Politecnico di Torino

Politecnico di Torino

Laurea Magistrale in Ingegneria Informatica

A.a. 2024/2025

Sessione di laurea Aprile 2025

Evoluzione delle Super App: Modelli Architetturali e Soluzioni Offline-First

Relatori:

Luigi De Russis

Roberto De Giuseppe

Candidato:

Lorenzo Greco

Ringraziamenti

Desidero esprimere la mia più sincera gratitudine a tutte le persone che hanno contribuito alla realizzazione di questo lavoro e che mi hanno accompagnato nel mio percorso di crescita professionale e personale.

Un ringraziamento speciale va al professore Luigi De Russis, relatore di questa tesi, per la disponibilità e la gentilezza con cui si è sempre rivolto a me. Il suo supporto e i suoi consigli sono stati preziosi nel guidarmi lungo questo percorso, permettendomi di affrontare il lavoro con maggiore consapevolezza e attenzione ai dettagli.

Un sincero grazie a Roberto De Giuseppe, partner di Blue Reply, per l'opportunità che mi è stata concessa e per aver reso possibile questa esperienza. Avere la possibilità di lavorare in un contesto così stimolante e innovativo è stato fondamentale per il mio sviluppo.

Un sentito riconoscimento ai manager Andrea Marchetti e Lorenzo Cavallera, per essersi presi cura del mio percorso con grande attenzione e professionalità. La loro fiducia in me, la costante disponibilità e la capacità di coinvolgermi attivamente nelle attività aziendali mi hanno permesso di crescere e affrontare ogni sfida con maggiore sicurezza.

Un pensiero di profonda riconoscenza va a Stefano Monti, senior consultant, per il supporto instancabile e la guida sempre presente. La sua esperienza, unita alla pazienza e alla disponibilità con cui mi ha seguito passo dopo passo, è stata un punto di riferimento imprescindibile.

Ora che il mio percorso in Blue Reply prosegue, sono entusiasta di poter continuare a lavorare in un ambiente dinamico e stimolante, con l'opportunità di crescere ulteriormente e affrontare nuove sfide professionali.

Infine, un ringraziamento profondo alla mia famiglia, che con il loro sostegno incondizionato mi hanno accompagnato in ogni fase di questo percorso. Grazie per aver sempre creduto in me, per avermi incoraggiato nei momenti difficili e per aver condiviso con me ogni soddisfazione lungo il cammino. Questa tesi è anche il risultato del vostro affetto e della vostra presenza.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Le Super App rappresentano un'innovazione significativa nel panorama tecnologico e digitale. Queste applicazioni multifunzionali si sono rapidamente diffuse, in particolare nei mercati orientali, acquisendo un ruolo di primaria importanza nella vita quotidiana degli utenti grazie alla loro capacità di integrare una vasta gamma di servizi in un'unica piattaforma.[1].

Le Super App costituiscono un modello architetturale interessante e complesso, in grado di integrare funzionalità eterogenee come comunicazione, commercio elettronico, pagamenti digitali e servizi logistici, tutto all'interno di un'unica applicazione.[2] Questo paradigma, pur avendo raggiunto un successo significativo in molte aree del mondo, presenta alcune difficoltà di adozione nei mercati occidentali, dove le dinamiche culturali, tecnologiche ed economiche risultano differenti.

In questo lavoro di tesi si cercherà di analizzare i fattori che hanno contribuito alla diffusione delle Super App nei mercati orientali, con un focus sull'architettura che le contraddistingue, evidenziando vantaggi e svantaggi che queste comportano. Sarà inoltre esaminato lo stato dell'arte attuale per comprendere le ragioni per cui la loro adozione in Occidente appare meno immediata e più complessa.

1.1 Perché sono emerse le Super app

Introdotte per la prima volta intorno al 2010, le Super App hanno progressivamente guadagnato popolarità, in particolare nei mercati orientali, registrando una crescita costante in termini di adozione da parte degli utenti.

Quali fattori hanno determinato la nascita e la diffusione delle Super App?

L'aumento del numero di applicazioni installate sui dispositivi mobili e utilizzate con diversa frequenza rende sempre più complessa la loro gestione da parte degli utenti.

Per rispondere a questa esigenza, si è diffuso progressivamente il concetto di un ecosistema digitale unificato, integrato all'interno di un'unica applicazione in grado di offrire un'esperienza più efficiente rispetto a quella garantita dall'uso di applicazioni separate.

In Italia, il concetto di Super App è ancora poco conosciuto, nonostante l'ampia diffusione dell'uso delle applicazioni mobile. Le Super App hanno avuto origine per la prima volta nel mercato digitale cinese, caratterizzato da un approccio mobile-first, ovvero privilegiando in prima istanza l'interfaccia della versione mobile, per arrivare solo in un secondo momento al suo layout "ampliato" e ottimizzato per schermo desktop.[3]

Le Super App si differenziano dalle applicazioni tradizionali poiché costituiscono un ecosistema digitale integrato. All'interno di queste applicazioni troviamo diverse mini-applicazioni (*Mini App*), ciascuna progettata per fornire un servizio specifico. Queste app *potenziate* consentono all'utente di accedere a numerosi e diversificati servizi con un solo download e un solo accesso, il tutto da un'unica piattaforma.

Figura 1.1: Cos'è una Super App?

L'evoluzione dalle singole app alle super-app è stata guidata da diversi fattori e ha seguito percorsi di crescita e posizioni di mercato diversi in Oriente e in Occidente.

