un database open-source, mentre Firebase è una soluzione proprietaria di Google.

Pro di AWS:

- Eccellente performance anche con carichi di lavoro elevati.
- Facile da configurare e iniziare, specialmente per chi ha già esperienza con cloud computing.

Contro di AWS:

- Costo dipendente dal traffico, che può aumentare rapidamente se non gestito correttamente.
- Modello di pagamento *pay-per-use*, che può diventare costoso in caso di carichi elevati.

5.3.2 Firebase vs Heroku

Heroku è una piattaforma di cloud computing che permette agli sviluppatori di implementare, gestire e scalare le applicazioni con facilità. A differenza di Firebase, Heroku supporta diversi database come MongoDB e MySQL, offrendo maggiore flessibilità dal punto di vista del database.

Differenze principali:

- **Facilità di configurazione del backend**: Firebase semplifica maggiormente la configurazione del backend rispetto a Heroku.
- **Log files**: Heroku fornisce strumenti più potenti per l'accesso e la gestione dei file di log.

Pro di Heroku:

- Facilità di rollback delle applicazioni.
- Integrazione semplice con altri servizi SaaS (Software as a Service).

Contro di Heroku:

- Non offre indirizzi IP statici.
- Il database di Heroku non è ottimizzato per l'estrazione di file di log complessi.

5.3.3 Firebase vs Azure

Azure, il cloud di Microsoft, è una piattaforma PaaS (Platform as a Service) che si distingue da Firebase per l'uso di database relazionali compatibili con SQL Server. Azure è indicato per progetti che richiedono una scalabilità avanzata e una gestione complessa delle risorse.

Differenze principali:

- **Categoria di servizio**: Azure è una PaaS, mentre Firebase è una BaaS (Backend as a Service).
- **Scalabilità**: Azure tende a scalare meglio rispetto a Firebase in ambienti enterprise.

Pro di Azure:

- Sicurezza informatica avanzata.
- Offre un modello di pagamento *pay-as-you-go*, permettendo di pagare solo per le risorse effettivamente utilizzate.

Contro di Azure:

- Richiede una gestione più complessa.
- È necessario un alto livello di competenza per ottimizzare la piattaforma.

5.3.4 Firebase vs Parse Server

Parse Server è una piattaforma open-source che consente agli sviluppatori di auto-ospitare il proprio backend. In confronto a Firebase, che utilizza Google Cloud, Parse Server può essere eseguito su qualsiasi cloud, offrendo maggiore flessibilità.

Differenze principali:

- **Cloud**: Firebase utilizza Google Cloud, mentre Parse può essere eseguito su qualsiasi piattaforma cloud.
- **Geo Queries**: Firebase non supporta le geo-query, una funzionalità invece supportata da Parse.

Pro di Parse Server:

- Facile da implementare.
- Consente l'uso di diversi database e file system.

Contro di Parse Server:

- Configurazione locale complessa.
- La distribuzione dell'applicazione richiede sforzi aggiuntivi.

5.3.5 Firebase vs Hoodie

Hoodie è una piattaforma open-source che gestisce attività di backend, ma non è progettata per la scalabilità avanzata come Firebase. Offre un interessante sistema di sincronizzazione offline, ma non consente il controllo diretto sui servizi di backend.

Differenze principali:

- **Data storage**: Hoodie memorizza i dati localmente e li sincronizza quando possibile, mentre Firebase memorizza i dati centralmente su Google Cloud.
- **Open-source**: Hoodie è open-source, mentre Firebase no.

Pro di Hoodie:

- Facilità di scrittura di plugin.
- Sincronizzazione offline per i client.

Contro di Hoodie:

- Nessun controllo sui servizi di backend.

5.3.6 Firebase vs Back4App

Back4App è un servizio SaaS che offre molte delle funzionalità di Parse Server con aggiunte come il monitoraggio delle prestazioni e l'infrastruttura pronta all'uso. Firebase invece è una piattaforma BaaS chiusa.

Differenze principali:

- **Open-source vs Closed**: Back4App è open-source, mentre Firebase è un sistema chiuso.
- **Tipologia di servizio**: Back4App è un SaaS, mentre Firebase è un BaaS.

Pro di Back4App:

- Database facili da usare, simili a fogli di calcolo.
- Integrazione con servizi di caching.
- Supporto clienti 24/7.

Contro di Back4App:

- Necessità di utilizzare database auto-ospitati.
- Maggiore complessità nella gestione del backend rispetto a Firebase.

5.3.7 Firebase vs CloudKit

CloudKit è una soluzione Apple che sincronizza i dati tra dispositivi tramite iCloud. È particolarmente indicata per le app che devono sincronizzare dati in modo sicuro tra dispositivi appartenenti a un singolo utente.

Differenze principali:

- **Database**: CloudKit utilizza un database relazionale tradizionale, mentre Firebase è basato su un database NoSQL.
- **Autenticazione**: Firebase supporta l'autenticazione tramite terze parti (Google, Facebook, ecc.), mentre CloudKit richiede un account Apple.

Pro di CloudKit:

- Autenticazione automatica e sicura.
- Servizio gratuito.
- Sincronizzazione tra dispositivi Apple.

Contro di CloudKit:

- Oggetti non nidificabili all'interno di altri oggetti.
- Non supporta login di terze parti come Facebook o Google.

5.3.8 Conclusione

Firebase è una piattaforma altamente efficiente per lo sviluppo di backend, ma esistono diverse alternative con caratteristiche e vantaggi specifici. La scelta della soluzione migliore dipende dalle esigenze specifiche del progetto, dai costi e dalla scalabilità desiderata. È consigliabile considerare tutte le opzioni disponibili e scegliere quella che meglio si adatta ai requisiti e agli obiettivi del proprio progetto, figura 5.2.

5.4 Vantaggi specifici di Firebase nel contesto dell'e-commerce

Nel contesto dell'e-commerce, come suggerisce [6], Firebase offre vantaggi specifici che possono migliorare l'efficienza e l'efficacia delle applicazioni di vendita online. Questi vantaggi includono:

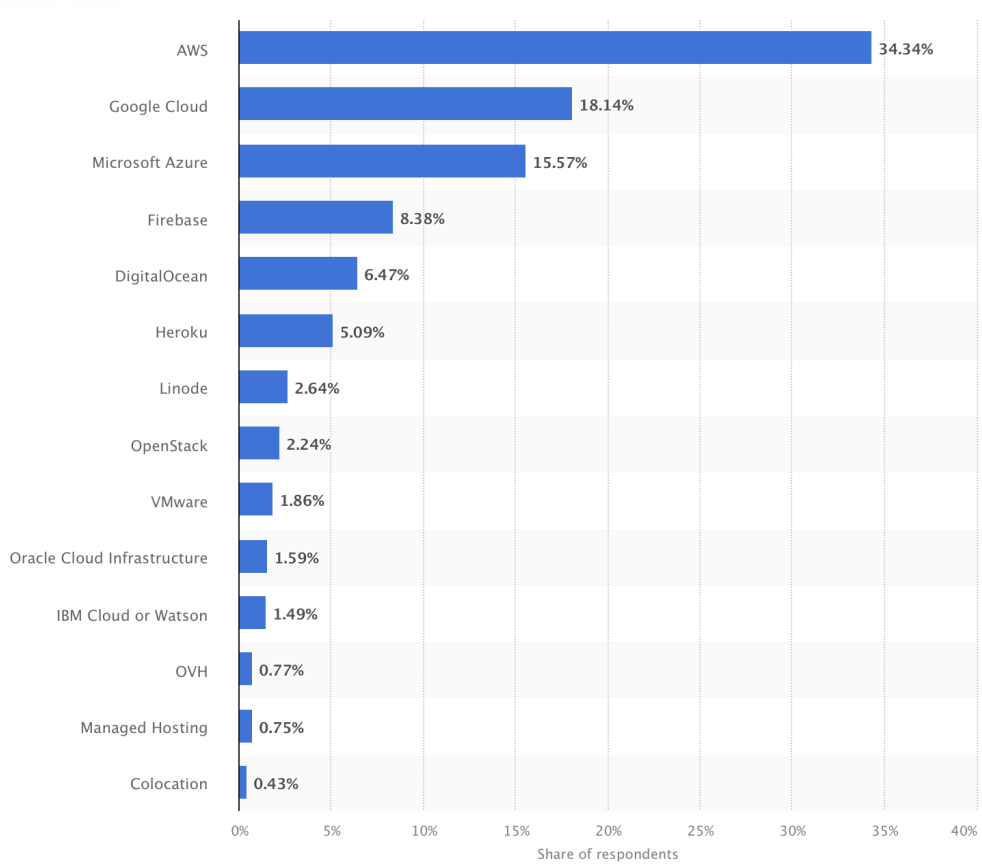


Figure 5.2: Cloud platform più utilizzate dagli sviluppatori nel 2022 secondo Statista [23].

5.4.1 Gestione degli Utenti e Sicurezza

Firebase Authentication semplifica la gestione dell'autenticazione degli utenti, offrendo supporto per diversi metodi di accesso, tra cui email e password, OAuth e provider esterni. La funzionalità di Firebase App Check aiuta a proteggere le risorse backend da abusi come frodi e phishing, garantendo la sicurezza delle transazioni e dei dati degli utenti. Ad esempio, è possibile implementare un sistema di login sicuro che utilizza autenticazione a due fattori per migliorare la protezione dell'account utente.

5.4.2 Analytics e Personalizzazione

Google Analytics per Firebase offre strumenti avanzati per monitorare e analizzare l'utilizzo dell'applicazione, fornendo insight dettagliati sul comportamento degli utenti. Questo consente di ottimizzare le strategie di marketing e personalizzare l'esperienza utente. Con la funzionalità di A/B Testing, è possibile testare diverse versioni di funzionalità o interfacce per determinare quale versione ottiene i migliori risultati in termini di conversioni e soddisfazione degli utenti.

5.4.3 Notifiche e Comunicazione

Firebase Cloud Messaging (FCM) consente di inviare notifiche push agli utenti per mantenerli informati su promozioni, offerte e aggiornamenti importanti. Le notifiche push possono essere personalizzate e indirizzate a segmenti specifici di utenti, migliorando l'engagement e stimolando le vendite. Ad esempio, è possibile inviare notifiche personalizzate ai clienti che hanno abbandonato il carrello per incoraggiarli a completare l'acquisto.

5.4.4 Scalabilità e Prestazioni

Firebase è progettato per gestire carichi di lavoro elevati, con una scalabilità automatica che si adatta alle esigenze dell'applicazione. Questa capacità di scalabilità è cruciale per le piattaforme e-commerce, che devono gestire un alto volume di traffico e dati senza compromettere le prestazioni. Firebase fornisce anche strumenti per monitorare le prestazioni dell'applicazione e ottimizzare l'esperienza utente.

5.5 Firebase Realtime Database e la sua integrazione con Flutter

Il Firebase Realtime Database è un database NoSQL ospitato nel cloud che permette la sincronizzazione dei dati in tempo reale tra tutti i client connessi. I dati sono memorizzati come JSON e possono essere letti e scritti in tempo reale, offrendo un'esperienza utente altamente reattiva. Questo database è ideale per applicazioni che richiedono aggiornamenti immediati dei dati, come chat in tempo reale, giochi multiplayer e applicazioni di collaborazione [19][7].

5.5.1 Integrazione con Flutter

Flutter è un framework UI open-source sviluppato da Google che consente di creare applicazioni native per iOS e Android utilizzando un'unica base di codice. Per integrare Firebase Realtime Database con Flutter, è necessario seguire questi passaggi:

5.5.1.1 Passaggi per l'Integrazione

- **Configurazione del Progetto**: Creare un progetto Firebase nella console Firebase e registrare l'app Flutter. Aggiungere il file di configurazione 'google-services.json' per Android o 'GoogleService-Info.plist' per iOS al progetto Flutter.

- **Installazione del Pacchetto**: Aggiungere il pacchetto 'firebase_database' al file 'pubspec.yaml' del progetto Flutter:

Listing 5.1: Aggiunta pacchetto Firebase alle dipendenze

```
dependencies:
  firebase_core: latest_version
  firebase_database: latest_version
```

- **Accesso e Scrittura dei Dati**: Utilizzare le API fornite dal pacchetto 'firebase_database' per leggere e scrivere dati nel Realtime Database. Creare listener per aggiornamenti in tempo reale e gestire le modifiche ai dati. Esempio di codice:

Listing 5.2: Esempio di utilizzo FirebaseDatabase

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

final databaseReference = FirebaseDatabase.instance.reference();

void addData() {
  databaseReference.child("orders").push().set({
    'item': 'Product Name',
    'quantity': 1,
  });
}