In Oriente, le super-app hanno radici in mercati con redditi più bassi, dove le prime esperienze degli utenti con Internet sono avvenute tramite dispositivi mobili.[4] Le super-app come WeChat, Alipay e Kakao Talk hanno iniziato ad offrire una vasta gamma di servizi all'interno di un'unica piattaforma fin dall'inizio, rispondendo rapidamente alle esigenze quotidiane dei consumatori.

D'altra parte, negli Stati Uniti, le aziende tecnologiche hanno iniziato a offrire servizi limitati su PC prima dell'era degli smartphone. L'adozione delle applicazioni mobili ha portato alla frammentazione del mercato delle app, con ogni azienda che offre servizi specifici attraverso app separate. Questo approccio ha soddisfatto le esigenze dei consumatori statunitensi, ma ha impedito lo sviluppo di super-app consolidate.

Inoltre, le preoccupazioni riguardanti l'esperienza utente e i problemi tecnici hanno scoraggiato il consolidamento dei servizi in un'unica app. Anche le differenze culturali hanno influenzato il successo delle super-app.

In Asia, la maggiore fiducia nell'innovazione guidata dagli utenti e una cultura più collettivista hanno favorito l'adozione delle super-app. Inoltre, i sistemi di pagamento mobile hanno contribuito alla crescita delle super-app in Asia, dove le carte di credito non si sono mai diffuse tanto quanto in Occidente.

Infine, le differenze normative hanno giocato un ruolo chiave nell'adozione delle super-app. In Cina, alcune super-app hanno beneficiato di una limitata regolamentazione pubblica e di una maggiore protezione dallo sviluppo delle app straniere, che ha favorito la crescita delle super-app locali.

In conclusione, l'adozione delle super-app è stata guidata da una combinazione di fattori culturali, normativi, tecnologici ed economici, che hanno contribuito alla loro diffusione in Oriente e alla loro mancata adozione in Occidente fino ad ora. Tuttavia, ci sono segnali che suggeriscono un crescente interesse per le super-app anche negli Stati Uniti, con molte aziende che esplorano modelli simili per offrire una vasta gamma di servizi all'interno di una singola piattaforma.

Un esempio concreto di questo interesse crescente è rappresentato da Meta, che ha manifestato l'intenzione di ampliare le funzionalità di WhatsApp, estendendone il raggio d'azione fino all'e-commerce. Un altro caso significativo è quello di Elon Musk, che con l'acquisizione di Twitter, successivamente rinominata X, ha avviato un processo di trasformazione della piattaforma in un ecosistema più ampio, in grado di offrire una gamma diversificata di servizi, ispirandosi al modello delle Super App.[5]

1.2 Cosa sono le Super App

Ma cosa sono le Super App?

Un'interessante metafora, coniata dalla società di consulenza statunitense Gartner, descrive le Super App come un *Swiss army knife* (coltellino svizzero), evidenziandone la versatilità e la capacità di offrire una vasta gamma di servizi all'utente attraverso un'unica piattaforma. [6]

Dal punto di vista dei consumatori, le Super App rappresentano un'innovazione rilevante. L'accesso unificato a una pluralità di servizi all'interno di un'unica

applicazione costituisce un vantaggio evidente, semplificando l'esperienza utente e riducendo la necessità di interagire con più applicazioni distinte.

Grazie alle Super App, non è più necessario passare da un'app all'altra a seconda del prodotto o servizio da utilizzare. Basta avere un'unica icona sullo smartphone per eseguire tutte le operazioni necessarie. Inoltre, le Super App sono altamente personalizzabili e si adattano alle esigenze e alle preferenze individuali, offrendo servizi in linea con il nostro stile di vita personale.

Anche per le aziende, l'adozione del modello delle Super App comporta un cambiamento significativo. Esse infatti, per distinguersi e diventare punto di riferimento quasi esclusivo per gli utenti, sono tenute a creare ecosistemi aperti che contengono al loro interno una vasta gamma di servizi eterogenei concentrando la loro attenzione sulla customer experience.

Le aziende, grazie allo sviluppo di un ecosistema principale, possono ottimizzare i costi e i tempi di sviluppo per le mini-applicazioni collegate. Se per la loro attività può essere utile sviluppare più di un'app, considerare di realizzarne una "super" potrebbe essere la scelta più indicata. Ma i vantaggi non sono solo in termini di investimento, ma anche a livello di comunicazione. Con queste App, infatti, ogni azienda può ottenere una maggiore fidelizzazione dei propri utenti e disporre di una conoscenza ancora più approfondita del proprio target, traducendosi in migliori decisioni strategiche.

Dal punto di vista competitivo, il settore delle Super App è caratterizzato da un'elevata concorrenza, con numerose piattaforme che competono per attrarre e fidelizzare gli utenti. Ciò può portare a una lotta costante per l'attenzione degli utenti e a una pressione crescente per migliorare continuamente l'esperienza dell'utente e offrire nuove funzionalità.

1.3 Le sfide e le criticità delle Super App

Sebbene le Super App abbiano ottenuto un successo consolidato nei mercati asiatici, la loro diffusione su scala globale, e in particolare nei paesi occidentali, procede a un ritmo più lento.

Ciò è dovuto a motivi di svariata natura, non solo tecnologica ma soprattutto normativa e culturale ed in primis ad un diverso approccio nella cultura orientale ed occidentale a importanti aspetti inerenti le relazioni sociali in particolari campi come, ad esempio, la gestione della privacy e la sicurezza dei dati. Le Super App infatti, raccogliendo e quindi avendo a disposizione un'enorme quantità di dati, sensibili e non sensibili degli utenti, rischiano di confliggere con le normative in vigore, primo fra tutti il GPDR in Europa. La fiducia degli utenti nei confronti di un'unica piattaforma che gestisce molteplici aspetti della loro vita digitale

diventa quindi un elemento critico per il successo delle Super App nei mercati più regolamentati.

Un'altra difficoltà riguarda il rischio di monopolizzazione del mercato. Le Super App tendono a creare ecosistemi chiusi in cui gli utenti si affidano a un'unica piattaforma per numerosi servizi, riducendo la concorrenza e limitando l'innovazione. Questo fenomeno ha sollevato preoccupazioni in termini di regolamentazione antitrust, spingendo alcuni governi a valutare interventi per garantire un equilibrio tra innovazione e concorrenza leale.

Dal punto di vista tecnico, la creazione di un'architettura scalabile ed efficiente è una delle sfide più complesse. Le Super App devono integrare servizi eterogenei sviluppati da terze parti, garantendo al contempo un'esperienza utente fluida e prestazioni ottimali. La gestione delle Mini App, che spesso devono essere eseguite in un ambiente controllato all'interno dell'applicazione principale, richiede soluzioni avanzate per la comunicazione tra i moduli, la sicurezza delle API e l'allocazione delle risorse.

Infine, vi è la sfida della localizzazione. Le Super App devono adattarsi alle specificità di ogni mercato, affrontando differenze culturali, abitudini di consumo e normative differenti. Se nei mercati asiatici la centralizzazione dei servizi in un'unica piattaforma è stata accolta positivamente, nei paesi occidentali gli utenti tendono a preferire soluzioni più specializzate e frammentate, rendendo più complessa l'adozione di questo modello.

1.4 Obiettivi della tesi

I principali obiettivi di questo lavoro consistono nell'analizzare le caratteristiche tecnologiche delle Super App, con particolare attenzione alla loro architettura e al modello di sviluppo che ne ha favorito il successo in alcune aree del mondo, come accennato prima, e nella progettazione e sviluppo di una demo funzionale di Super App che integri una delle caratteristiche più rilevanti e innovative per questi sistemi: il supporto alle applicazioni web con approccio "Offline-First". Questo approccio consente di ottimizzare l'esperienza utente in contesti caratterizzati da connettività limitata o instabile, permettendo l'uso delle funzionalità principali anche in assenza di rete.

Il lavoro pratico della tesi sarà quindi volto non solo a comprendere gli aspetti teorici e architetturali delle Super App, ma anche a realizzare un prototipo che dimostri le potenzialità di un'applicazione Offline-First all'interno di questo ecosistema. Attraverso questa implementazione, si cercherà di fornire un esempio concreto delle opportunità offerte da tali tecnologie, esplorando sia i vantaggi che le possibili criticità.

1.5 Struttura della tesi

La tesi è organizzata in sette capitoli, ciascuno dedicato a un aspetto specifico dell'analisi e dello sviluppo della **Super App** con supporto **Offline-First**.

- **Capitolo 2 - Background**
  Fornisce una panoramica generale sul concetto di **Super App**, illustrandone l'origine, gli esempi più rilevanti e il modello di business. Inoltre, introduce il paradigma **Offline-First**, spiegandone vantaggi, sfide e componenti chiave.

- **Capitolo 3 - Tecnologie e scelte progettuali**
  Analizza le tecnologie utilizzate per la progettazione e lo sviluppo della Super App. Viene illustrata l'architettura dell'applicazione, approfondendo il ruolo delle **Mini App** e le soluzioni adottate per la loro integrazione tramite **WebView**. Si discute inoltre l'approccio Offline-First, valutando diverse tecniche di memorizzazione locale dei dati e gestione della persistenza. Infine, viene giustificata la scelta del framework di sviluppo, con un confronto tra **Flutter** e **React Native**.

- **Capitolo 4 - Il prototipo di Super App: struttura e comunicazione**
  Descrive il progetto della Super App sviluppata, illustrandone l'architettura, le funzionalità principali e le modalità di comunicazione con le Mini App. Viene approfondito l'uso delle WebView, il sistema di login automatico e il meccanismo di scambio dati basato su **JS-Channel**.

- **Capitolo 5 - Implementazione Offline-First**
  Analizza le sfide dell'approccio Offline-First e illustra come l'applicazione è stata progettata per supportare l'utilizzo senza connettività. Sono descritte le tecniche di gestione del catalogo prodotti, del carrello e delle operazioni offline con successiva sincronizzazione dei dati.

- **Capitolo 6 - Risultati e considerazioni finali**
  Presenta un'analisi dei risultati ottenuti durante l'implementazione della Super App, evidenziando vantaggi e limitazioni delle tecnologie adottate. Particolare attenzione è dedicata alle prestazioni dell'approccio Offline-First e ai limiti dell'utilizzo delle **Shared Preferences**.

- **Capitolo 7 - Conclusioni e lavori futuri**
  Riassume il lavoro svolto, evidenziandone i principali contributi. Vengono inoltre suggerite possibili evoluzioni della Super App, tra cui un'integrazione più strutturata delle Mini App e l'adozione di soluzioni più avanzate per la gestione dei dati in locale.