void fetchData() {
  databaseReference.child("orders").once().then((DataSnapshot snapshot) {
    print('Data : ${snapshot.value}');
  });
}
```

- **Gestione degli Errori e Ottimizzazione**: Implementare una gestione degli errori per gestire eventuali problemi di rete o di sincronizzazione. Ottimizzare le performance per garantire che l'applicazione rimanga reattiva e affidabile.

5.5.2 Esempi di Utilizzo

- **Sincronizzazione dei Prodotti**: Utilizzare Firebase Realtime Database per mantenere sincronizzate le informazioni sui prodotti tra tutti i dispositivi degli utenti. Quando un prodotto viene aggiornato, tutte le istanze dell'applicazione visualizzeranno immediatamente le modifiche.

- **Gestione degli Ordini**: Implementare una funzionalità per la gestione degli ordini in tempo reale, consentendo agli utenti di vedere lo stato degli ordini e ricevere aggiornamenti istantanei sullo stato della loro spedizione.

- **Monitoraggio delle Interazioni degli Utenti**: Utilizzare Firebase Realtime Database per monitorare le interazioni degli utenti con l'applicazione e fornire un'esperienza personalizzata basata sui dati in tempo reale.

5.6 Aspetti di sicurezza, autorizzazione e sincronizzazione in tempo reale

Firebase eccelle nella gestione della sicurezza e della sincronizzazione dei dati in tempo reale, due aspetti fondamentali per le app e-commerce, dove la protezione dei dati sensibili degli utenti e l'aggiornamento tempestivo delle informazioni sono cruciali.

5.6.1 Autenticazione e gestione degli utenti

Firebase Authentication fornisce un modo semplice per autenticare gli utenti utilizzando email e password, oltre a supportare provider di autenticazione esterni come Google, Facebook, Twitter e altri. Grazie al supporto delle regole di sicurezza Firebase, è possibile controllare dettagliatamente chi può accedere ai dati e quali operazioni possono essere effettuate, garantendo che solo gli utenti autorizzati abbiano accesso alle risorse critiche.

5.6.2 Regole di sicurezza e Firestore

Firestore, il database NoSQL di Firebase, permette di definire regole di sicurezza dettagliate per controllare quali utenti possono accedere ai dati e in che modo. Questo sistema di regole basato su espressioni condizionali consente di creare una sicurezza granulare a livello di singoli documenti o collezioni, proteggendo al tempo stesso l'integrità e la riservatezza dei dati. Le regole possono essere modificate in modo dinamico, adattandosi rapidamente a nuove esigenze di business senza dover riprogettare l'infrastruttura di backend.

5.6.3 Sincronizzazione in tempo reale con Real-time Database e Firestore

Firebase offre due opzioni per la gestione dei dati in tempo reale:

- **Firebase Realtime Database**: un database NoSQL che consente la sincronizzazione dei dati in tempo reale tra client e server. Quando i dati

cambiano, tutti i client collegati vengono automaticamente aggiornati.

- **Cloud Firestore**: un database flessibile, scalabile e di nuova generazione, che combina le migliori caratteristiche del Realtime Database con funzionalità avanzate di query e indicizzazione.

Questi database sono ideali per applicazioni e-commerce dove la sincronizzazione dei dati in tempo reale, come lo stato degli ordini, l'inventario e la gestione dei clienti, è essenziale. Entrambi i database supportano regole di sicurezza personalizzabili per garantire che solo gli utenti autorizzati possano accedere o modificare i dati.

5.6.4 Cloud Functions per la sicurezza e automazione

Le *Cloud Functions* sono un servizio serverless che permette di eseguire codice backend in risposta a eventi. Ad esempio, è possibile utilizzare le Cloud Functions per eseguire controlli di sicurezza quando un utente modifica i dati del proprio account o effettua un acquisto. Questo aiuta a prevenire attività malevole e a mantenere l'integrità dell'applicazione.

5.7 Monitoraggio e analisi con strumenti Firebase

Per garantire il successo di un'app e-commerce, è fondamentale monitorare continuamente il suo stato e le sue prestazioni. Firebase fornisce una vasta gamma di strumenti per aiutare gli sviluppatori a monitorare e ottimizzare le prestazioni e la qualità delle app.

5.7.1 Crashlytics per il monitoraggio degli errori

Firebase Crashlytics offre un sistema di monitoraggio in tempo reale che traccia e analizza i crash dell'app. Questo strumento aiuta a identificare e risolvere rapidamente i problemi, mostrando informazioni dettagliate su come e quando si

verificano i crash. Con l'integrazione dell'intelligenza artificiale di Gemini, Crashlytics può anche fornire approfondimenti assistiti dall'AI, aiutando gli sviluppatori a risolvere i problemi più velocemente.

5.7.2 Performance Monitoring per ottimizzare le prestazioni

Firebase Performance Monitoring fornisce un quadro completo delle prestazioni della tua app, identificando eventuali colli di bottiglia. Permette di analizzare la latenza di rete, il tempo di rendering delle pagine e le chiamate alle API per ottimizzare il flusso dell'app. In un'app e-commerce, questo può essere cruciale per garantire che i clienti non incontrino difficoltà durante la navigazione o il completamento di un acquisto.

5.7.3 Google Analytics per Firebase

Uno degli strumenti più potenti di Firebase è Google Analytics, che permette di tracciare il comportamento degli utenti all'interno dell'app e fornire metriche utili per migliorare l'esperienza dell'utente. Google Analytics può aiutare a monitorare le conversioni, i percorsi di acquisto e altri eventi chiave, offrendo insight su come ottimizzare l'interfaccia e aumentare le vendite.

5.7.4 Remote Config e A/B Testing per ottimizzare l'esperienza utente

Firebase *Remote Config* permette di modificare il comportamento e l'aspetto dell'applicazione senza dover rilasciare nuovi aggiornamenti. Insieme ad A/B Testing, Remote Config consente di testare differenti versioni dell'app su segmenti di utenti e monitorare quale versione produce i migliori risultati in termini di coinvolgimento o vendite.

5.7.5 Integrazione con AI per il monitoraggio avanzato

Grazie all'integrazione con le API AI come *Gemini* e *Vertex AI*, Firebase offre la possibilità di implementare funzionalità avanzate per il monitoraggio intelligente. Queste tecnologie possono essere utilizzate per migliorare l'esperienza utente, fornendo suggerimenti intelligenti e ottimizzando automaticamente alcuni aspetti dell'app, come le raccomandazioni sui prodotti.

5.8 Conclusioni

Firebase si dimostra una piattaforma versatile e potente per la costruzione, la gestione e il monitoraggio di applicazioni e-commerce. Grazie alla sua integrazione con Google Cloud e alle nuove funzionalità AI, rappresenta una scelta eccellente per gli sviluppatori che vogliono creare app moderne, scalabili e pronte per il futuro.[18]

Chapter 6

Caso di Studio: Flutter E-commerce

In questo capitolo verrà esaminato l'utilizzo di Flutter per la realizzazione di un'applicazione di e-commerce. A tal fine, sono state sviluppate due applicazioni base: un e-commerce multi-vendor e un e-commerce single-vendor. Queste applicazioni consentono di esplorare le potenzialità della piattaforma Flutter e di confrontare le loro similitudini e differenze.

6.1 Obiettivi e Requisiti

6.1.1 Obiettivi

L'obiettivo di un'applicazione e-commerce, come noto, è quello di consentire agli utenti di visualizzare prodotti, selezionarli, aggiungerli al carrello e procedere all'acquisto.

Nel primo caso di studio, l'applicazione rappresenta un e-commerce di prodotti per animali, dove piccole aziende possono registrarsi e vendere i loro prodotti ai proprietari di animali. Nel secondo caso, è stata sviluppata un'applicazione per la rivendita di libri cinesi, con un focus maggiore sull'accessibilità e il supporto multilinguistico. In quest'ultimo esempio, è stata progettata un'interfaccia utente moderna, alla quale si sono aggiunte funzionalità social, come la possibilità di postare e chattare tra utenti, offrendo un'esperienza di acquisto più coinvolgente.

6.1.2 Requisiti

Uno dei requisiti principali per entrambe le applicazioni è la possibilità di leggere, creare e modificare i dati da un database, mediante chiamate a Firebase, e di visualizzarli in modo chiaro e accessibile per gli utenti finali.

La scelta di utilizzare Flutter per lo sviluppo di queste applicazioni deriva dalla sua capacità di supportare più piattaforme, riducendo così i tempi di sviluppo (time-to-market) per un'azienda che desideri entrare nel mercato con un'app di e-commerce, figura 6.1.

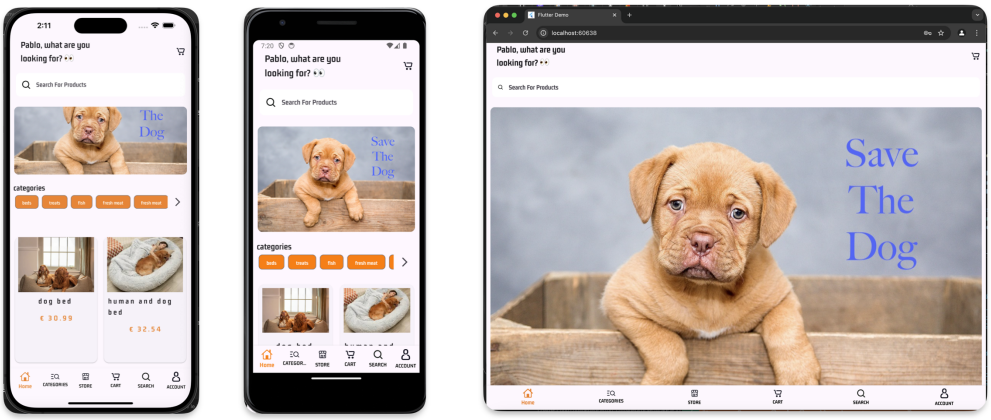


Figure 6.1: Multivendor HomePage testata su iOS Android e Google Chrome

Le due applicazioni si distinguono in particolare per le rispettive schermate principali:

Multi-vendor:

- Lato Acquirente:
  - Home
  - Categories
  - store
  - Cart
  - Search
  - Account

- Lato Venditore:
  - Earnings
  - Upload
  - Edit
  - Orders
  - Logout

Single-vendor:

- Home
- Favorites
- Comunity
- Cart
- Account

Entrambe le applicazioni includono una piattaforma web gestionale. Per il multi-vendor, questa piattaforma consente il controllo dei venditori, dei prodotti e altre funzioni amministrative, pensate per i tecnici e gli operatori del sistema. Nel caso dell'e-commerce single-vendor, la piattaforma gestionale è simile ma pensata per un unico venditore, e consente l'inserzione di prodotti, la gestione di banner e categorie, e la gestione delle spedizioni, figura 6.2.

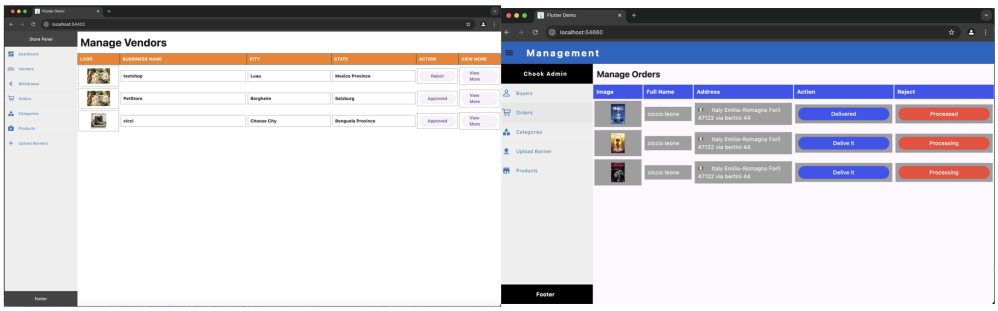


Figure 6.2: Gestionali web per le due piattaforme

6.2 E-commerce Multi-vendor

Questa sezione descrive le principali schermate dell'e-commerce multi-vendor, sia dal punto di vista dell'utente finale (Buyer) che del venditore (Vendor).

6.2.1 Lato Buyer

L'interfaccia utente per l'acquirente consente di gestire il proprio account e di effettuare acquisti in modo semplice e intuitivo.

6.2.1.1 Home

La pagina principale permette agli utenti di cercare prodotti tramite una barra di ricerca, visualizzare banner pubblicitari e scorrere una lista di prodotti con nome e prezzo. È possibile filtrare i prodotti per categoria cliccando sulla rispettiva categoria. Inoltre, è visibile il numero di articoli nel carrello.

6.2.1.2 Categorie

In questa pagina si visualizzano le categorie di prodotti con immagini e nomi. Cliccando su una categoria, l'utente viene reindirizzato alla lista dei prodotti corrispondenti.

6.2.1.3 Store

Questa schermata mostra una lista dei negozi disponibili con nome, immagine e nazionalità. Selezionando un negozio, si accede ai prodotti in vendita di quel negozio specifico.

6.2.1.4 Carrello

L'utente può visualizzare i prodotti inseriti nel carrello, modificarne le quantità o eliminarli.

6.2.1.5 Ricerca

Una pagina dedicata esclusivamente alla ricerca di prodotti in base alle richieste dell'utente.

6.2.1.6 Account

Questa pagina consente all'utente di visualizzare e modificare il proprio profilo, gestire gli ordini e monitorare il loro stato.

6.2.2 Lato Vendor

L'interfaccia del venditore consente di gestire le attività commerciali e monitorare i guadagni.

6.2.2.1 Guadagni

La homepage del lato venditore mostra il totale dei guadagni e il numero di ordini ricevuti.

6.2.2.2 Caricamento Prodotti

Questa schermata permette di aggiungere nuovi prodotti compilando le informazioni necessarie.

6.2.2.3 Modifica Prodotti

Permette al venditore di gestire, modificare o disattivare i prodotti pubblicati.

6.2.2.4 Ordini

In questa pagina è possibile visualizzare i dettagli degli ordini e monitorare lo stato delle spedizioni.

6.2.2.5 Logout

Una schermata dedicata alla gestione del profilo del venditore e alla disconnessione.

6.3 E-commerce Single-vendor

Questa sezione descrive le principali schermate dell'applicazione single-vendor, rivolta alla vendita di libri cinesi.

6.3.1 Home

La homepage dell'applicazione consente di visualizzare i prodotti in vendita, aggiungerli ai preferiti cliccando su un'icona a forma di cuore e vedere dettagli come il numero di prodotti venduti, la valutazione tramite stelle e il numero di commenti. È possibile accedere alle categorie di libri tramite icone illustrative.

Inoltre, l'utente può visualizzare notifiche in tempo reale tramite un'icona di messaggio (per i messaggi non letti) e una campanella (per aggiornamenti di stato).

6.3.2 Preferiti

In questa pagina è possibile gestire i prodotti aggiunti ai preferiti e rimuoverli se necessario.

6.3.3 Comunità

Questa pagina rappresenta una funzionalità social integrata nell'applicazione, pensata per incentivare l'interazione tra gli utenti. Come dimostrato in precedenza nella tesi, i social network occupano gran parte del tempo trascorso dagli utenti sulle applicazioni.

La sezione comunitaria consente agli utenti di visualizzare i post, cercarli e avviare conversazioni con altri utenti, rendendo l'esperienza di acquisto più coinvolgente e partecipativa.

6.3.4 Carrello

Permette all'utente di visualizzare gli articoli nel carrello, modificarne le quantità o eliminarli.

6.3.5 Account

Questa pagina consente all'utente di gestire il proprio profilo, impostare la lingua preferita tramite le impostazioni, visualizzare lo stato degli ordini e modificare i propri dati personali.

6.4 Design per l'e-commerce

6.4.1 Scelte architetturali

In questa sezione vengono presentate le scelte architetturali adottate per lo sviluppo delle due applicazioni: multi-vendor e single-vendor. Per ogni applicazione, vengono discussi sia l'architettura delle pagine e i loro collegamenti, sia i casi d'uso principali. Inoltre, vengono giustificate le scelte tecnologiche in termini di usabilità, scalabilità e gestione delle funzionalità.

6.4.1.1 Architettura dell'Applicazione Multi-vendor

Per la realizzazione dell'applicazione è stato adottato il modello Model-View-Controller (MVC) insieme al pattern Provider per la gestione dello stato. Questa scelta consente una chiara separazione delle responsabilità e una gestione efficiente della comunicazione tra i vari componenti dell'applicazione.

6.4.1.2 Modello Model-View-Controller (MVC) + Provider

Il modello Model-View-Controller (MVC) è stato scelto per strutturare l'applicazione in modo da facilitare la manutenzione e la scalabilità. Questo modello divide l'applicazione in tre componenti principali:

- **Model**: Rappresenta i dati dell'applicazione e la logica di business. In questo caso, il modello gestisce i dati relativi ai prodotti, al carrello e alle informazioni degli utenti. Gli oggetti modello comunicano con il database e aggiornano lo stato dell'applicazione.
- **View**: Gestisce la presentazione dei dati all'utente e la visualizzazione dell'interfaccia utente. I widget Flutter che mostrano le schermate e gli

elementi dell'applicazione fanno parte della View. Ogni schermata e componente visivo è rappresentato da un widget che può essere riutilizzato in diverse parti dell'applicazione.

- **Controller**: Gestisce le interazioni dell'utente e coordina la comunicazione tra il Model e la View. I controller gestiscono le operazioni come l'autenticazione degli utenti, l'aggiornamento dei dati e la navigazione tra le schermate. Utilizzano i dati provenienti dal Model per aggiornare la View e viceversa.

- **Provider**: Oltre al modello MVC, è stato utilizzato il pattern Provider per la gestione dello stato dell'applicazione. Provider consente di gestire e condividere i dati tra i diversi widget in modo efficiente, garantendo che le modifiche ai dati siano riflettute in tempo reale nella View.

6.4.1.3 Lato Buyer

Controller: Il controller principale per la gestione dell'autenticazione e delle operazioni utente è AuthController. Questo controller interagisce con Firebase per la gestione degli utenti e delle immagini di profilo, gestendo operazioni come la registrazione, il login e l'upload delle immagini.

Model: I modelli CartAttr e ProductProvider rappresentano i dati relativi al carrello e ai prodotti. CartAttr gestisce le informazioni specifiche di un prodotto nel carrello, mentre ProductProvider si occupa dei dettagli dei prodotti, come nome, prezzo e immagini.

Provider: CartProvider è responsabile della gestione dello stato del carrello. Gestisce l'aggiunta, la rimozione e l'aggiornamento degli articoli nel carrello e calcola il totale.

View: I widget sono responsabili della visualizzazione delle schermate e degli elementi dell'interfaccia utente, come la home page, il carrello e le pagine di dettaglio dei prodotti.

6.4.1.4 Lato Vendor

Controller Il VendorController gestisce le operazioni legate ai venditori, come la registrazione del venditore e l'upload delle immagini del negozio. Utilizza Firebase per gestire i dati dei venditori e per l'archiviazione delle immagini.

Model Il VendorUserModel definisce le strutture dati relative ai venditori, inclusi i dettagli del negozio e le informazioni di contatto. Questo modello è utilizzato per gestire e memorizzare i dati dei venditori.

View: La View comprende tutti i widget che costituiscono l'interfaccia utente , dalla home page, all'upload dei prodotti all'editing fino alla gestione dell'account.

La view si occupa della visualizzazione delle schermate principali e dei dettagli delle pagine.

6.4.1.5 Architettura della View

L'architettura dell'applicazione multi-vendor è stata progettata per gestire sia i buyer che i vendor, mantenendo separati i percorsi di navigazione per questi due ruoli. Le pagine sono strutturate in modo che ciascun utente possa accedere alle funzionalità rilevanti attraverso un'interfaccia semplice e ben definita. In figura 6.3, viene mostrata l'architettura degli Widget dedicati alle pagine dell'applicazionie, l'app utilizza il sistema di Navigator per consentire la navigazione tra le diverse schermate, costruendo una sequenza di percorsi (Route) che permette agli utenti di spostarsi tra i vari widget e tornare indietro lungo il percorso navigato. Questo meccanismo facilita la gestione delle transizioni tra le schermate dell'applicazione.

Al primo avvio, l'app presenta una schermata di login, dove l'utente può inserire le proprie credenziali per accedere come buyer. In alternativa, è possibile registrare un nuovo account o accedere alla pagina di login per i vendor, per la gestione dell'account venditore.

6.4.1.6 Casi d'uso

La figura 6.4 illustra i casi d'uso dell'applicazione per gli utenti Buyer e Vendor. Entrambi i ruoli possono autenticarsi e visualizzare lo stato degli ordini. L'autenticazione estende una verifica delle credenziali. Il cliente può cercare, visualizzare e aggiungere prodotti al carrello. Può inoltre modificare il carello e

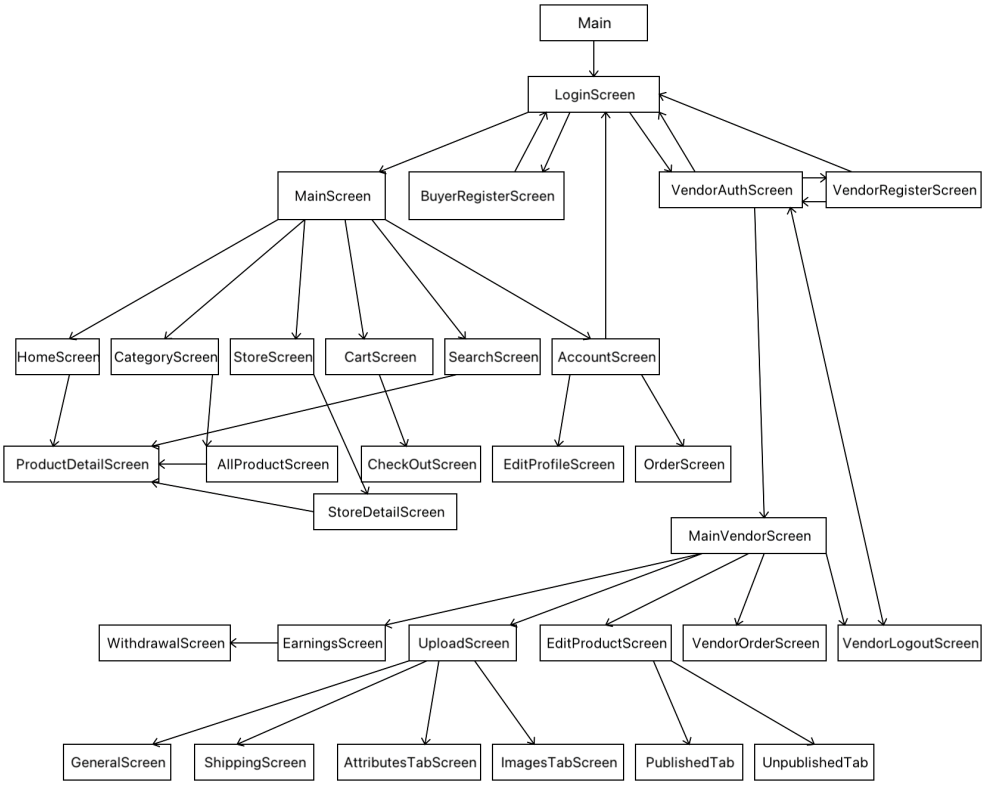


Figure 6.3: Architettura degli screen Multivendor

procedere al checkout, la procedura di checkout include la selezione del metodo di pagamento e la conferma d'ordine.

Il venditore invece può aggiungere prodotti che include inserirne i dettagli, disattivare o riattivare quelli già inseriti oppure modificarli.

6.4.1.7 Architettura dell'Applicazione Single-vendor

L'applicazione Single-vendor è organizzata in modo simile all'app Multi-vendor, con alcune differenze chiave che si riflettono nell'architettura e nei casi d'uso specifici. Entrambe le applicazioni utilizzano il modello Model-View-Controller (MVC) e il pattern Provider per la gestione dello stato, ma l'app Single-vendor è ottimizzata per un solo venditore offrendogli più servizi, semplificando alcuni aspetti dell'architettura.

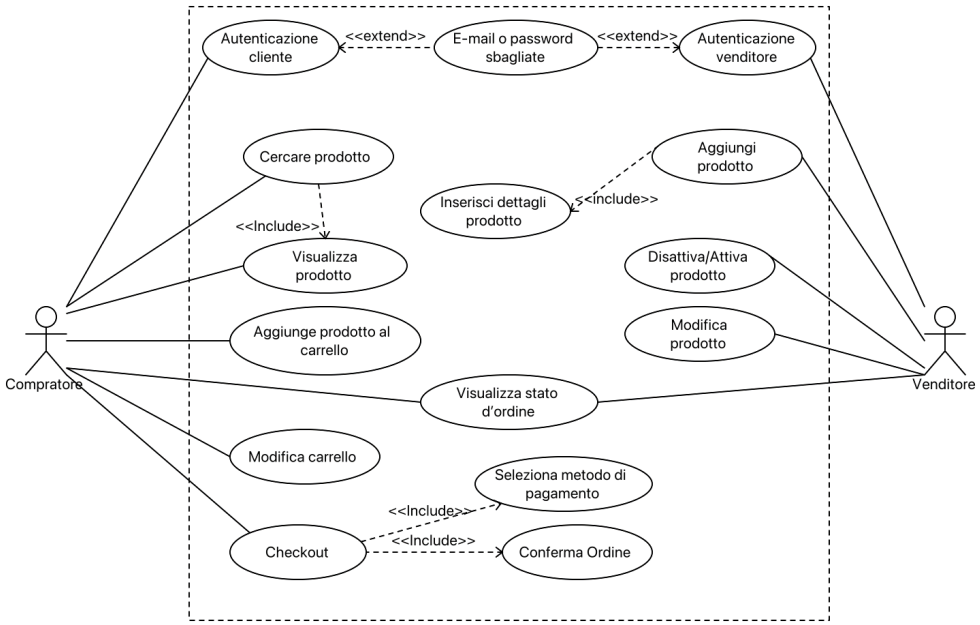


Figure 6.4: Casi d'uso Multivendor

6.4.1.8 Architettura del Model-View-Controller (MVC) + Provider

Model:

Il Model è responsabile della gestione dei dati e della logica di business. Per l'app Single-vendor, i modelli principali sono:

CartModel: Rappresenta un articolo nel carrello dell'utente. Include dettagli come nome del prodotto, prezzo, quantità e descrizione.

CategoryModel: Gestisce le categorie di prodotti disponibili. Ogni categoria ha un nome e un'immagine associata.

FavoriteModel: Rappresenta un prodotto salvato tra i preferiti dell'utente. Include il nome del prodotto, il suo ID e altre informazioni rilevanti.

ReviewModel: Gestisce le recensioni dei prodotti, con dettagli su chi ha scritto la recensione, il punteggio e il testo della recensione.

Controller:

I controller gestiscono le interazioni tra l'utente e il sistema. Per l'app Single-vendor, i principali controller sono:

AuthController: Gestisce l'autenticazione degli utenti. Permette la regis-

trazione e il login dei clienti, interagendo con Firebase per la gestione degli utenti e la verifica delle credenziali.

BannerController: Gestisce la visualizzazione dei banner pubblicitari. Recupera le URL delle immagini dei banner da Firebase e fornisce un flusso di dati per aggiornare la View.

CategoryController: Gestisce le categorie di prodotti. Recupera e aggiorna le informazioni sulle categorie da Firebase, mantenendo un elenco reattivo di categorie che viene aggiornato automaticamente nella View.

HeaderController: Gestisce la visualizzazione dei messaggi non letti. Recupera le informazioni sui messaggi non letti per l'utente attualmente autenticato e aggiorna la View di conseguenza.

Provider:

Il Provider è utilizzato per gestire e condividere lo stato tra i widget. Le principali classi Provider sono:

cartProvider: Utilizza la classe **CartNotifier** per gestire lo stato del carrello. Consente di aggiungere, rimuovere, incrementare e decrementare gli articoli nel carrello e calcolare il totale.

favoriteProvider: Utilizza la classe **FavoritesNotifier** per gestire lo stato dei prodotti preferiti. Permette di aggiungere e rimuovere prodotti dai preferiti e fornisce l'elenco aggiornato dei prodotti salvati.

Questa organizzazione consente una gestione centralizzata dei dati e un aggiornamento in tempo reale dell'interfaccia utente, assicurando che le modifiche ai dati siano immediatamente riflesse nella View. L'approccio MVC + Provider garantisce che le funzionalità dell'app Single-vendor siano efficienti e facilmente scalabili, pur mantenendo una chiara separazione delle responsabilità tra i vari componenti dell'applicazione.

6.4.1.9 Architettura della View

L'architettura dell'applicazione single-vendor è stata progettata per ottimizzare l'usabilità per i clienti. In figura 6.5, viene mostrata l'architettura degli Widget dedicati alle pagine dell'applicazionie, Anche qui l'utilizzo di Navigator permette la navigazione tra le diverse schermate.

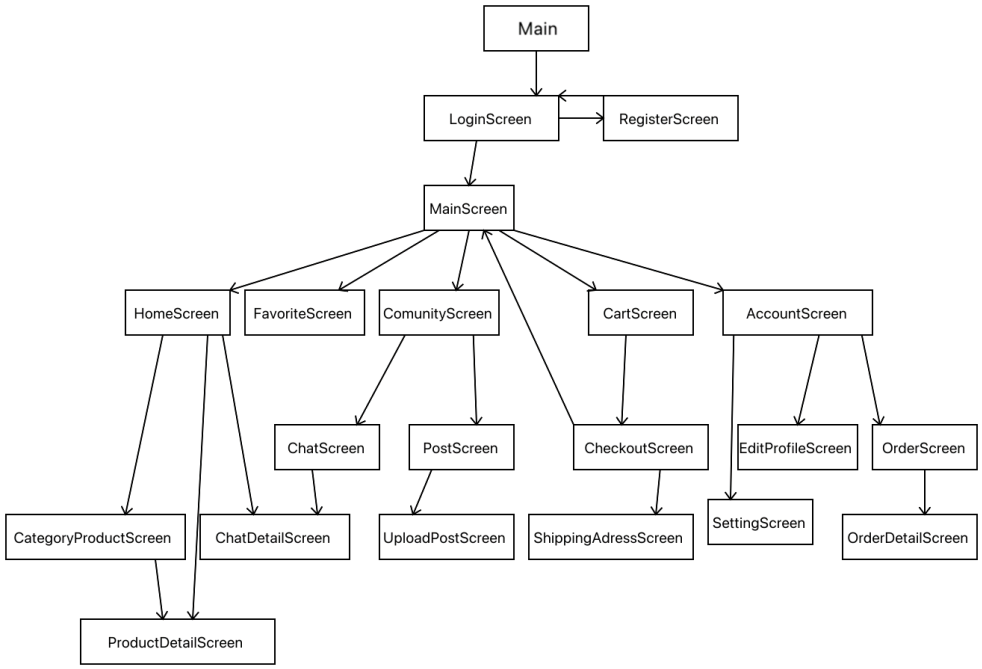


Figure 6.5: Architettura degli screen Single-Vendor

Al primo avvio, l'app presenta una schermata di login, dove l'utente può autenticarsi per essere poi indirizzato alla homescreen, la principale differenza tra le due applicazioni è la possibilità di accedere all'interfaccia social dell'app per potersi mettere in contatto con altri utenti dell'app, tramite post o chat.

6.4.1.10 Casi d'uso

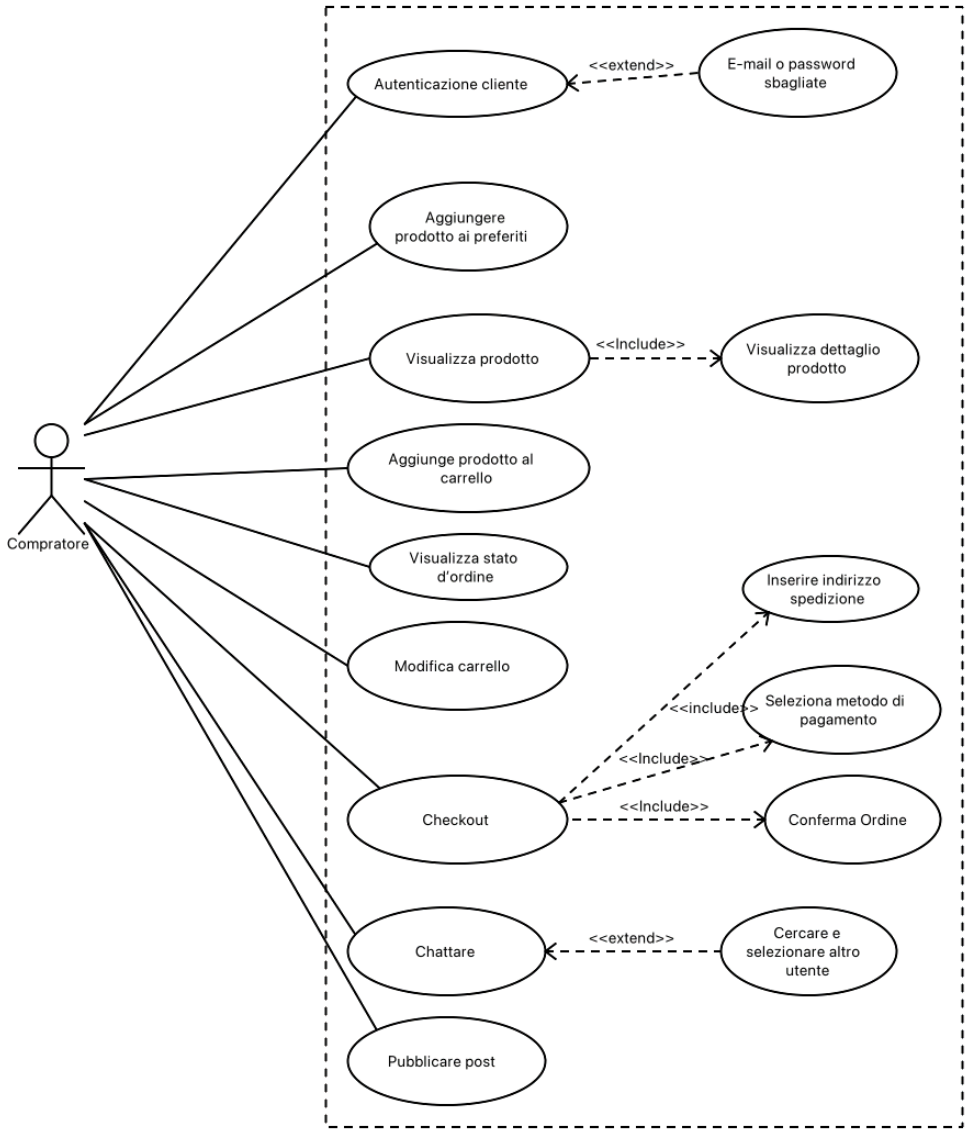


Figure 6.6: Casi d'uso Single-vendor

La figura 6.6 illustra i casi d'uso del cliente. Che si differenzia dal cliente del multi-vendor per questi aspetti: -Aggiungere prodotti ai preferiti; -L'indirizzo di spedizione può essere inserito al checkout; -Si possono recensire gli ordini ricevuti; -Effettuata la ricerca di un buyer è possibile chattare con lui -E' possibile pubblicare

post.

6.4.2 Struttura del database e gestione dei dati

L'applicazione utilizza Firebase Cloud Firestore come sistema di gestione del database. Firestore è un database NoSQL che consente di memorizzare dati in forma di documenti organizzati in collezioni. Ogni documento è schema-less, ovvero non ha una struttura fissa e può contenere campi di vario tipo, come stringhe, numeri, array, oggetti nidificati e timestamp.

Di seguito viene descritta la struttura dei dati per le entità principali della piattaforma **Multi-vendor**: *buyers*, *vendors*, *orders*, *categories*, *withdrawals* e *banners*. È importante notare che gli ID degli utenti (buyers e vendors) coincidono con lo UID dell'authentication Firebase, garantendo l'univocità dei record e una facile associazione con l'autenticazione.

- **Buyers** (Collezione: buyers):
  - **buyerId** (stringa): UID assegnato automaticamente da Firebase Authentication.
  - **fullName** (stringa): Nome completo del buyer.
  - **email** (stringa): Indirizzo email del buyer.
  - **phoneNumber** (stringa): Numero di telefono del buyer.
  - **address** (stringa): Indirizzo di residenza del buyer.
  - **profileImage** (stringa): URL dell'immagine del profilo salvata su Firebase Storage.
- **Vendors** (Collezione: 'vendors'):
  - **vendorId** (stringa): UID assegnato automaticamente da Firebase Authentication.
  - **businessName** (stringa): Nome del negozio.
  - **email** (stringa): Indirizzo email del venditore.
  - **phoneNumber** (stringa): Numero di telefono del venditore.

  - **countryValue** (stringa): Paese del negozio.
  - **stateValue** (stringa): Stato o provincia di residenza.
  - **cityValue** (stringa): Città del negozio.
  - **storeImage** (stringa): URL dell'immagine del negozio salvata su Firebase Storage.
  - **taxNumber** (stringa): Numero di partita IVA o codice fiscale.
  - **taxRegistered** (stringa): Indica se il venditore è registrato ai fini fiscali (es. "YES").
  - **approved** (booleano): Indica se il venditore è stato approvato.

- **Orders** (Collezione: 'orders'): Ogni ordine è identificato da un `orderId` univoco e contiene le seguenti informazioni:
  - **orderId** (stringa): ID dell'ordine.
  - **buyerId** (stringa): UID del buyer che ha effettuato l'ordine.
  - **vendorId** (stringa): UID del venditore che riceve l'ordine.
  - **productId** (stringa): ID del prodotto ordinato.
  - **productName** (stringa): Nome del prodotto.
  - **productPrice** (numero): Prezzo del prodotto.
  - **productQuantity** (numero): Quantità ordinata del prodotto.
  - **productSize** (stringa): Dimensione del prodotto, se applicabile.
  - **productImage** (array di stringhe): URL delle immagini del prodotto salvate su Firebase Storage.
  - **buyerAddress** (stringa): Indirizzo di spedizione del buyer.
  - **buyerFullName** (stringa): Nome completo del buyer.
  - **buyerPhone** (stringa): Numero di telefono del buyer.
  - **accepted** (booleano): Stato di accettazione dell'ordine da parte del venditore.
  - **orderDate** (timestamp): Data e ora in cui l'ordine è stato effettuato.

  - **scheduleDate** (timestamp): Data prevista per la consegna dell'ordine.

- **Categories** (Collezione: 'categories'):
  - **categoryName** (stringa): Nome della categoria di prodotti (es. "beds").
  - **image** (stringa): URL dell'immagine rappresentativa della categoria salvata su Firebase Storage.

- **Withdrawals** (Collezione: 'withdrawals'):
  - **withdrawalId** (stringa): ID univoco della richiesta di prelievo.
  - **amount** (stringa): Somma richiesta per il prelievo.
  - **bankAccountName** (stringa): Nome del titolare del conto bancario.
  - **bankAccountNumber** (stringa): Numero del conto bancario.
  - **bankName** (stringa): Nome della banca.
  - **mobile** (stringa): Numero di telefono del venditore associato al conto bancario.
  - **name** (stringa): Nome del venditore che ha richiesto il prelievo.

- **Banners** (Collezione: 'banners'):
  - **bannerId** (stringa): ID univoco del banner pubblicitario.
  - **image** (stringa): URL dell'immagine del banner salvata su Firebase Storage.

Di seguito, vengono descritti i principali documenti e campi utilizzati nel database **Single-vendor**.

Buyers (Acquirenti): Gli acquirenti sono identificati tramite il loro UID assegnato dall'autenticazione Firebase. Ogni documento all'interno della raccolta rappresenta un acquirente e contiene le seguenti informazioni:

- `uid`: L'UID dell'acquirente (*stringa*).
- `fullName`: Nome completo dell'acquirente (*stringa*).

- `email`: Email dell'acquirente (*stringa*).
- `phoneNumber`: Numero di telefono dell'acquirente (*stringa*).
- `profileImage`: URL dell'immagine profilo dell'acquirente (*stringa*).
- `city`, `state`, `country`, `road`, `postNumber`, `postCode`: Indirizzo dell'acquirente (*stringa*).

Categories (Categorie): Ogni categoria di prodotto è identificata da un ID e include informazioni come:

- `categoryName`: Nome della categoria (*stringa*).
- `categoryImage`: URL dell'immagine della categoria (*stringa*).

Chats: Le chat tra gli acquirenti sono archiviate in una raccolta di chat, dove ciascun documento contiene:

- `chatId`: ID della chat, generato come concatenazione degli UID dei partecipanti (*stringa*).
- `lastMessage`: Ultimo messaggio inviato (*stringa*).
- `lastMessageTimestamp`: Timestamp dell'ultimo messaggio (*timestamp*).
- `participants`: Elenco degli UID dei partecipanti (*array di stringhe*).
- **messages** (sotto-raccolta di `chats`):
  - `messageId`: Identificativo del messaggio (stringa).
  - `text`: Testo del messaggio (stringa).
  - `senderId`: UID del mittente del messaggio (stringa).
  - `timestamp`: Timestamp di invio del messaggio (timestamp).
  - `viewedBy`: Array degli UID dei partecipanti che hanno visualizzato il messaggio (array di stringhe).
  - `viewedTimestamp`: Mappa che contiene per ogni partecipante il timestamp in cui ha visualizzato il messaggio (mappa di UID a timestamp).

Orders (Ordini): Gli ordini effettuati dagli acquirenti sono archiviati in una raccolta dedicata. Ogni ordine contiene:

- `orderId`: ID dell'ordine (*stringa*).
- `buyerId`: UID dell'acquirente (*stringa*).
- `fullName`, `email`, `phoneNumber`: Informazioni di contatto dell'acquirente (*stringhe*).
- `productId`: ID del prodotto ordinato (*stringa*).
- `productName`: Nome del prodotto (*stringa*).
- `productImage`: URL dell'immagine del prodotto (*stringa*).
- `quantity`: Quantità ordinata (*numero*).
- `price`: Prezzo del prodotto (*numero*).
- `delivered`: Stato della consegna (*booleano*).

Posts (Post): I post degli utenti (ad esempio, recensioni di prodotti) vengono archiviati con i seguenti campi:

- `postId`: ID del post (*stringa*).
- `buyerId`: UID dell'utente che ha pubblicato il post (*stringa*).
- `description`: Descrizione del post (*stringa*).
- `postImage`: URL dell'immagine allegata al post (*stringa*).
- `timeStamp`: Data e ora del post (*timestamp*).

ProductReviews (Recensioni dei prodotti): Le recensioni lasciate dagli acquirenti sui prodotti sono archiviate nella raccolta delle recensioni. Ogni documento contiene:

- `reviewId`: ID della recensione (*stringa*).

- `buyerId`: UID dell'acquirente che ha scritto la recensione (*stringa*).
- `productId`: ID del prodotto recensito (*stringa*).
- `rating`: Valutazione data al prodotto (*numero*).
- `review`: Testo della recensione (*stringa*).
- `timeStamp`: Data e ora della recensione (*timestamp*).

Products (Prodotti): I prodotti venduti nel sistema single vendor sono memorizzati con informazioni dettagliate come:

- `productId`: ID del prodotto (*stringa*).
- `productName`: Nome del prodotto (*stringa*).
- `category`: Categoria del prodotto (*stringa*).
- `description`: Descrizione del prodotto (*stringa*).
- `productImage`: Elenco di URL delle immagini del prodotto (*array di stringhe*).
- `productPrice`: Prezzo del prodotto (*numero*).
- `quantity`: Quantità disponibile (*numero*).
- `rating`: Valutazione media del prodotto (*numero*).
- `totalSold`: Quantità totale venduta (*numero*).

La gestione dei dati avviene tramite interazioni asincrone con Firebase, sfruttando le funzionalità di `cloud functions` e `Firebase Authentication` per gestire le operazioni di CRUD (Create, Read, Update, Delete). Inoltre, Firebase Storage è utilizzato per memorizzare le immagini dei profili, dei prodotti, dei negozi e dei banner.

6.4.3 Scelte progettuali

6.4.3.1 Widget dell'applicazione

L'applicazione è stata suddivisa in una serie di widget modulari per garantire una migliore organizzazione del codice e la riusabilità dei componenti. In totale, per la gestione dell'app lato *multi-vendor*, sono stati creati 40 widget, 17 per la parte *vendor* e 21 per la parte *buyer*. A questi si aggiungono i widget per la gestione del *main screen* e dell'integrazione con Firebase.

In questa sezione non li illustreremo tutti ma solo quelli principali.

Multi-vendor Il widget principale del lato *buyer* è il `MainScreen`, che include una *BottomNavigationBar* per la navigazione tra le varie schermate principali dell'app, come la homepage, il carrello e il profilo dell'utente. Di seguito viene mostrato uno snippet di codice per illustrare la struttura della `BottomNavigationBar`:

Listing 6.1: BottomNavigationBar all'interno del MainScreen lato buyer