Capitolo 2

Background

2.1 Super App

2.1.1 La prima Super App: WeChat

Ovviamente quando si parla di Super App, il pensiero vola subito a WeChat, che è stata certamente la capostipite di tale genere di applicazioni.

WeChat è un'applicazione cinese multifunzionale sviluppata da Tencent, che offre servizi di messaggistica istantanea, social media e pagamenti mobile.

Concepita nel 2010 come una semplice app di messaggistica, grazie all'intuizione di Allen Zhang, responsabile di QQ, un servizio simile a Facebook del gruppo Tencent, essa fu lanciata per la prima volta nel 2011. [7]

Da allora, prendendo spunto dai social network occidentali, WeChat ha continuato a crescere e diversificare i propri servizi. Ha introdotto funzionalità come il QR Code, che consente di connettersi con amici e aziende tramite la scansione del codice, e i *Moments*, una sezione per la condivisione di post visibili ai propri contatti, simile a quanto avviene su Facebook. Tuttavia, la vera rivoluzione è avvenuta nel 2013 con l'introduzione di WeChat Pay, un portafoglio digitale che consente di pagare bollette domestiche o il conto in un ristorante direttamente dall'applicazione. [8]

L'aggiunta di "store virtuali" e "mini-programmi" (*Mini App*) ha consolidato l'orientamento commerciale di WeChat. I mini-programmi altro non sono piccole app accessibili direttamente da WeChat, che fungono da strumento per le aziende per fidelizzare gli utenti. Essi, per esempio sfruttando applicazioni votate soprattutto al gioco, hanno lo scopo di far virare l'utilizzo dell'utente verso applicazioni di altro genere, soprattutto di e-commerce. Tali app non hanno necessità di essere a loro volta installate in quanto già presenti nella Super App madre e questo ha certamente contribuito al loro successo, che, stando alle ultime stime è veramente vertiginoso: dai dati di gennaio del 2022 si desume che il numero di utenti che

li utilizza giornalmente è salito a 450 milioni e nel contempo il volume delle transazioni nel settore della ristorazione, del turismo e della vendita a dettaglio dovuto all'e-commerce su base annua si è praticamente raddoppiato.

Essa quindi, grazie anche alla dinamicità e all'occhio volto al mercato dei suoi sviluppatori, ha rapidamente guadagnato popolarità diventando la più grande app mobile autonoma al mondo nel 2018, con oltre 1,3 miliardi di utenti attivi mensili secondo dati di Statista. Attualmente, WeChat si posiziona al quinto posto tra i social network più utilizzati a livello mondiale, con un quarto degli utenti cinesi che trascorrono più di quattro ore al giorno sull'app.

Ci siamo soffermati su WeChat, in quanto il suo esempio è cruciale e sintetizza tutte le caratteristiche una Super App deve possedere, ovvero in sostanza la capacità di offrire una vasta gamma di servizi all'interno di un'unica piattaforma. Questi servizi includono funzionalità di messaggistica, commercio elettronico e molte altre, rendendo l'applicazione essenziale per molti utenti.

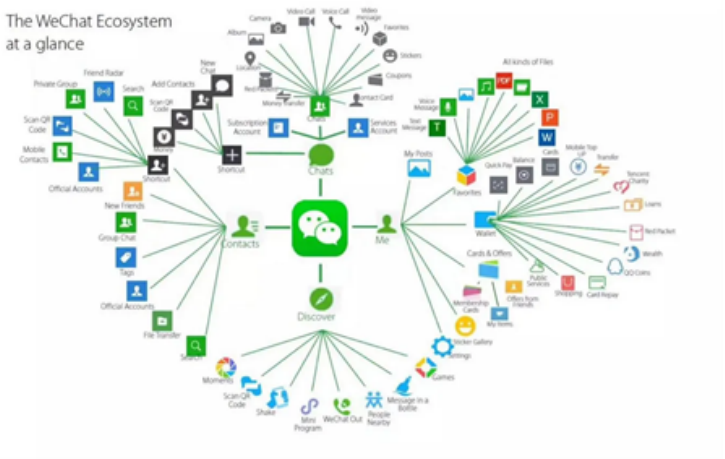

Figura 2.1: L'ecosistema di WeChat in sintesi

Il successo di WeChat in particolare risiede nella sua capacità di unire in un'unica app le funzionalità chiave dei principali social occidentali. Questo successo è stato in parte favorito dalla mancanza di concorrenza esterna, in quanto il governo cinese ha vietato l'accesso alle piattaforme straniere sul mercato domestico, consentendo a WeChat di crescere in una sorta di monopolio controllato. La storia di WeChat riflette la trasformazione della Cina da "fabbrica del mondo" a motore dell'innovazione globale. Tale successo ha spinto anche giganti occidentali, come Facebook, a tentare di replicare alcuni servizi offerti da WeChat. In tal modo, WeChat ha influenzato significativamente il modo in cui le app sono concepite.[9]

A differenza di molte start-up e giganti della tecnologia occidentali che dipendono pesantemente dalle entrate pubblicitarie, WeChat ha adottato un approccio diverso.