```
    List<Widget> _pages = [
    HomeScreen(),
    CategoryScreen(),
    StoreScreen(),
    CartScreen(),
    SearchScreen(),
    AccountScreen(),
  ];

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      bottomNavigationBar: BottomNavigationBar(
          type: BottomNavigationBarType.fixed,
          currentIndex: _pageIndex,
          onTap: (value) {
            setState(() {
              _pageIndex = value;
            });
          },
          unselectedItemColor: Colors.black,
          selectedItemColor: Colors.yellow.shade900,
          items: [
            BottomNavigationBarItem(
              icon: Icon(CupertinoIcons.home),
```

```
            label: 'Home',
          ),
          BottomNavigationBarItem(
            icon: SvgPicture.asset(
              'assets/icons/explore.svg',
              width: 20,
            ),
            label: 'CATEGORIES',
          ),
          BottomNavigationBarItem(
            icon: SvgPicture.asset(
              'assets/icons/shop.svg',
              width: 20,
            ),
            label: 'STORE',
          ),
          BottomNavigationBarItem(
            icon: SvgPicture.asset(
              'assets/icons/cart.svg',
              width: 20,
            ),
            label: 'CART',
          ),
          BottomNavigationBarItem(
            icon: SvgPicture.asset(
              'assets/icons/search.svg',
              width: 20,
            ),
            label: 'SEARCH',
          ),
          BottomNavigationBarItem(
            icon: SvgPicture.asset(
              'assets/icons/account.svg',
              width: 20,
            ),
            label: 'ACCOUNT',
          ),
        ]),
        body: _pages[_pageIndex],
    );
  }
```

La struttura è composta da una lista di widget che rappresentano le schermate principali dell'app (`HomeScreen`, `CategoryScreen`, `StoreScreen`, `CartScreen`, `SearchScreen`, `AccountScreen`), che vengono visualizzate in base alla selezione dell'utente nella barra di navigazione.

Uno dei widget principali all'interno del lato *buyer* è la `HomeScreen`, che rap-

presenta la homepage. Essa include vari widget secondari come `WelcomeText`, `SearchInputWidget`, e `BannerWidget`, progettati per suddividere le diverse sezioni della schermata in componenti autonomi. Di seguito un esempio di codice per la gestione della `HomeScreen`:

Listing 6.2: Multi-Vendor HomeScreen lato buyer

```
class HomeScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return SingleChildScrollView(
      child: Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: [
          WelcomeText(),
          SizedBox(height: 5),
          SearchInputWidget(),
          BannerWidget(),
          CategoryText(),
        ],
      ),
    );
  }
}
```

Per la gestione dell'app lato *vendor*, il widget principale è il `MainVendorScreen`, che consente la navigazione attraverso le funzionalità principali dell'applicazione per i venditori, come la visualizzazione dei guadagni, l'upload di nuovi prodotti e la gestione degli ordini. Anche in questo caso, viene utilizzata una *BottomNavigationBar* per permettere la navigazione tra le diverse sezioni, come mostrato nell'esempio seguente:

Listing 6.3: BottomNavigationBar lato vendor

```
List<Widget> _pages = [
  EarningsScreen(),
  UploadScreen(),
  EditProductScreen(),
  VendorOrderScreen(),
  VendorLogoutScreen(),
];

return Scaffold(
  bottomNavigationBar: BottomNavigationBar(
    currentIndex: _pageIndex,
    onTap: (value) {
```

```
      setState(() {
        _pageIndex = value;
      });
    },
    type: BottomNavigationBarType.fixed,
    unselectedItemColor: Colors.black,
    selectedItemColor: Colors.yellow.shade900,
    items: [
      BottomNavigationBarItem(icon: Icon(CupertinoIcons.money_dollar), label: '
          EARNINGS'),
      BottomNavigationBarItem(icon: Icon(Icons.upload), label: 'UPLOAD'),
      BottomNavigationBarItem(icon: Icon(Icons.edit), label: 'EDIT'),
      BottomNavigationBarItem(icon: Icon(CupertinoIcons.shopping_cart), label: '
          ORDERS'),
      BottomNavigationBarItem(icon: Icon(Icons.logout), label: 'LOGOUT'),
    ],
  ),
  body: _pages[_pageIndex],
);
```

Una delle pagine più rilevanti per il lato *vendor* è la `UploadScreen`, che permette di caricare nuovi prodotti e gestire i relativi dettagli come prezzo, descrizione e immagini. Questa schermata è suddivisa in diverse `Tab` gestite da una `TabBar`, ognuna delle quali è dedicata a un aspetto specifico del prodotto, come mostrato di seguito:

Listing 6.4: TabBar dell'UploadScreen lato vendor

```
return DefaultTabController(
  length: 4,
  child: Form(
    key: _formKey,
    child: Scaffold(
      appBar: AppBar(
        bottom: TabBar(tabs: [
          Tab(child: Text('General')),
          Tab(child: Text('Shipping')),
          Tab(child: Text('Attributes')),
          Tab(child: Text('Images')),
        ]),
      ),
      body: TabBarView(children: [
        GeneralScreen(),
        ShippingScreen(),
        AttributesTabScreen(),
        ImagesTabScreen(),
      ]),
    ),
```

```
  ),
);
```

Schermate di login Un'altra differenza progettuale significativa tra il lato *buyer* e *vendor* risiede nelle schermate di login.

Per il lato *buyer*, è stato implementato un form di login personalizzato, progettato manualmente, che include campi per l'inserimento delle credenziali come l'indirizzo email e la password. Il codice per la validazione dei campi e l'autenticazione è stato sviluppato utilizzando il framework di Flutter, permettendo una completa personalizzazione dell'interfaccia utente. L'uso di questa soluzione consente maggiore flessibilità e controllo sugli elementi dell'interfaccia e sulle dinamiche di autenticazione, oltre a permettere l'integrazione di logiche personalizzate, come il feedback immediato sui campi o il controllo di eventuali errori di inserimento (ad esempio, campi vuoti o password errata).

Di seguito un esempio di codice per la gestione del form di login per il lato *buyer*:

Listing 6.5: login lato buyer

```
Form(
  key: _formKey,
  child: Column(
    children: [
      TextFormField(
        decoration: InputDecoration(labelText: 'Email'),
        validator: (value) {
          if (value == null || value.isEmpty) {
            return 'Please enter your email';
          }
          return null;
        },
        onChanged: (value) {
          email = value;
        },
      ),
      TextFormField(
        decoration: InputDecoration(labelText: 'Password'),
        obscureText: true,
        validator: (value) {
          if (value == null || value.isEmpty) {
            return 'Please enter your password';
          }
          return null;
        },
```

```
        onChanged: (value) {
          password = value;
        },
      ),
      // Pulsante per il login
      ElevatedButton(
        onPressed: _loginUsers,
        child: Text('Login'),
      ),
    ],
  ),
)
```

Dall'altro lato, per il *vendor*, si è optato per un approccio diverso, sfruttando la libreria *FirebaseUIAuth*. Questo approccio offre un'interfaccia predefinita per la gestione dell'autenticazione tramite l'*EmailAuthProvider* di Firebase, riducendo notevolmente il carico di lavoro legato alla personalizzazione dell'interfaccia utente e alla gestione della sicurezza. Utilizzando questa libreria, si garantisce una maggiore sicurezza e conformità agli standard di autenticazione, con minimi interventi da parte dello sviluppatore.

Ecco un esempio del codice di implementazione dell'interfaccia di login per il lato *vendor*:

Listing 6.6: login lato vendor

```
SignInScreen(
  providers: [
    EmailAuthProvider(),
  ],
)
```

Queste scelte progettuali riflettono le diverse esigenze degli utenti e le funzionalità richieste da ciascuna parte dell'app: nel caso del *buyer*, si è data priorità alla personalizzazione dell'esperienza utente, mentre per il *vendor* l'accento è stato posto su un'implementazione rapida e sicura, grazie alla soluzione completa offerta da FirebaseUIAuth.

Queste differenze sottolineano le potenzialità di Flutter nel permettere approcci diversi in base alle esigenze specifiche di ciascun utente, offrendo sia la possibilità di creare interfacce completamente personalizzabili che di utilizzare librerie già pronte all'uso per risolvere problematiche comuni, come l'autenticazione.

Single-Vendor Nel contesto dell'applicazione *single-vendor*, è stata prestata grande attenzione alla *user experience* e all'estetica visiva, utilizzando vari elementi grafici e *widget modulari* per mantenere la struttura del codice chiara ed estendibile.

L'interfaccia dell'app single-vendor si basa su un approccio modulare, composta da *31 widget*, inclusi i widget di schermate e interni. La schermata principale è gestita dal *MainScreen* che implementa la navigazione tra diverse sezioni dell'app tramite una *BottomNavigationBar*.

Il widget *MainScreen* è il punto di accesso principale che consente la navigazione tra le sezioni principali dell'app, come la Home, i Preferiti, il Carrello e l'Account. Questo è possibile grazie alla barra di navigazione in fondo alla schermata, che cambia dinamicamente la schermata attiva in base all'interazione dell'utente.

Listing 6.7: MainScreen single-vendor

```
Scaffold(
      bottomNavigationBar: BottomNavigationBar(
        selectedItemColor: Colors.purple,
        unselectedItemColor: Colors.grey,
        currentIndex: _pageIndex,
        onTap: (value) {
          setState(() {
            _pageIndex = value;
          });
        },
        type: BottomNavigationBarType.fixed,
        items: [
          BottomNavigationBarItem(
              icon: Semantics(
                  label: translate("Home"),
                  child: const Icon(CupertinoIcons.home)),
              label: translate("Home")),
          BottomNavigationBarItem(
              icon: Semantics(
                label: translate("Favorites"),
                child: Image.asset(
                  "assets/icons/love.png",
                  width: 25,
                ),
              ),
              label: translate("Favorites")),
          BottomNavigationBarItem(
              icon: Semantics(
                label: translate("Comunity"),
```

```
              child: const Icon(CupertinoIcons.chat_bubble_2),
            ),
            label: translate("Comunity")),
        BottomNavigationBarItem(
            icon: Semantics(
              label: translate("Cart"),
              child: Image.asset(
                "assets/icons/cart.png",
                width: 25,
              ),
            ),
            label: translate("Cart")),
        BottomNavigationBarItem(
            icon: Semantics(
              label: translate("Account"),
              child: Image.asset(
                "assets/icons/user.png",
                width: 25,
              ),
            ),
            label: translate("Account")),
      ],
    ),
    body: _pages[_pageIndex],
  );
```

La schermata Home è uno dei widget più complessi, essendo suddivisa in più widget secondari, come il *BannerWidget*, *CategoryItem*, e *ProductWidget*. Qui, vengono utilizzati widget specifici per mostrare i prodotti più popolari, le categorie e le raccomandazioni personalizzate.

Per migliorare ulteriormente l'esperienza utente, il widget *ProductItemWidget* utilizza elementi grafici avanzati come *Positioned* e *Stack*, arricchendo la presentazione visiva dei prodotti. Viene utilizzato *CachedNetworkImage* per il caricamento ottimizzato delle immagini, e sono presenti effetti visivi come ombre e bordature per dare rilievo agli elementi chiave.

Nel widget dei prodotti, sono presenti elementi grafici posizionati con precisione, che includono `effetti di ombre`, `transizioni visive` e `opacità` per migliorare l'interfaccia visiva.

Listing 6.8: Utilizzo di Positioned e altri elementi grafici

```
Positioned(
  left: 278,
  top: 19,
```

```
  child: Opacity(
    opacity: 0.5,
    child: Container(
      width: 60,
      height: 60,
      clipBehavior: Clip.antiAlias,
      decoration: BoxDecoration(
        border: Border.all(
          width: 12,
          color: const Color(0xFF103DE5),
        ),
        borderRadius: BorderRadius.circular(30),
      ),
    ),
  ),
)
```

Questi effetti vengono riprodotti anche nel pulsante della schermata di login, che utilizza un textttStack di elementi grafici per migliorare l'appeal visivo e fornire un'esperienza utente dinamica.

Il design della GUI per l'app single-vendor si concentra sul mantenere l'interfaccia intuitiva e coinvolgente per gli utenti, attraverso l'uso di componenti grafiche ben strutturate e animazioni fluide. Le scelte grafiche, insieme all'uso estensivo di `Semantics`, garantiscono inoltre che l'app sia accessibile per tutti gli utenti. Inoltre, è stato implementato anche `translate` per garantire l'accessibilità a persone che parlano lingue diverse, rendendo l'app fruibile e comprensibile a un pubblico internazionale.

In sintesi, l'app single-vendor utilizza un'architettura modulare per gestire la complessità delle interfacce utente, migliorando l'esperienza utente e assicurando una manutenzione più semplice del codice nel lungo periodo.

6.4.3.2 package

Il file `pubspec.yaml` di un'app Flutter gestisce le dipendenze necessarie per il progetto. Di seguito vengono elencate le librerie utilizzate per le due versioni dell'app: multi-vendor e single-vendor. Queste librerie sono fondamentali per la gestione della UI, della logica di autenticazione e delle funzionalità aggiuntive.

Librerie comuni Le seguenti librerie sono utilizzate sia nella versione multi-vendor che in quella single-vendor dell'app:

- **cupertino_icons:** `1.0.2`

  Fornisce icone in stile Cupertino per l'interfaccia utente di iOS.

  Disponibile su `https://pub.dev/packages/cupertino_icons`.

- **cached_network_image:** `3.4.0`

  Permette di caricare e memorizzare nella cache le immagini da una rete.

  Disponibile su `https://pub.dev/packages/cached_network_image`.

- **firebase_core:** `3.3.0`

  Fornisce l'integrazione di base con Firebase.

  Disponibile su `https://pub.dev/packages/firebase_core`.

- **firebase_auth:** `5.1.3`

  Consente l'autenticazione degli utenti tramite Firebase.

  Disponibile su `https://pub.dev/packages/firebase_auth`.

- **cloud_firestore:** `5.2.0`

  Permette l'interazione con il database Firestore di Firebase.

  Disponibile su `https://pub.dev/packages/cloud_firestore`.

- **image_picker:** `1.1.2`

  Consente di selezionare immagini e video dalla galleria o dalla fotocamera.

  Disponibile su `https://pub.dev/packages/image_picker`.

- **flutter_easyloading:** `3.0.5`

  Fornisce un'interfaccia semplice per la visualizzazione di indicatori di caricamento.

  Disponibile su `https://pub.dev/packages/flutter_easyloading`.

- **uuid:** `4.4.2`

  Genera identificatori univoci universali.

  Disponibile su `https://pub.dev/packages/uuid`.

Librerie specifiche per la versione multi-vendor Le seguenti librerie sono utilizzate esclusivamente nella versione multi-vendor dell'app:

- **flutter_svg:** `2.0.9`

  Permette di caricare e visualizzare immagini SVG.

  Disponibile su `https://pub.dev/packages/flutter_svg`.

- **shimmer_animation:** `2.1.0+1`

  Aggiunge effetti di shimmer (luccichio) alle animazioni.

  Disponibile su `https://pub.dev/packages/shimmer_animation`.

- **firebase_ui_auth:** `1.13.1`

  Fornisce widget di autenticazione predefiniti per Firebase.

  Disponibile su `https://pub.dev/packages/firebase_ui_auth`.

- **country_state_city_picker:** `1.2.8`

  Offre una selezione di paesi, stati e città.

  Disponibile su `https://pub.dev/packages/country_state_city_picker`.

- **provider:** `6.1.2`

  Gestisce la fornitura di oggetti e dipendenze in modo reattivo.

  Disponibile su `https://pub.dev/packages/provider`.

- **date_time_format:** `2.0.1`

  Fornisce formattazione avanzata di date e orari.

  Disponibile su `https://pub.dev/packages/date_time_format`.

- **photo_view:** `0.14.0`

  Consente la visualizzazione e lo zoom delle immagini.

  Disponibile su `https://pub.dev/packages/photo_view`.

- **flutter_slidable:** `3.1.0`

  Implementa effetti di scorrimento per le azioni degli elementi della lista.

  Disponibile su `https://pub.dev/packages/flutter_slidable`.

Librerie specifiche per la versione single-vendor Le seguenti librerie sono utilizzate esclusivamente nella versione single-vendor dell'app:

- **google_fonts:** `6.2.1`

  Fornisce l'accesso a un'ampia collezione di font di Google.

  Disponibile su `https://pub.dev/packages/google_fonts`.

- **get:** `4.6.6` Un potente strumento per la gestione dello stato e la navigazione. Utilizzato per la gestione dello stato dei provider, aggiornando gli stati del carrello e dei preferiti

  Disponibile su `https://pub.dev/packages/get`.

- **flutter_riverpod:** `2.5.1`

  Una libreria per la gestione dello stato reattivo con Riverpod.

  Disponibile su `https://pub.dev/packages/flutter_riverpod`.

- **badges:** `3.1.2`

  Aggiunge badge e indicatori ai widget. Utilizzato per gli elementi di notifica dei cambiamenti di stato, carrello, preferiti, messaggi

  Disponibile su `https://pub.dev/packages/badges`.

- **flutter_rating_bar:** `4.0.1`

  Fornisce una barra di valutazione con stelle per le recensioni dei prodotti.

  Disponibile su `https://pub.dev/packages/flutter_rating_bar`.

- **country_state_city_picker_2:** `1.0.5` Un'alternativa aggiornata per la selezione di paesi, stati e città. Utilizzato per compilare i form dell'indirizzo di spedizionie.

  Disponibile su `https://pub.dev/packages/country_state_city_picker_2`.

- **flutter_local_notifications:** `17.2.2`

  Consente la gestione delle notifiche locali.

  Disponibile su `https://pub.dev/packages/flutter_local_notifications`.

- **shared_preferences:** `2.3.2`

  Permette di salvare e recuperare dati semplici e persistenti.

  Disponibile su `https://pub.dev/packages/shared_preferences`.

- **flutter_localization:** `0.2.2`

  Gestisce la localizzazione e le traduzioni all'interno dell'app.

  Disponibile su `https://pub.dev/packages/flutter_localization`.

- **flutter_translate:** `4.1.0`

  Supporta la traduzione dei testi in più lingue.

  Disponibile su `https://pub.dev/packages/flutter_translate`.

- **intl:** `0.19.0`

  Fornisce funzionalità per la localizzazione e formattazione dei testi.

  Disponibile su `https://pub.dev/packages/intl`.

Queste librerie contribuiscono in modo significativo a diverse aree dell'applicazione, dalla gestione dello stato e della navigazione alle animazioni e all'internazionalizzazione. Utilizzando librerie specifiche per le esigenze di ciascuna versione dell'app, è possibile ottimizzare le funzionalità e migliorare l'esperienza utente.

`pub.dev` è un repository cruciale per lo sviluppo di applicazioni Flutter, fungendo da principale punto di riferimento per le librerie e i pacchetti che arricchiscono l'ecosistema Flutter. Questo servizio non solo facilita l'integrazione di pacchetti esterni, ma offre anche un'ampia selezione di strumenti e risorse che possono accelerare il processo di sviluppo e migliorare la qualità del software. Grazie alla possibilità di visualizzare valutazioni, recensioni e documentazione dettagliata per ciascun pacchetto, gli sviluppatori possono prendere decisioni informate sulla selezione dei pacchetti più adatti alle loro esigenze. Inoltre, `pub.dev` consente di gestire e aggiornare facilmente le dipendenze del progetto, garantendo l'accesso alle versioni più recenti e sicure delle librerie utilizzate. In questo contesto, l'uso di `pub.dev` si rivela essenziale per mantenere le applicazioni aggiornate e ottimizzate, facilitando l'adozione delle migliori pratiche di sviluppo e la gestione delle dipendenze.

6.4.3.3 Notifiche

L'applicazione implementa un sistema di notifiche per tenere l'utente informato su eventuali nuovi messaggi o eventi. Le notifiche vengono mostrate tramite icone che appaiono nell'interfaccia utente e aggiornano l'utente in tempo reale, garantendo un'esperienza interattiva e reattiva.

Messaggi non letti L'icona del messaggio (rappresentata da una busta) nell'interfaccia della schermata principale indica se ci sono nuovi messaggi non letti. Per visualizzare i messaggi non letti, viene utilizzata una funzionalità basata su un contatore che mostra il numero totale di messaggi non letti. Il seguente frammento di codice mostra come l'app gestisce il conteggio dei messaggi non letti:

Listing 6.9: Conteggio dei messaggi non letti

```
int unreadMessageCount = 0;
List<Map<String, dynamic>> unreadMessages = [];

Future<void> _getTotalUnreadMessagesCount() async {
    final result = await _controller.getUnreadMessages();
    setState(() {
      unreadMessageCount = result['totalUnreadCount'] ?? 0;
      unreadMessages = result['unreadMessages'] ?? [];
```

```
    });
}
```

L'icona del messaggio contiene anche un badge rosso che visualizza il numero di messaggi non letti tramite il widget `Positioned` e `Container`. Di seguito è riportato l'esempio:

Listing 6.10: Badge di notifica dei messaggi non letti

```
if (unreadMessageCount > 0)
  Positioned(
    right: 0,
    top: 0,
    child: Container(
      padding: const EdgeInsets.all(2),
      decoration: BoxDecoration(
        color: Colors.red,
        borderRadius: BorderRadius.circular(12),
      ),
      constraints: const BoxConstraints(
        minWidth: 20,
        minHeight: 20,
      ),
      child: Text(
        unreadMessageCount.toString(),
        style: const TextStyle(
          color: Colors.white,
          fontSize: 12,
          fontWeight: FontWeight.bold,
        ),
        textAlign: TextAlign.center,
      ),
    ),
  ),
```

Interazione con le notifiche Gli utenti possono interagire con le notifiche di nuovi messaggi selezionando l'icona del messaggio nell'interfaccia. Quando si tocca l'icona, viene visualizzato un menu a tendina con l'elenco dei messaggi non letti. Ogni voce nel menu mostra il nome del mittente, il testo del messaggio e un timestamp. Se l'utente seleziona un messaggio, viene rediretto alla schermata di dettaglio della chat tramite la funzione `Navigator.push`. Ecco come viene gestita l'interazione:

Listing 6.11: PopupMenuButton dei messaggi non letti ordinati dal più recente che permette il reindirizzamento a ChatDetailScreen del messaggio sellezionato

```
PopupMenuButton<String>(
  icon: const Icon(Icons.message, color: Colors.white),
  onSelected: (String senderId) {
    final selectedMessage = unreadMessages.firstWhere(
      (message) => message['sender'] == senderId,
      orElse: () => {},
    );
    final chatId = selectedMessage['chatId'] ?? '';
    _navigateToChatDetailScreen(senderId, chatId);
  },
  itemBuilder: (BuildContext context) {
    return unreadMessages.map((message) {
      return PopupMenuItem<String>(
        value: message['sender'],
        child: ListTile(
          leading: CircleAvatar(
            backgroundImage: message['senderImage'] != null &&
                             message['senderImage'].isNotEmpty
                ? NetworkImage(message['senderImage'])
                : const AssetImage('assets/icons/profile.png'),
          ),
          title: Text(message['senderName']),
          subtitle: Column(
            crossAxisAlignment: CrossAxisAlignment.start,
            children: [
              Text(
                message['text'],
                maxLines: 1,
                overflow: TextOverflow.ellipsis,
              ),
              const SizedBox(height: 4),
              Text(
                '${timestamp.day}/${timestamp.month}/${timestamp.year} '
                '${timestamp.hour}:${timestamp.minute.toString().padLeft(2, '0')}'
                    ,
                style: const TextStyle(
                  fontSize: 12,
                  color: Colors.grey,
                ),
              ),
            ],
          ),
        ),
      );
    }).toList();
  },
)
```

Visualizzazione dei Messaggi nella ChatScreen Nella ChatScreen, la visualizzazione dei messaggi avviene tramite l'interfaccia di chat. Ecco come funziona:

- **Recupero delle Chat Attive:** Utilizziamo Firestore per recuperare le chat attive per l'utente corrente. Il metodo `_getOngoingChats()` recupera i documenti delle chat che contengono l'ID dell'utente attuale tra i partecipanti e li ordina per timestamp dell'ultimo messaggio.
- **Conteggio dei Messaggi Non Letti:** La funzione `_getUnreadMessagesCount(chatId)` conta i messaggi non letti per una chat specifica. Questo viene fatto verificando se l'ID dell'utente corrente è presente nella lista `viewedBy` di ciascun messaggio.
- **Visualizzazione della Chat e delle Notifiche:** Le chat vengono visualizzate in un elenco (`ListView`). Per ciascuna chat, viene mostrato il nome del partecipante, l'ultima anteprima del messaggio e, se ci sono messaggi non letti, un badge con il conteggio dei messaggi non letti.

Listing 6.12: Porzione di codice di ChatScreen per la visualizzazione delle chat in corso

```
return FutureBuilder<DocumentSnapshot>(
  future: _firestore.collection('buyers').doc(otherParticipantId).get(),
  builder: (context, buyerSnapshot) {
    if (buyerSnapshot.connectionState == ConnectionState.waiting) {
      return const SizedBox(); // Placeholder durante il caricamento
    }
    if (buyerSnapshot.hasError) {
      return Center(child: Text('Error loading buyer: ${buyerSnapshot.error}'));
    }
    if (!buyerSnapshot.hasData || !buyerSnapshot.data!.exists) {
      return const SizedBox(); // Placeholder se non esiste
    }

    final buyerData = buyerSnapshot.data!.data() as Map<String, dynamic>;
    final fullName = buyerData['fullName'] ?? 'Unknown';
    final profileImage = buyerData['profileImage'] ?? '';

    return FutureBuilder<int>(
```

```
      future: _getUnreadMessagesCount(chatDoc.id),
      builder: (context, unreadSnapshot) {
        final unreadCount = unreadSnapshot.data ?? 0;

        return ListTile(
          leading: CircleAvatar(
            backgroundImage: profileImage.isNotEmpty
                ? NetworkImage(profileImage)
                : const AssetImage('assets/icons/profile.png') as ImageProvider,
          ),
          title: Text(fullName),
          subtitle: Text(chatData['lastMessage'] ?? 'No messages yet'),
          trailing: unreadCount > 0
              ? CircleAvatar(
                  backgroundColor: Colors.red,
                  radius: 12,
                  child: Text(
                    unreadCount.toString(),
                    style: const TextStyle(color: Colors.white, fontSize: 12),
                  ),
                )
              : null,
          onTap: () {
            Navigator.push(
              context,
              MaterialPageRoute(
                builder: (context) => ChatDetailScreen(
                  receiverId: otherParticipantId,
                  receiverFullName: fullName,
                  receiverImage: profileImage,
                  chatId: chatDoc.id,
                ),
              ),
            );
          },
        );
      },
    );
  },
);
```

Gestione delle Notifiche con get per il Carrello e i Preferiti Nel caso del carrello e dei preferiti, la gestione delle notifiche viene effettuata tramite provider e il pacchetto `get`. Ecco come viene gestito:

- **Provider per il Carrello:** Utilizziamo `flutter_riverpod` per gestire lo

stato del carrello. Le notifiche per il numero di articoli nel carrello vengono visualizzate tramite un badge sull'icona del carrello.

- **Provider per i Preferiti:** Sebbene non sia stato mostrato direttamente, la stessa logica si applica ai preferiti. Usando `flutter_riverpod`, possiamo mantenere lo stato dei preferiti e aggiornare l'interfaccia utente di conseguenza.

Listing 6.13: Porzione di codice per il carrello con `flutter_riverpod`

```
class CartScreen extends ConsumerStatefulWidget {
  const CartScreen({super.key});

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

class _CartScreenState extends ConsumerState<CartScreen> {
  @override
  Widget build(BuildContext context) {
    final cartData = ref.watch(cartProvider);
    final _cartProvider = ref.read(cartProvider.notifier);
    final totalAmount = ref.read(cartProvider.notifier).calculateTotalAmount();
    return Scaffold(
      appBar: PreferredSize(
        preferredSize: Size.fromHeight(MediaQuery.of(context).size.height * 0.20),
        child: Container(
          width: MediaQuery.of(context).size.width,
          height: 100,
          clipBehavior: Clip.hardEdge,
          decoration: const BoxDecoration(
            image: DecorationImage(
              image: AssetImage('assets/icons/cartb.png'),
              fit: BoxFit.cover,
            ),
          ),
          child: Stack(
            children: [
              Positioned(
                left: 310,
                top: 42,
                child: Stack(
                  children: [
                    Image.asset(
                      'assets/icons/cart.png',
                      width: 40,
                      height: 40,
```