Piuttosto che sovraccaricare gli utenti con annunci pubblicitari, WeChat ha generato entrate integrando una vasta gamma di servizi, tra cui messaggistica, prenotazioni, pagamenti, e-commerce e vari servizi di utilità. Questo approccio ha contribuito a garantire una maggiore adozione dell'app e ha reso WeChat una parte essenziale della vita quotidiana per molti utenti cinesi, ampliando così il suo ruolo nel panorama tecnologico globale.

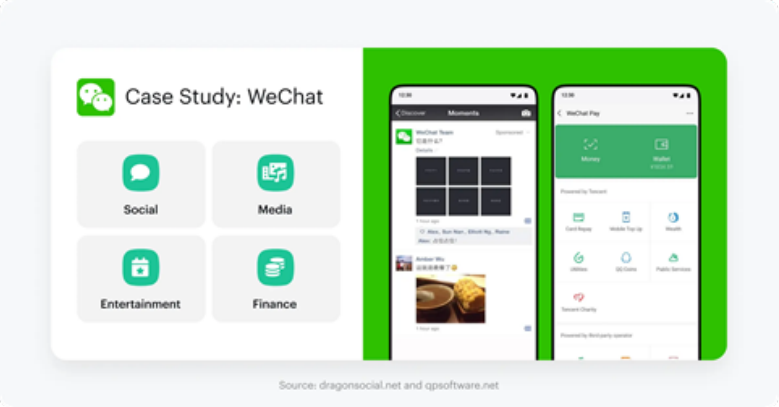

Figura 2.2: Caso studio: WeChat

2.1.2 Esempi di Super App

WeChat non è però l'unica Super App presente oggi sul mercato, per cui val la pena di fornire qualche altro esempio, senza la pretesa di essere né completi né esaustivi anche perché non è questo l'obiettivo cruciale della tesi.

Alipay

Alipay per esempio è un'applicazione fintech fondata nel 2004 da Jack Ma, fondatore di Alibaba, come parte del suo vasto impero di e-commerce. Inizialmente concepita come un'app di pagamento e portafoglio digitale, Alipay consente agli utenti di effettuare transazioni finanziarie in modo semplice e sicuro, eliminando la necessità di utilizzare denaro contante. Tuttavia, nel corso degli anni, Alipay ha ampliato la sua gamma di servizi per includere funzionalità aggiuntive come la prenotazione di taxi, l'ottenimento di carte di credito e l'acquisto di prodotti assicurativi. [10]

Dal 2008, Alipay ha esteso i suoi servizi per includere il pagamento di servizi pubblici, coprendo più di 300 città in Cina e collaborando con oltre 1.200 organizzazioni partner. Oltre ai classici servizi di pagamento delle utenze come acqua e luce, Alipay offre anche la possibilità di pagare multe per il trasporto, tasse di proprietà e tasse televisive via cavo.

Al di fuori della Cina, Alipay ha consolidato la sua presenza internazionale, con oltre 300.000 commercianti in tutto il mondo che accettano pagamenti diretti dai consumatori cinesi tramite l'app. Attualmente Alipay supporta transazioni in 18 diverse valute straniere, facilitando gli scambi finanziari transfrontalieri per gli utenti.

Figura 2.3: Alipay

Revolut

Con oltre 35 milioni di clienti globali, Revolut emerge come una delle principali Super App finanziarie a livello mondiale, offrendo soluzioni per migliorare le decisioni finanziarie degli utenti. Recentemente, l'azienda ha annunciato il lancio di una funzione di messaggistica istantanea, con l'obiettivo di rendere più agevole e meno imbarazzante parlare di questioni finanziarie, mantenendo tutti i dettagli relativi ai soldi in un'unica piattaforma integrata. [11]

Nel corso del 2021, Revolut ha introdotto Stays, una nuova funzionalità che semplifica la prenotazione di vacanze direttamente dall'app Revolut. Questo ulteriore sviluppo amplia l'offerta di servizi dell'app, consentendo agli utenti di gestire non solo le loro finanze, ma anche di pianificare e prenotare viaggi in modo rapido e conveniente, tutto da un'unica piattaforma centralizzata.

2.1.3 Il mercato delle Super App

Le Super App quindi, nate all'inizio come abbiamo visto quasi esclusivamente come app di messaggistica, si sono in realtà rapidamente evolute e adesso offrono servizi

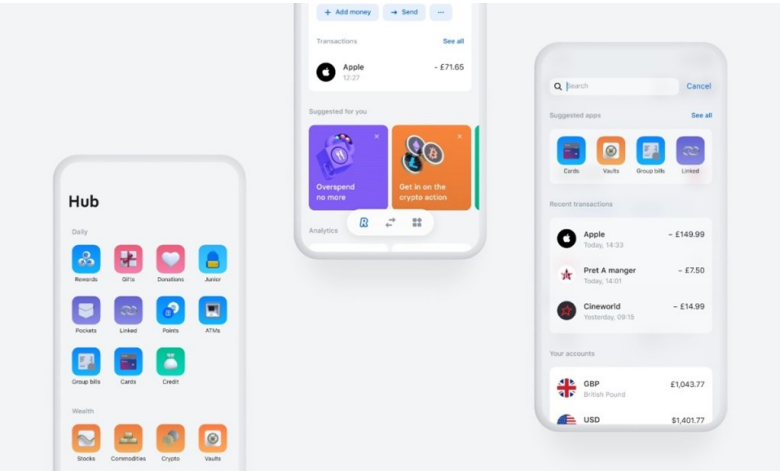

Figura 2.4: App Revolut

essenziali di e-commerce a milioni di utenti, riducendo al minimo l'uso del contante negli scambi commerciali.