```
                    color: Colors.white,
                  ),
                  Positioned(
                    top: -6,
                    right: 0,
                    child: badges.Badge(
                      badgeStyle: badges.BadgeStyle(
                        badgeColor: Colors.yellow.shade800,
                      ),
                      badgeContent: Text(
                        cartData.length.toString(),
                        style: GoogleFonts.lato(
                          color: Colors.white,
                          fontWeight: FontWeight.bold,
                        ),
                      ),
                    ),
                  ),
                ],
              ),
            ),
            Positioned(
              left: 31,
              top: 41,
              child: Text(
                'My Cart',
                style: GoogleFonts.lato(
                  color: Colors.white,
                  fontSize: 18,
                  fontWeight: FontWeight.w600,
                ),
              ),
            ),
          ],
        ),
      ),
    ),
    // ... (resto del codice per la schermata del carrello)
  );
 }
}
```

In conclusione il confronto tra la gestione delle notifiche via Firebase e quella tramite provider mette in evidenza due strategie complementari per il miglioramento dell'esperienza utente:

- **Real-Time Updates vs. State Management:** La modalità basata su Firebase offre aggiornamenti in tempo reale, particolarmente utile per le chat

e le comunicazioni, garantendo che l'utente riceva notifiche immediatamente. Al contrario, la gestione tramite provider fornisce un controllo più centralizzato e predefinito dello stato dell'app, facilitando aggiornamenti coerenti e reattivi per elementi come il carrello e i preferiti.

- **Complexity and Flexibility:** L'approccio Firebase può comportare una maggiore complessità nella configurazione e nella gestione delle connessioni in tempo reale, mentre l'uso di provider semplifica la gestione dello stato e l'aggiornamento dell'interfaccia utente. Tuttavia, la combinazione di entrambi i metodi può offrire una soluzione completa e robusta per le notifiche e la gestione dello stato.

Entrambe le modalità hanno i propri punti di forza e possono essere implementate in modo complementare per ottimizzare l'esperienza dell'utente. Utilizzare Firebase per aggiornamenti in tempo reale e provider per una gestione centralizzata dello stato consente di ottenere un'app altamente reattiva e ben gestita, soddisfacendo le esigenze diverse degli utenti.

6.4.4 View dell'applicazione

In questa sezione andremo ad esplorare le diverse schermate dell'applicazione. Gli screenshot sono stati presi indistintamente dagli emulatori Android e iOS, precisamente dai dispositivi Google Pixel 3 e iPhone 15 Pro Max, per enfatizzare la capacità multiplatform di Flutter.

6.4.4.1 Multi-vendor

Lato Vendor In figura 6.7, possiamo vedere il design predefinito dell'autenticazione tramite la libreria `FirebaseUIAuth`.

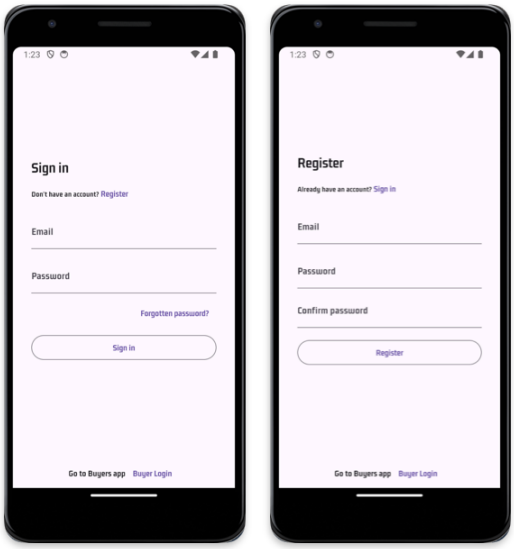


Figure 6.7: Screen di autenticazione lato vendor

Nelle schermate in figura 6.8 possiamo osservare le pagine di navigazione per il lato venditore. In queste pagine, è possibile visualizzare il totale dei guadagni, il numero degli ordini, modificare i prodotti inseriti e monitorare lo stato degli ordini.

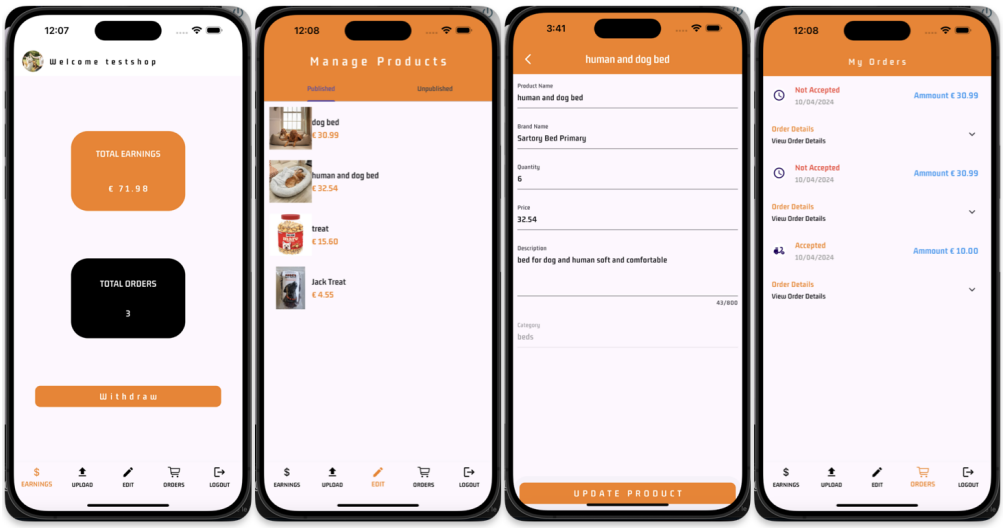


Figure 6.8: Pagine di navigazione lato venditore

Le seguenti pagine 6.9 mostrano i form di compilazione richiesti per caricare

un nuovo prodotto, nome prodotto, descrizione, marca, taglia, quantità, categoria, prezzo, prezzo di spedizione, foto ecc.

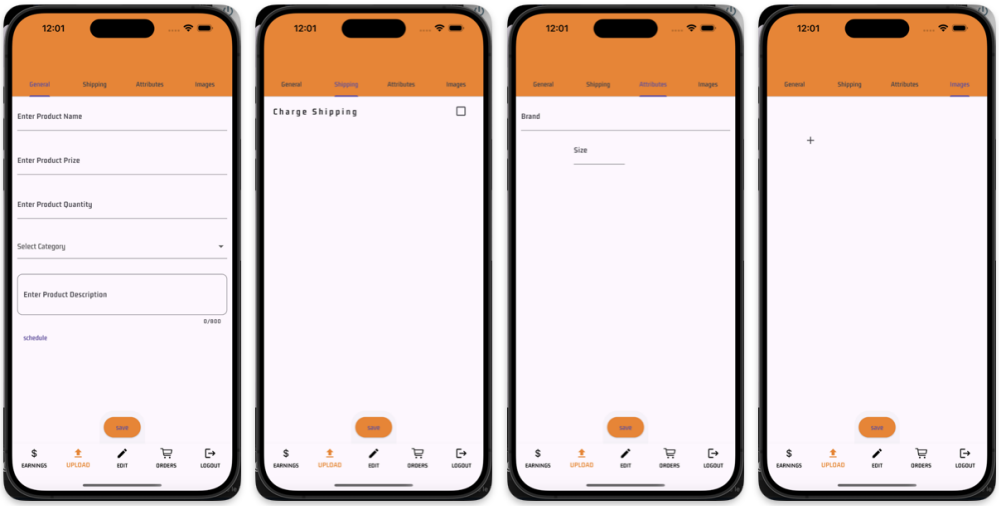


Figure 6.9: Pagine per compilare i form dei prodotti

Lato Buyer In figura 6.10 mostra l'autenticazione personalizzata con controllo utente gestito tramite `_authController.loginUser(email, password)`.

In questa schermata è possibile notare una maggiore personalizzazione, che include i pulsanti e il form di registrazione. Quest'ultimo è stato modificato per permettere l'inserimento di un maggior numero di campi.

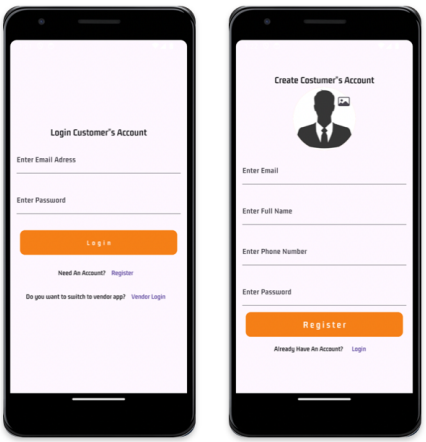


Figure 6.10: Pagine di autenticazione dei clienti

Le pagine seguenti 6.11 mostrano la home page dove i prodotti sono rappresentati con foto, nome e prezzo. La pagina dedicata alle categorie mostra immagine e nome della categoria e quella dedicata ai negozi foto nome e nazionalità di provenienza dei negozi.

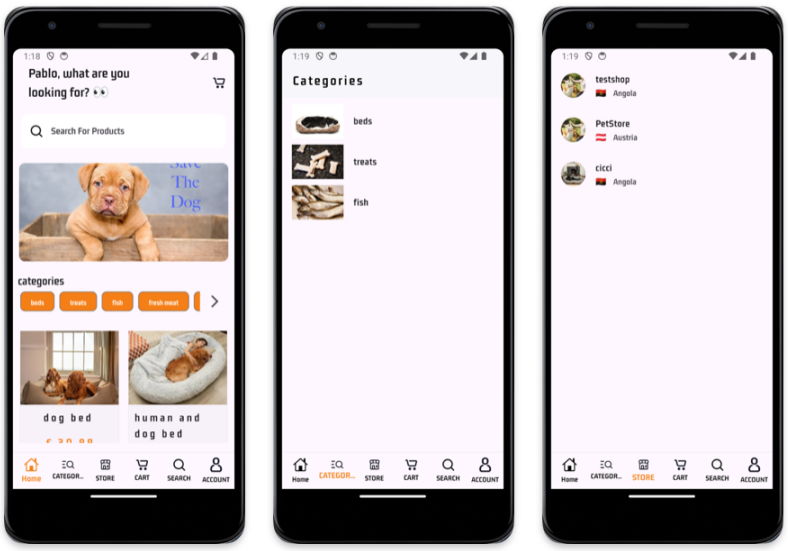


Figure 6.11: Pagine di navigazione clienti

In figura 6.12 è possibile vedere per prima la pagina del carrello in cui è possibile ridurre/incrementare le quantità dei singoli prodotti scelti oppure eliminarli, eventualmente è anche possibile svuotare l'intero carrello. La seconda è la pgina di ricerca in cui è sufficiente digitare sulla barra di ricerca per visionar ei prodotti. Per ultima si vede la pagina del profilo

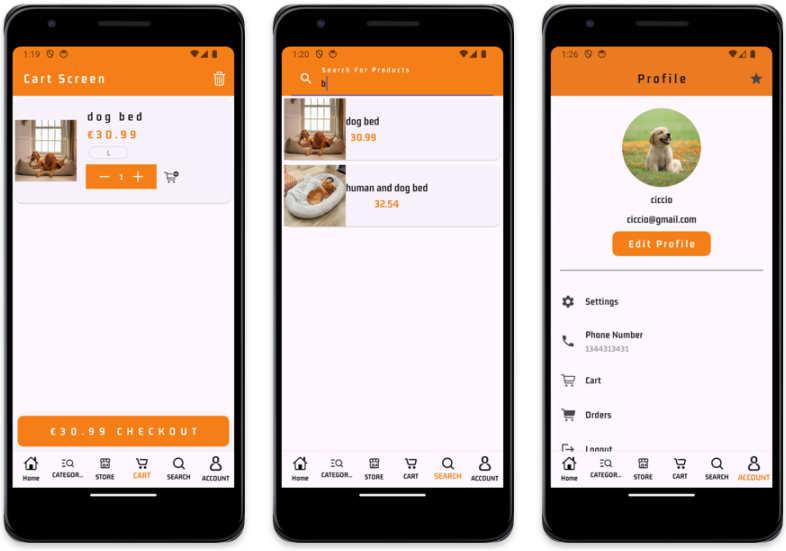


Figure 6.12: Pagine di navigazione clienti

Le pagine di 6.13 rappresentano la navigazione interna nel caso dela pagina delle categorie, la pgina del prodotto nel dettaglio e la pagina responsabile di mostrare i prodotti relatovi a un negozio.

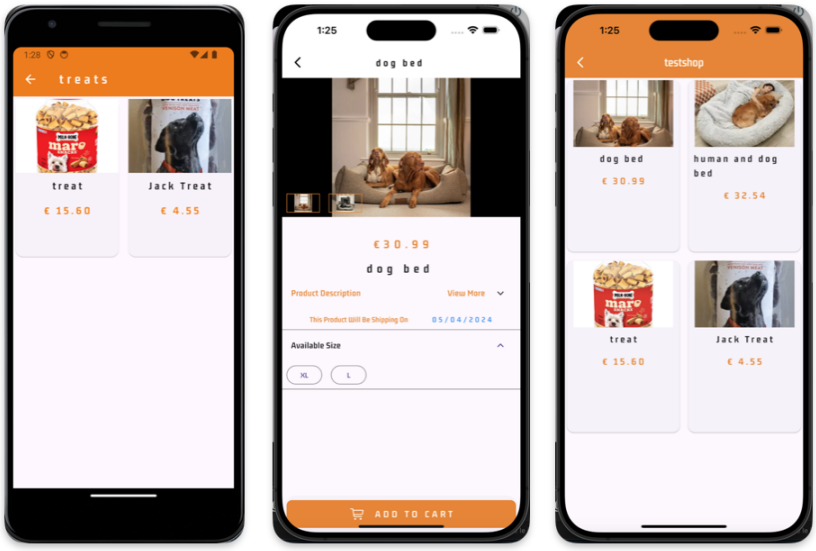


Figure 6.13: Pagine di navigazione interna dei clienti

In fine a 6.14 possiamo visionare la pagina di checkout, quella di editing del profilo e quella dello stato degli ordini

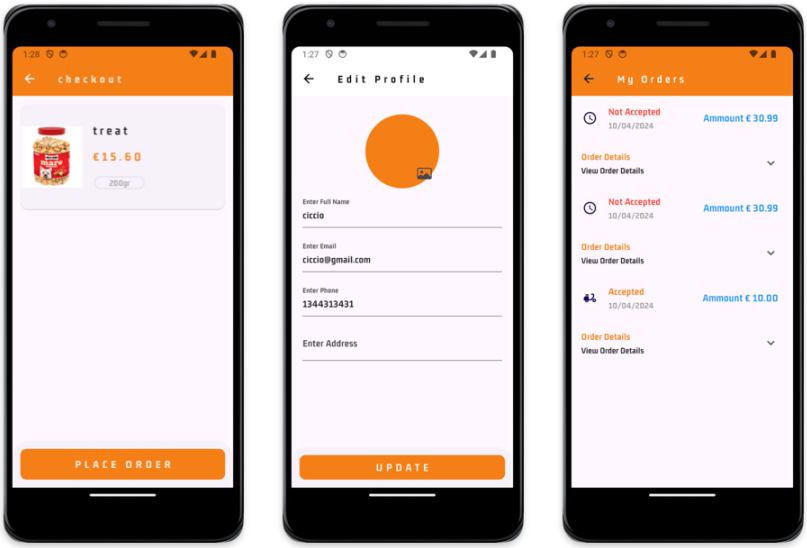


Figure 6.14: Pagine di navigazione interna dei clienti

6.4.4.2 Single-vendor

Per quanto riguarda l'applicazione Single Vendor possiamo notare subito in figura 6.15 un design più moderno con un approccio multilinguistico.

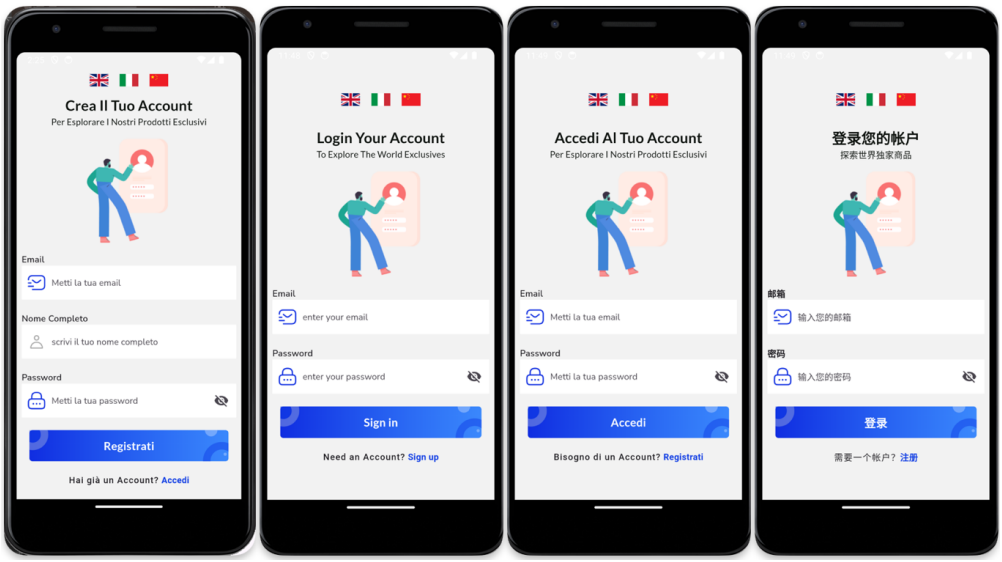


Figure 6.15: Autenticazione Single-vendor

Questo approccio si può vedere una volta effettuato il login anche nella home

page.6.16 dove è possibile notare anche la gestione delle notifiche dei messaggi, una volta cliccata l'icona apposita, tramite un menù a tendina.

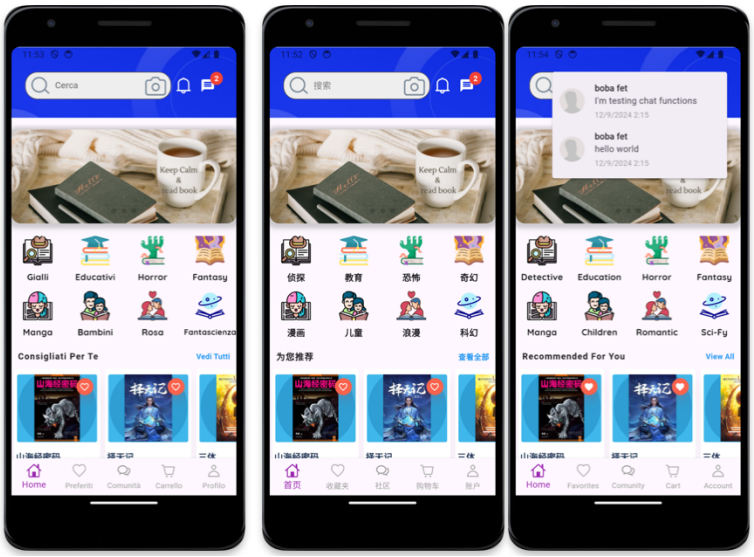


Figure 6.16: Home page

Di seguito 6.17, sono raffigurate le pagine relaative ai prodotti, la prima è quella di dettaglio del prodotto. Nelle due figure centrali invece possiamo vedere la pagina di dettaglio di un ordine in cui l'ultimo bottone in basso serve per lasciare una recensione ed è visibile solo se l'ordine risulta consegnato, in oltre come in questo caso il bottone appare con il messaggio 'update' (aggiorna) nel caso sia già stato effettuata una recensione da parte del cliente per il prodotto inquestione. L'ultima pagina invece illustra la sezione dei preferiti.

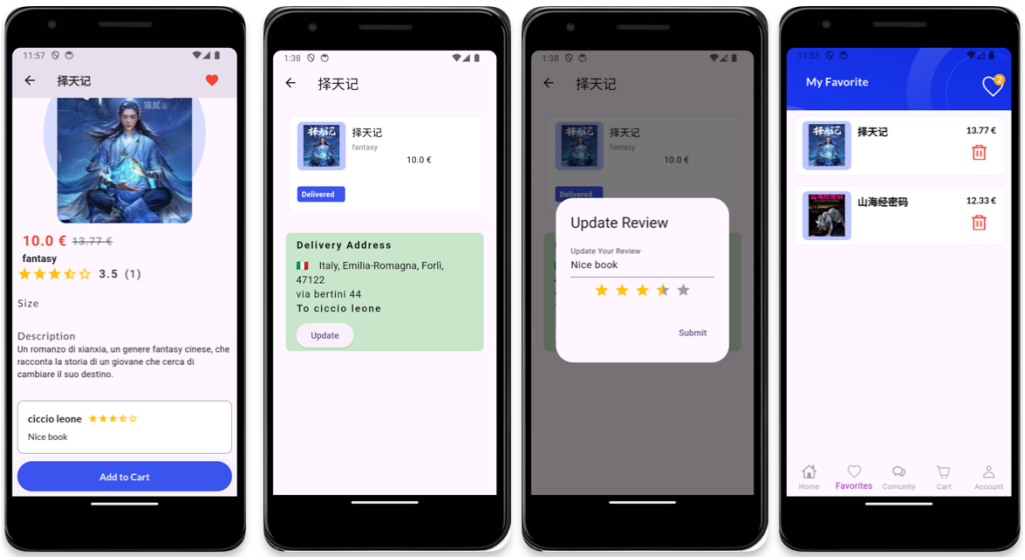


Figure 6.17: Pagine relative ai prodotti

In figura 6.18, possiamo vedere, per prima la schermata dedicata alla sellezione per categoria e di seguito invece le schermate dedicate al checkout, dal carrello, alla scelta del metodo di pagamento, fino all'inserimento dell'indirizzo

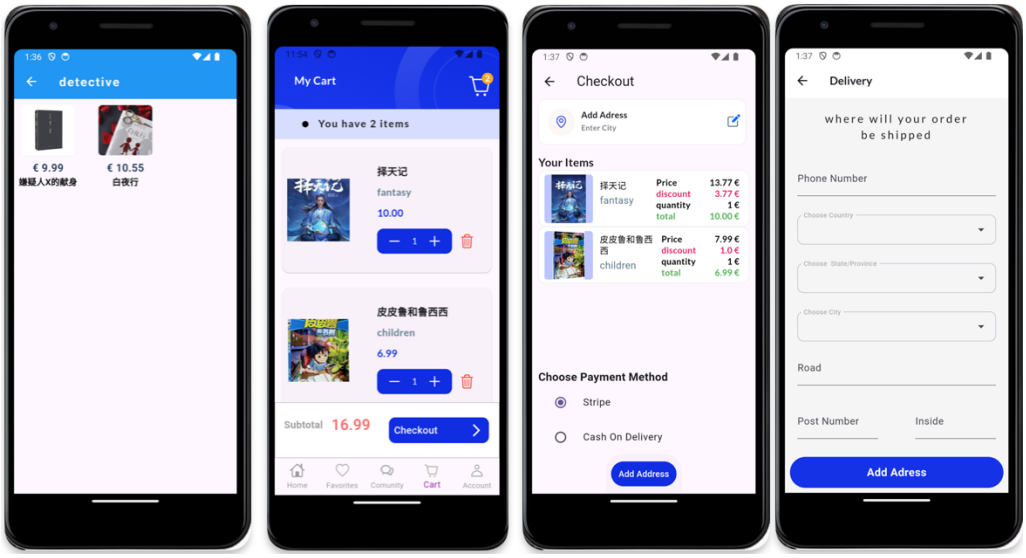


Figure 6.18: Chokout screens

Le pagine che più distinguo l'applicazione Singl-vendor sono visibili in figura 6.19, che rappresentano le pagine destinate alla comunity, dalla visione dei post

all'inserimento di un nuovo post tramite il pulsante centrale '+', fino alla visione dell'anteprima dei messaggi che permette di essere reindirizzati alla chat.

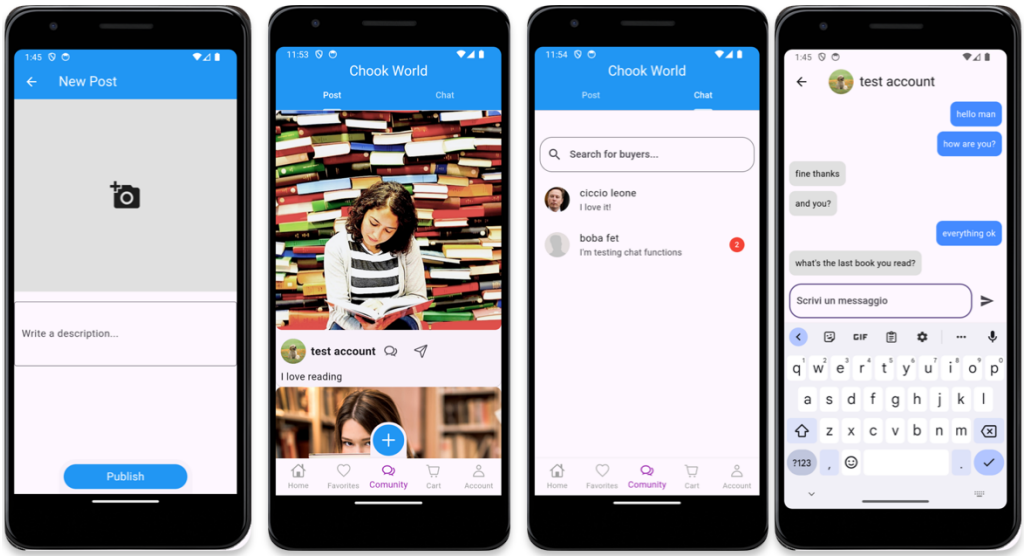


Figure 6.19: Pagine della Comunity

In fine è possibile visionale le pagine per la gestione del profilo 6.20, con la sezione di editing e la visualizzazione degli stati degli ordini

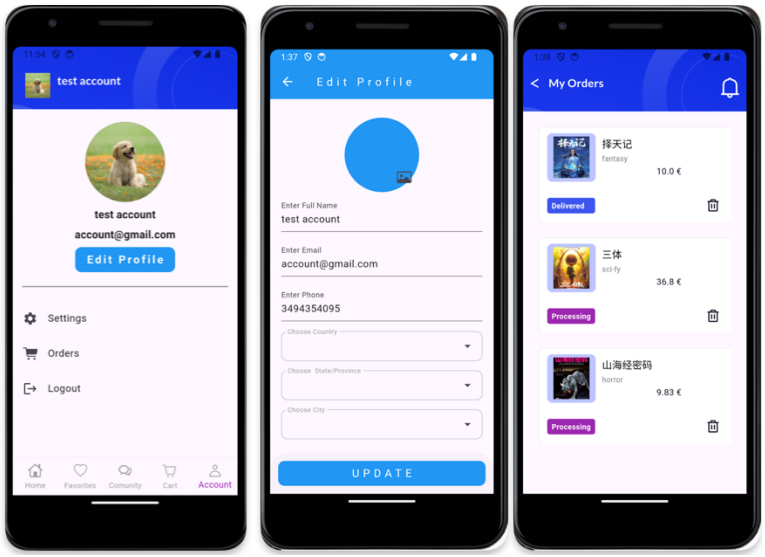


Figure 6.20: Pagine del profilo

Chapter 7

Considerazioni sull'Usabilità e il Design

7.1 Ruolo cruciale del design nell'esperienza utente

L'usabilità e il design sono aspetti fondamentali nello sviluppo di applicazioni moderne. Un'interfaccia ben progettata non solo aumenta la facilità d'uso, ma rende l'interazione più intuitiva e piacevole per l'utente. In particolare, il design dell'interfaccia deve bilanciare estetica e funzionalità per garantire un'esperienza fluida e accessibile a tutti gli utenti, indipendentemente dalle loro competenze tecniche o fisiche.

Nel contesto dello sviluppo dell'applicazione *Stingle-vendor*, l'uso del framework Flutter ha permesso di sfruttare widget predefiniti e personalizzabili per costruire interfacce dinamiche, responsive e accessibili. L'obiettivo principale è garantire che l'interazione con l'applicazione sia ottimale su dispositivi diversi, mantenendo la coerenza visiva e una navigazione chiara. A tal proposito, è stato adottato un approccio incentrato sull'utente, focalizzandosi su:

- Facilità di accesso a funzioni principali.
- Semplicità d'uso delle interfacce.

- Personalizzazione dell'esperienza utente (ad esempio, la selezione della lingua).
- Design responsivo per adattarsi a diverse risoluzioni di schermo.

7.2 Principi di design adottati per migliorare l'usabilità

Il codice seguente rappresenta la schermata di login, che è una delle prime interazioni che l'utente ha con l'applicazione. Il design di questa schermata incorpora vari aspetti dell'usabilità, come la possibilità di cambiare lingua, feedback in tempo reale sull'accesso e una disposizione chiara e intuitiva dei componenti.

7.2.1 Selezione della lingua

Nella schermata di login, una caratteristica fondamentale per l'inclusività è la possibilità di selezionare la lingua preferita (inglese, italiano o cinese). Questo è realizzato con una semplice fila di pulsanti che visualizzano le bandiere dei rispettivi paesi:

Listing 7.1: Selezione della lingua nella schermata di login