Come ampiamente descritto in precedenza esse si sono prima diffuse soprattutto nei mercati emergenti, Cina, India, Brasile, dove si sono rivelate uno strumento ben più efficace dei tradizionali e a volte obsoleti servizi finanziari e in alcuni casi si stanno anche proponendo come importante strumento di gestione di risorse finanziarie.

Un'immediata e già percepibile conseguenza dell'uso delle Super App è come dicevamo la migrazione verso un'economia *cashless*, cui segue una contestuale demonetizzazione che ha fra gli altri insindacabili vantaggi anche quello di una chiara tracciabilità delle operazioni finanziarie, rendendo più trasparente l'economia e quindi di conseguenza più difficile l'evasione fiscale.

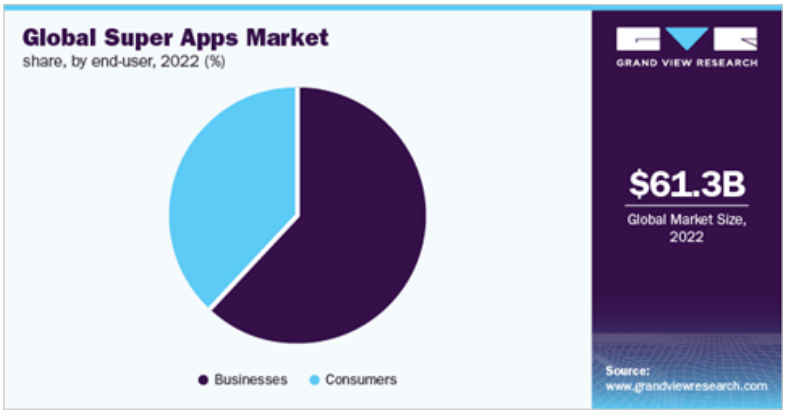

Figura 2.5: Mercato globale delle Super app

Secondo il report, nel 2022 il mercato globale delle Super app è stato valutato a 61,30 miliardi di dollari, con una prospettiva di crescita significativa. Si prevede infatti che questo settore crescerà a un tasso di crescita annuale composto (CAGR) del 27,8 % dal 2023 al 2030. Le Super App oggi rappresentano dunque un ecosistema completo che offre una vasta gamma di servizi, tra cui pagamenti digitali, social media, prenotazioni di taxi e biglietti per il cinema. Inoltre, consentono agli utenti di effettuare acquisti direttamente dall'applicazione stessa, fornendo una soluzione integrata per molteplici esigenze. [12]

La Cina si è affermata come leader nel mercato delle Super App, caratterizzato da un ampio numero di utenti e da una vasta gamma di servizi offerti. Inoltre, l'Asia Pacifica ha visto una rapida adozione delle Super App, trainata dalla presenza di una popolazione di 4,3 miliardi di persone, che rappresenta circa il 60% della popolazione mondiale. Questo evidenzia il potenziale di crescita significativo di questo settore nell'ambito regionale e globale. [13]

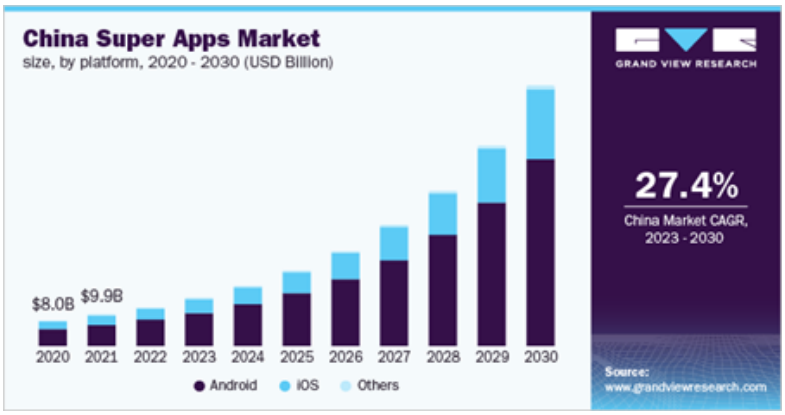

Figura 2.6: Mercato cinese delle Super App

L'impatto della pandemia da COVID-19 ha accelerato la digitalizzazione e l'adozione di nuove tecnologie in molti settori, compreso quello delle Super App. A causa del lavoro da casa e dell'istruzione online, c'è stato un aumento significativo del tempo trascorso sugli smartphone, portando a una maggiore dipendenza dalle applicazioni mobile per svolgere varie attività quotidiane. [13]

Inoltre, servizi offerti dalle Super App, come l'e-commerce e i servizi di consegna su richiesta, hanno registrato un aumento della domanda durante la pandemia. Le restrizioni legate al COVID-19 hanno spinto i consumatori a cercare soluzioni digitali per soddisfare le loro esigenze di acquisto e consegna, contribuendo così a una maggiore adozione e utilizzo delle Super App.

Ci sono alcuni trend particolari e casi specifici di Super App che evidenziano ulteriori sviluppi e sottolineano l'importanza di determinati settori o funzionalità.