```
Row(
    mainAxisAlignment: MainAxisAlignment.center,
    children: [
        IconButton(
            icon: Image.asset(
                'assets/icons/uk_flag.png',
                width: 30,
                height: 20,
                fit: BoxFit.cover,
                semanticLabel: translate('en'),
            ),
            onPressed: () => changeLocale(context, 'en'),
            tooltip: translate('en'),
        ),
        IconButton(
            icon: Image.asset(
                'assets/icons/it_flag.png',
                width: 30,
                height: 20,
```

```
                fit: BoxFit.cover,
                semanticLabel: translate('it'),
            ),
            onPressed: () => changeLocale(context, 'it'),
            tooltip: translate('it'),
        ),
        IconButton(
            icon: Image.asset(
                'assets/icons/cn_flag.png',
                width: 30,
                height: 20,
                fit: BoxFit.cover,
                semanticLabel: translate('zh'),
            ),
            onPressed: () => changeLocale(context, 'zh'),
            tooltip: translate('zh'),
        ),
    ],
)
```

Questa funzione garantisce che gli utenti possano selezionare immediatamente la lingua che preferiscono, migliorando l'esperienza complessiva e abbassando le barriere linguistiche nell'uso dell'applicazione, inoltre l'inclusione di etichette semantiche e attributi di accessibilità, come `semanticLabel` associato a `translate`, è un esempio di come l'accessibilità sia una priorità nello sviluppo dell'interfaccia. Questi attributi sono essenziali per supportare i lettori di schermo, rendendo l'app utilizzabile anche da utenti con disabilità visive.

7.2.2 Accessibilità e Feedback Utente

Per migliorare l'accessibilità, i campi del modulo sono accompagnati da etichette e icone intuitive. Inoltre, ogni campo utilizza il componente `Semantics` di Flutter per fornire descrizioni alternative e migliorare la compatibilità con i lettori di schermo. Questo è particolarmente utile per migliorare l'interazione tramite tecnologie assistive:

Listing 7.2: Utilizzo di `Semantic` per migliorare l'accessibilità

```
Semantics(
    label: translate('Email'),
    child: TextFormField(
        onChanged: (value) {
            email = value;
```

```
        },
        validator: (value) {
            if (value!.isEmpty) {
                return translate('enter your email');
            } else {
                return null;
            }
        },
        decoration: InputDecoration(
            fillColor: Colors.white,
            filled: true,
            border: OutlineInputBorder(
                borderRadius: BorderRadius.circular(9),
            ),
            labelText: translate('enter your email'),
            prefixIcon: Padding(
                padding: const EdgeInsets.all(10.0),
                child: Image.asset(
                    'assets/icons/email.png',
                    width: 20,
                    height: 20,
                    semanticLabel: translate("Email icon"),
                ),
            ),
        ),
    ),
)
```

Un altro aspetto critico per l'usabilità è fornire feedback immediato all'utente. Ad esempio, se le credenziali sono corrette, l'applicazione visualizza un messaggio di conferma tramite uno `SnackBar`:

Listing 7.3: `SnackBar` di notifica

```
ScaffoldMessenger.of(context)
    .showSnackBar(SnackBar(content: Text(translate('Logged in'))));
```

Se le credenziali sono errate, viene mostrato un messaggio di errore simile.

7.2.3 Esclusione di elementi non necessari con `ExcludeSemantics`

Alcuni elementi visivi dell'interfaccia non sono essenziali per l'esperienza utente e possono confondere gli utenti che utilizzano lettori di schermo. Flutter offre il widget `ExcludeSemantics`, che permette di escludere determinati elementi dall'albero

semantico, rendendo l'esperienza di navigazione più fluida per gli utenti che utilizzano strumenti di accessibilità.

Ad esempio, l'illustrazione nella schermata di login non è necessaria per la comprensione dell'interfaccia e viene quindi esclusa:

Listing 7.4: Utilizzo di `ExcludeSemantics`

```
ExcludeSemantics(
    child: Image.asset(
        'assets/images/Illustration.png',
        width: 200,
        height: 200,
        semanticLabel: translate("Login illustration"),
    ),
)
```

Questo approccio riduce il rumore semantico, consentendo agli utenti di concentrarsi sulle informazioni rilevanti senza essere distratti da contenuti puramente decorativi. Anche se l'elemento visivo può migliorare l'estetica dell'interfaccia, la sua esclusione dall'albero semantico garantisce che non appesantisca l'esperienza di navigazione per utenti con disabilità visive.

7.2.4 Animazioni e Transizioni per una Migliore Esperienza Utente

Flutter supporta animazioni fluide che migliorano l'interazione. Un esempio è l'uso di `CircularProgressIndicator` durante il caricamento:

Listing 7.5: Utilizzo di `CircularProgressIndicator`

```
Center(
    child: _isLoading
        ? CircularProgressIndicator(
            color: Colors.white,
        )
        : Text(
            translate('Sign in'),
            style: GoogleFonts.getFont(
                'Lato',
                color: Colors.white,
                fontSize: 18,
                fontWeight: FontWeight.bold,
            ),
        ),
```

```
15 )
```

Questo tipo di feedback visivo immediato contribuisce a migliorare l'esperienza utente, poiché gli utenti possono vedere chiaramente che il sistema sta elaborando la loro richiesta, riducendo così l'incertezza.

7.3 Conclusioni

L'implementazione di una buona usabilità e accessibilità è stata un obiettivo chiave nello sviluppo dell'interfaccia utente. Grazie all'uso di strumenti di Flutter come `Semantics`, `ExcludeSemantics`, etichette alternative e animazioni, l'applicazione offre un'esperienza utente inclusiva, intuitiva e piacevole. L'approccio adottato garantisce che anche utenti con disabilità possano accedere facilmente alle funzioni dell'app, riducendo le barriere e migliorando l'accessibilità globale dell'applicazione.

Chapter 8

Valutazione e Conclusione

8.1 Rispetto degli Obiettivi Iniziali del Progetto

L'obiettivo principale del progetto era la realizzazione di un'applicazione e-commerce utilizzando Flutter, in grado di consentire agli utenti di visualizzare prodotti, aggiungerli al carrello e procedere all'acquisto. Sono state sviluppate due soluzioni, una *multi-vendor* e una *single-vendor*, ciascuna con caratteristiche specifiche e focalizzata su casi d'uso differenti.

8.1.1 Soluzione Multi-vendor

Nel caso della soluzione multi-vendor, l'applicazione consente a piccoli commercianti di registrarsi e vendere prodotti per animali domestici. Questa soluzione ha soddisfatto gli obiettivi principali, permettendo agli utenti di:

- Visualizzare i prodotti,
- Aggiungere articoli al carrello,
- Procedere all'acquisto.

Anche se tutte queste funzionalità sono state implementate, per motivi pratici, non è stato aggiunto il supporto per l'integrazione con servizi di pagamento e le relative API per transazioni con carte di credito. L'inclusione di tali funzionalità avrebbe reso l'applicazione pronta per l'uso in un ambiente di produzione, ma

il focus del progetto era limitato a un prototipo funzionante. Di conseguenza, entrambi gli scenari (*multi-vendor* e *single-vendor*) non includono un flusso di pagamento completo.

Dal lato utente, l'applicazione multi-vendor presenta diverse funzionalità, come la homepage, le categorie di prodotti, la gestione del carrello e l'account utente. Dal lato venditore, l'applicazione permette di gestire i guadagni, caricare nuovi prodotti e monitorare gli ordini.

In aggiunta, è stata creata una piattaforma web gestionale, che consente di gestire i venditori, i prodotti e altre funzioni amministrative. Questo approccio ha garantito una separazione chiara tra l'esperienza dell'utente finale e quella del venditore, semplificando la gestione del sistema da parte di tecnici e operatori.

8.1.2 Soluzione Single-vendor

La soluzione single-vendor si è concentrata sulla vendita di libri cinesi, mettendo un particolare accento sull'accessibilità e il supporto multilinguistico. In questo contesto, sono stati rispettati tutti gli obiettivi iniziali, compresa l'implementazione di funzionalità social che permettono agli utenti di postare e chattare tra loro, rendendo l'esperienza di acquisto più interattiva e coinvolgente.

Anche in questa soluzione, l'integrazione con un servizio di pagamento non è stata implementata, ma a livello funzionale, l'applicazione consente di visualizzare prodotti, aggiungerli al carrello e simulare un processo di acquisto. La piattaforma gestionale per l'applicazione single-vendor è progettata in modo simile a quella multi-vendor, ma adattata alle esigenze di un unico venditore.

8.1.3 Rispetto dei Requisiti

Entrambe le soluzioni soddisfano i requisiti tecnici discussi nella sezione *Obiettivi e Requisiti* (Sezione 6.1). In particolare, l'uso di Firebase per la gestione del database e delle chiamate è stato implementato con successo. Le applicazioni sono in grado di leggere, creare e modificare i dati in tempo reale, garantendo così un'interazione dinamica con il database Firestore.

Flutter si è rivelato una scelta eccellente per lo sviluppo di queste soluzioni, grazie alla sua capacità di supportare più piattaforme con un unico codice base,

riducendo notevolmente i tempi di sviluppo (*time-to-market*) e garantendo una distribuzione su più dispositivi (iOS, Android e web).

8.2 Riflessioni Finali sull'Esperienza di Sviluppo con Flutter

8.2.1 Sviluppo Multiplatform con Flutter

L'utilizzo di Flutter per lo sviluppo di un'applicazione e-commerce ha offerto notevoli vantaggi, soprattutto per quanto riguarda il supporto multiplatform. A differenza di strumenti come Android Studio e Xcode, che forniscono funzionalità come il *drag and drop* per la costruzione di interfacce grafiche senza scrivere codice, Flutter si basa interamente sul codice per la creazione dell'interfaccia utente. Tuttavia, questo approccio ha dimostrato di essere altrettanto efficiente, soprattutto grazie al processo di *hot reload* e *hot restart*, che consente di vedere in tempo reale le modifiche effettuate nel codice.

Questa funzionalità di *hot reload* estende il suo supporto non solo agli emulatori Android e iOS, ma anche alle applicazioni web, consentendo una rapida iterazione e verifica delle funzionalità su tutti i dispositivi contemporaneamente. Questo rappresenta un chiaro vantaggio rispetto a strumenti come Android Studio e Xcode, che limitano l'emulazione solo ai rispettivi dispositivi.

8.2.2 Gestione degli Widget e Responsività

Flutter si è dimostrato uno strumento potente per la creazione di interfacce responsive, ma richiede una profonda comprensione della gerarchia dei widget e delle relazioni tra genitori e figli. Una cattiva gestione della struttura dei widget può compromettere l'esperienza utente, soprattutto quando si sviluppa per diverse piattaforme. Il passaggio da dispositivi mobili a web, in particolare, richiede attenzione per garantire che la risoluzione e l'esperienza utente siano consistenti.

In Flutter, i widget non sono solo componenti visuali, ma strutture fondamentali che determinano il layout e il comportamento dell'interfaccia. La loro gestione richiede pianificazione, poiché cambiamenti nelle dimensioni dello schermo

o nelle proporzioni possono influenzare il risultato finale, in particolare quando si passa da mobile a web. È fondamentale progettare un layout flessibile che si adatti automaticamente a diverse risoluzioni e formati di schermo, per mantenere un'esperienza utente coerente su tutte le piattaforme.

8.2.3 Firebase e Firestore: Integrazione Semplice e Potente

Uno degli aspetti più apprezzabili nello sviluppo dell'app è stato l'uso di Firebase, in particolare Firestore, per la gestione del database. L'integrazione di Firebase in Flutter è risultata fluida e immediata. Firestore ha consentito di eseguire query, leggere e scrivere dati in tempo reale senza incontrare particolari difficoltà tecniche.

Tuttavia, per sfruttare appieno il potenziale di Firebase, è necessario comprendere bene come strutturare le chiamate e ottimizzare le query, specialmente quando si lavora con grandi volumi di dati. Nonostante ciò, con un po' di attenzione e studio, è stato possibile implementare tutte le funzionalità richieste senza problemi significativi.

Oltre alla gestione dei dati, Firebase offre una serie di strumenti aggiuntivi che potrebbero essere esplorati in futuro per migliorare ulteriormente l'esperienza utente. Ad esempio:

- **Firebase Machine Learning**: offre la possibilità di integrare algoritmi di intelligenza artificiale, come il riconoscimento delle immagini o del testo, per fornire raccomandazioni di prodotto più accurate.
- **Firebase Analytics**: consente di raccogliere e analizzare dati sull'utilizzo dell'app da parte degli utenti, fornendo insight preziosi per migliorare le funzionalità e l'esperienza utente.
- **Firebase Cloud Messaging (FCM)**: permette di implementare notifiche push per tenere gli utenti informati su offerte, nuovi prodotti o aggiornamenti.
- **Firebase Predictions**: utilizza il machine learning per creare modelli predittivi sul comportamento degli utenti, consentendo di personalizzare

l'esperienza utente in base alle loro preferenze e abitudini.

8.2.4 Conclusioni sull'Esperienza di Sviluppo

Flutter si è dimostrato un framework estremamente efficiente per lo sviluppo di applicazioni multipiattaforma. Tuttavia, la sua flessibilità richiede una certa padronanza nella gestione dei widget e delle logiche di layout, per garantire che l'esperienza utente sia ottimale su tutte le piattaforme.

L'integrazione con Firebase ha rappresentato un ulteriore punto di forza, semplificando la gestione dei dati e aprendo la strada a funzionalità avanzate che potrebbero essere integrate in futuro, come l'analisi dei dati e l'intelligenza artificiale.

In conclusione, Flutter rappresenta uno strumento potente e versatile, ma come per ogni tecnologia, richiede una comprensione approfondita per poter essere sfruttato al meglio.

Chapter 9

Prospettive Future e Sviluppi del Progetto

9.1 Possibili Estensioni e Miglioramenti Futuri per l'E-commerce

Uno dei miglioramenti essenziali per l'applicazione di rivendita di libri cinesi sarebbe l'integrazione della Google Translate API per la traduzione automatica dei dati dinamici, come i prodotti, le categorie e i banner. Questa funzionalità potrebbe essere ulteriormente estesa per tradurre anche le chat e i post all'interno dell'applicazione, offrendo una comunicazione senza barriere linguistiche tra utenti di lingue diverse.

Un'altra funzionalità importante da implementare è l'uso delle notifiche *push*, permettendo all'app di interagire con il dispositivo e ricevere notifiche anche quando l'applicazione è chiusa o in background. Per questo, si potrebbe utilizzare un package appropriato di Flutter, come `firebase_messaging`, che supporta le notifiche *push* su tutte le piattaforme.

Come citato in precedenza, una caratteristica fondamentale per un e-commerce realmente funzionante è l'integrazione di API per i pagamenti con carte di credito. L'implementazione di una soluzione di pagamento sicura, come Stripe o PayPal, sarebbe cruciale per trasformare l'app in una vera piattaforma di vendita online.

Inoltre, si potrebbero migliorare e completare le piattaforme gestionali web

complementari all'applicazione. Questi gestionali permetterebbero una migliore gestione dei prodotti, degli ordini e delle attività commerciali da parte dei venditori e degli amministratori.

Un altro aspetto da considerare è garantire che tutto il codice sia accessibile e privo di bug, mantenendo la compatibilità multipiattaforma. Estendere l'esperienza multipiattaforma non dovrebbe solo focalizzarsi sul funzionamento, ma anche su come offrire esperienze personalizzate in base alle dimensioni dello schermo. Ciò potrebbe includere interazioni diversificate con gli elementi della GUI, fornendo così un'esperienza utente ottimizzata per dispositivi con schermi di varie dimensioni.

9.2 Considerazioni sulle Nuove Funzionalità o Tecnologie da Esplorare

Integrare tecnologie social all'interno dell'e-commerce rappresenta un'evoluzione interessante e attuale. Aggiungere la possibilità di creare una GUI dedicata e ottimizzata per le interazioni social potrebbe arricchire l'esperienza utente e rendere l'app più coinvolgente. Tuttavia, questo richiederebbe un'accurata ottimizzazione per garantire che l'esperienza social non comprometta le performance della piattaforma e-commerce.

Un'altra funzionalità fondamentale da esplorare è l'aggiunta della gestione delle pubblicità interne all'app, specialmente per la piattaforma multi-vendor. Questo permetterebbe ai venditori di promuovere i propri prodotti all'interno dell'app stessa, migliorando l'esperienza utente e aumentando le opportunità di guadagno per la piattaforma. Allo stesso tempo, sarebbe utile migliorare le opzioni di filtraggio per le ricerche e i prodotti, rendendo più semplice per gli utenti trovare ciò che cercano.

Inoltre, si potrebbero aggiungere form dedicati alla descrizione delle immagini per i prodotti caricati dai venditori. Questi form sarebbero utili per fornire descrizioni accurate delle immagini, consentendo ai non vedenti di comprendere meglio i contenuti attraverso l'uso di tecnologie di lettura dello schermo.

L'utilizzo delle tecnologie offerte da Firebase rappresenta un'opportunità sig-

nificativa per estendere le funzionalità dell'app. Firebase offre una serie di servizi come l'analisi dei dati, la gestione delle notifiche, il machine learning e molto altro. Esplorare l'uso di strumenti come Firebase Predictions e Firebase Analytics potrebbe migliorare l'esperienza utente fornendo consigli personalizzati sui prodotti e insight dettagliati sul comportamento degli utenti.

Per l'applicazione multi-vendor, un altro sviluppo interessante potrebbe essere l'integrazione con le API di Google Ads o Meta. Questo permetterebbe ai venditori non solo di gestire le pubblicità interne all'app, ma anche di estendere le proprie campagne pubblicitarie sui social network internazionali, ampliando il raggio d'azione della propria attività.

9.3 Riflessioni sulle Lezioni Apprese Durante il Processo di Sviluppo

Durante il processo di sviluppo, una delle prime lezioni apprese è stata l'importanza dell'uso dei mockup per ottenere risultati soddisfacenti. Durante lo sviluppo delle due applicazioni, sono stati seguiti diversi tutorial per apprendere i fondamenti di Flutter e comprenderne il funzionamento. Tuttavia, inizialmente mi sono concentrato più sull'implementazione dei requisiti tecnici piuttosto che sull'ottimizzazione dell'esperienza utente. Questo ha portato alla consapevolezza che, per realizzare un'app di successo, l'*user experience* deve essere considerata fin dalle prime fasi del progetto.

Un'altra importante lezione appresa riguarda il corretto utilizzo della gerarchia dei widget e dei *parent* e *child widgets*. Comprendere come questi elementi interagiscono è cruciale per costruire interfacce coerenti e funzionali. In particolare, l'allineamento e la scalabilità dei widget sono aspetti fondamentali per garantire che l'app si adatti correttamente a dispositivi di diverse dimensioni, mantenendo un'esperienza utente ottimale su mobile, tablet e web.

Infine, la creazione di layout responsive è stata un'altra importante lezione appresa. Flutter offre strumenti potenti per costruire layout che si adattano automaticamente a diverse risoluzioni, ma è necessaria una pianificazione attenta per assicurarsi che la GUI risponda correttamente alle variazioni di schermo e man-

tenga un aspetto consistente su tutte le piattaforme. Prestare attenzione a questi dettagli è fondamentale per migliorare l'usabilità e l'accessibilità delle applicazioni sviluppate co flutter.

Bibliography

[1] adtmag. *hybrid-beats-native*. Accessed: 2024-08-31. 2024. URL: https://adtmag.com/articles/2017/07/28/hybrid-beats-native.aspx.

[2] aglowiditsolutions. *cross-platform-app-development-analysis*. Accessed: 2024-08-31. 2024. URL: https://aglowiditsolutions.com/blog/cross-platform-app-development-analysis/.

[3] aglowiditsolutions. *cross-platform-app-development-analysis*. Accessed: 2024-08-31. 2024. URL: https://aglowiditsolutions.com/blog/cross-platform-app-development-analysis/.

[4] amity. *state-management-in-dart-flutter*. Accessed: 2024-09-01. 2024. URL: https://www.amity.co/tutorials/state-management-in-dart-flutter.

[5] appmysite. *top-75-mobile-app-statistics-market-size-usage*. Accessed: 2024-08-30. 2024. URL: https://www.appmysite.com/blog/top-75-mobile-app-statistics-market-size-usage/.

[6] bitcot. *why-choose-firebase-for-app-development*. Accessed: 2024-09-07. 2024. URL: https://www.bitcot.com/why-choose-firebase-for-app-development/.

[7] brandmed. *is-firebase-the-best-back-end-choice-for-a-flutter-app*. Accessed: 2024-09-07. 2024. URL: https://brandmed.com/blog/development/is-firebase-the-best-back-end-choice-for-a-flutter-app#:~:text=Using%20Firebase%20with%20Flutter%20ensures,quite%20reasonable%20and%20widely%20used..

[8] buildfire. *programming-languages-for-mobile-app-development*. Accessed: 2024-08-31. 2024. URL: https://buildfire.com/programming-languages-for-mobile-app-development/.

[9] codemagic. *dart-vs-javascript*. Accessed: 2024-09-01. 2024. URL: https://blog.codemagic.io/dart-vs-javascript/.

[10] codemagic. *dart-vs-kotlin*. Accessed: 2024-09-01. 2024. URL: https://blog.codemagic.io/dart-vs-kotlin/.

[11] colorwhistle. *mobile-app-evolution*. Accessed: 2024-08-29. 2024. URL: https://colorwhistle.com/evolution-of-mobile-apps/.

[12] dart. *language*. Accessed: 2024-09-01. 2024. URL: https://dart.dev/language.

[13] dart. *overview*. Accessed: 2024-09-01. 2024. URL: https://dart.dev/overview#platform.

[14] enterpriseappstoday. *mobile-app-industry-statistics*. Accessed: 2024-08-29. 2024. URL: https://www.enterpriseappstoday.com/stats/mobile-app-industry-statistics.html.

[15] fireart.studio. *top-most-popular-programming-languages-for-mobile-app-development*. Accessed: 2024-08-31. 2024. URL: https://fireart.studio/blog/top-most-popular-programming-languages-for-mobile-app-development/.

[16] flutter. *interactivity*. Accessed: 2024-09-01. 2024. URL: https://docs.flutter.dev/ui/interactivity.

[17] Google. *Flutter Documentation and Resources*. Accessed: 2024-09-05. 2024. URL: https://flutter.dev.

[18] Google. *Google Firebase*. Accessed: 2024-09-07. 2024. URL: https://firebase.google.com.

[19] google. *firebase per flutter*. Accessed: 2024-09-07. 2024. URL: https://firebase.google.com/docs/flutter?hl=it.

[20] jellyfishtechnologies. *best-mobile-app-development-languages*. Accessed: 2024-08-30. 2024. URL: https://www.jellyfishtechnologies.com/best-mobile-app-development-languages/.

[21] jetbrains. *cross-platform-frameworks*. Accessed: 2024-08-29. 2024. URL: https://www.jetbrains.com/help/kotlin-multiplatform-dev/cross-platform-frameworks.html#what-is-a-cross-platform-app-development-framework.

[22] netsolutions. *native-vs-hybrid-vs-cross-platform*. Accessed: 2024-08-31. 2024. URL: https://www.netsolutions.com/insights/native-vs-hybrid-vs-cross-platform/.

[23] statista. *worldwide-developer-survey-most-wanted-platform*. Accessed: 2024-09-07. 2024. URL: https://www.statista.com/statistics/793884/worldwide-developer-survey-most-wanted-platform/.

[24] os-system. *comparison-firebase-with-other-platforms*. Accessed: 2024-09-07. 2024. URL: https://os-system.com/blog/comparison-firebase-with-other-platforms/#alternate-list.

[25] taglineinfotech. *dart-and-java*. Accessed: 2024-09-01. 2024. URL: https://taglineinfotech.com/dart-and-java/#:~:text=Dart%20performs%20better%20as%20it,and%20quickly%20executes%20the%20code.&text=The%20code%20is%20compiled%20to,which%20requires%20JVM%20to%20run..

[26] techreport. *mobile-app-statistics*. Accessed: 2024-08-30. 2024. URL: https://techreport.com/statistics/software-web/mobile-app-statistics/.

[27] Wikipedia. *Cross-platform software*. Accessed: 2024-08-28. 2024. URL: https://en.wikipedia.org/wiki/Cross-platform_software.

[28] Wikipedia. *Flutter (software)*. Accessed: 2024-08-28. 2024. URL: https://en.wikipedia.org/wiki/Flutter_(software).