Per esempio, alcune Super App come WeChat e Tata Neu integrano sia funzionalità di social media che di e-commerce. Questo trend riflette la crescente interconnessione tra social media e e-commerce, consentendo agli utenti di socializzare, comunicare e fare acquisti direttamente dall'app stessa. Questo modello permette agli utenti di navigare tra interazioni sociali e transazioni commerciali senza dover cambiare piattaforma, migliorando così l'esperienza utente e aumentando l'engagement.

Inoltre, molte Super App come Alipay, Paytm e Revolut offrono servizi finanziari avanzati come pagamenti digitali, trasferimenti di denaro, investimenti e assicurazioni. Questo evidenzia un trend significativo verso la digitalizzazione dei servizi finanziari, con le Super App che agiscono da piattaforme centrali per gestire le esigenze finanziarie degli utenti. L'adozione di questi servizi è stata accelerata dalla pandemia COVID-19 e dalla crescente preferenza per i pagamenti senza contanti e le transazioni digitali.

Infine, altre Super App come Grab e Gojek offrono una vasta gamma di servizi multisettoriali che vanno oltre i tradizionali settori come il trasporto e la consegna di cibo. Queste app fungono da hub per una varietà di servizi, inclusi pagamenti, prenotazioni, assicurazioni e altro ancora. Questo modello di business multisettoriale evidenzia la tendenza verso l'integrazione di molteplici servizi all'interno di un'unica piattaforma, offrendo agli utenti maggiore convenienza e accesso a un'ampia gamma di servizi.

2.1.4 Il modello di business delle Super App

Fondamentale per l'approccio diversificato delle Super App è, come accennato in precedenza, il concetto di mini-programmi, che consente alle aziende di creare le proprie applicazioni all'interno dell'ecosistema della Super App madre. A differenza delle controparti occidentali che spesso impongono commissioni elevate (30%) e utilizzano sistemi di classificazione delle app, le Super App sviluppate in oriente (al

solito WeChat in primis) hanno adottato un modello più favorevole agli sviluppatori. Questo approccio ha permesso anche ai più piccoli sviluppatori di competere ad armi pari con le grandi aziende.

Il modello di business di tali Super App è progettato per rispettare le esperienze degli utenti e trattarli come amici. Questa filosofia è evidente come accennato in precedenza, nella loro politica pubblicitaria. A differenza di molte altre piattaforme social, tali Super App limitano il numero di annunci pubblicitari che un utente vedrà nella propria bacheca a soli circa 2 annunci al giorno. Questo approccio è in linea con la loro visione di non intromettersi troppo nell'esperienza dell'utente o di sovraccaricarlo di pubblicità.

L'obiettivo qui non è solo quello di creare una piattaforma adatta agli annunci, ma di offrire un ecosistema in cui i marchi possano interagire con gli utenti in modo più coinvolgente e personalizzato. Invece di trattare gli utenti come semplici consumatori di annunci, esse incoraggiano i marchi a creare servizi a valore aggiunto attraverso account ufficiali e mini-programmi. Queste piattaforme consentono quindi ai brand di offrire servizi direttamente agli utenti, promuovendo una forma di pubblicità più interattiva e partecipativa.

Figura 2.7: Esempi di Mini App di WeChat

Quindi esse utilizzano una combinazione di tecnologie per sviluppare le proprie piattaforme di Super App. In particolare la piattaforma di base di WeChat è sviluppata utilizzando tecnologie native sia per Android che per iOS. Questa piattaforma funge da "ospite" per i vari servizi, chiamati mini-app, offerti all'interno di essa.

2.2 Offline-First

Nella società odierna, l'utilizzo diffuso dei dispositivi mobile per accedere a Internet rappresenta un fenomeno sempre più rilevante.

A differenza dei computer desktop, generalmente fissi e con accesso fisso a Internet, i dispositivi mobile sono caratterizzati dalla loro portabilità e possono essere utilizzati ovunque dai loro proprietari. Tuttavia, questa mobilità comporta sfide significative in termini di connettività, poiché la copertura della banda larga mobile può variare notevolmente in base alla posizione geografica. Anche in contesti altamente urbanizzati e sviluppati, come le grandi città, non è garantita una connettività costante. Le zone senza copertura possono essere presenti anche in contesti apparentemente ben collegati. Ad esempio, potrebbe capitare di trovarsi in viaggio su un autobus per recarsi al lavoro, in un angolo isolato di una stanza o in un ascensore diretto all'ultimo piano di un edificio. Inoltre, aree affollate come festival, eventi sportivi e conferenze possono rappresentare un ulteriore ostacolo, dato che migliaia di persone condividono la stessa connessione WiFi o dati mobili.

Di conseguenza, la progettazione di un'architettura per le app mobile in grado di funzionare offline assume un'importanza cruciale, poiché assicura agli utenti l'accesso alle funzionalità essenziali anche in assenza di connessione internet.

2.2.1 Vantaggi, svantaggi e casi studio di apps Offline-First

La funzionalità offline in un'app mobile può portare numerosi vantaggi significativi; vediamo di elencarli:

- **Miglioramento dell'esperienza utente:** Le app offline consentono agli utenti di continuare a utilizzare l'app anche in zone con connettività Internet limitata o assente, garantendo un'esperienza utente più fluida e coerente.
- **Tempi di caricamento più rapidi:** Senza la necessità di recuperare dati attraverso la rete, le app offline possono caricarsi più velocemente, migliorando l'esperienza utente complessiva.
- **Minore utilizzo dei dati:** Le apps Offline-First richiedono un minor utilizzo di dati e questo risulta particolarmente vantaggioso sia per quegli utenti che utilizzano ancora abbonamenti con piani dati limitati, sia quando si ha a che fare con applicazioni che necessitano di una grande quantità di dati.
- **Maggiore privacy e sicurezza:** Considerato che i dati utilizzati vengono archiviati localmente sul proprio dispositivo, le apps Offline-First garantiscono maggiore sicurezza e privacy all'utente che non ha necessità di esportare in rete, o comunque lo ha in maniera molto minore, dati e informazioni personali.

- **Aumento dell'engagement:** Considerato che tali apps sono pensate anche per il loro utilizzo offline, esse riescono nello scopo di fidelizzare maggiormente il cliente.

Abbiamo visto i vantaggi della funzionalità offline delle app mobile; non nascondiamoci però che esse possono celare anche potenziali svantaggi. Analizziamoli:

- **Funzionalità limitate:** Per quanto la tecnologia è in rapida evoluzione, è del tutto ovvio che non è possibile che tali app possano offrire in assenza di rete, tutte quelle funzionalità di cui sono dotate quando invece sono online, in particolar modo quelle che lavorano in real time o che elaborano ingenti moli di dati su server.
- **Requisiti di archiviazione:**Lavorando offline, tali app necessitano di utilizzare un certo spazio di archiviazione sulla cache del dispositivo locale e questo potrebbe creare dei problemi nel normale utilizzo di tali dispositivi.
- **Sincronizzazione e aggiornamento dei dati:** Mantenere i dati sincronizzati tra l'app e il server può essere complesso. Se i dati vengono aggiornati sul server mentre l'app è offline, potrebbero diventare obsoleti o imprecisi, richiedendo meccanismi efficaci di sincronizzazione.
- **Rischi per la sicurezza:** Nonostante abbiamo annoverato la sicurezza tra i possibili vantaggi dell'offline, non si deve sottovalutare il risvolto della medaglia: le app offline potrebbero essere più vulnerabili ai rischi per la sicurezza, come il furto di dati o l'accesso non autorizzato, poiché i dati sono archiviati localmente sul dispositivo.
- **Manutenzione e supporto:** Lo sviluppo e la manutenzione delle funzionalità offline possono essere complessi, richiedendo la gestione di situazioni in cui l'utente è offline e il fornire meccanismi di gestione degli errori e risoluzione dei problemi adeguati.

Alcuni dei migliori casi d'uso in cui l'uso offline può essere molto utile per gli utenti possono essere ad esempio:

- **App di viaggio e turismo:** Durante i viaggi, soprattutto in aree remote o rurali, la connettività di rete può essere poco affidabile o addirittura assente. In tali contesti tali app possono risultare molto vantaggiose perché possono consentire agli utenti di accedere offline a tutte le informazioni necessarie (prenotazioni, particolari itinerari, ristorazione, attrazioni locali, eventi caratteristici) anche senza connessione internet, e garantendosi quindi anche in questo disagiato ma non improbabile contesto, un'esperienza di viaggio completa e rilassante.

- **App per la sanità:** Tali app che sono sempre maggiormente utilizzate, rivestono un ruolo cruciale nell'accesso ad informazioni personali e a registri medici, e quindi garantirsi l'accesso a tali informazioni anche in assenza di connessione e quindi in condizioni di emergenza, potrebbe essere di fondamentale importanza, in quanto garantirebbe agli operatori sanitari la continuità nelle cure del paziente e la possibilità di assumere decisioni informate anche in assenza di connessione internet.

- **App educative:** Le app nel campo dell'istruzione possono offrire un accesso offline a materiali didattici, libri di testo e altre risorse educative. Questa caratteristica è particolarmente vantaggiosa per gli studenti che potrebbero trovarsi in contesti in cui l'accesso a internet non è sempre disponibile o affidabile, come ad esempio nelle aree rurali o in paesi in via di sviluppo.

- **App per servizi sul campo e vendite:** I dipendenti che operano sul campo, come tecnici o venditori, potrebbero trovarsi in situazioni in cui la connettività di rete è limitata o assente. In questi casi, un'applicazione dedicata può consentire loro di accedere offline a informazioni critiche sui clienti, cataloghi di prodotti e altri dati essenziali, garantendo la continuità operativa e migliorando la qualità del servizio offerto ai clienti. [14]

2.2.2 Approccio Offline-First

Il concetto di "Offline-First" si riferisce dunque a un approccio nello sviluppo di app mobile che mette al centro la capacità dell'applicazione di funzionare senza la necessità di una connessione a internet. Questo principio è cruciale per garantire un'esperienza utente ottimale, specialmente in contesti in cui la connettività può essere limitata o assente. [15]

Come detto, gli sviluppatori web spesso operano partendo dall'assunto che gli utenti abbiano accesso costante e veloce a internet, poiché sviluppano le loro app utilizzando dispositivi desktop moderni e una connessione broadband stabile. Tuttavia, questa prospettiva non tiene conto delle varie situazioni in cui gli utenti potrebbero trovarsi, come ad esempio luoghi con connessione limitata o inesistente.

L'approccio Offline-First mira a correggere questa falla concentrandosi innanzitutto sull'esperienza offline dell'applicazione, garantendo che la maggior parte, se non tutte, delle sue funzionalità principali siano accessibili anche in assenza di connessione a internet. Questo approccio non dovrebbe essere considerato un'eccezione per i casi d'uso in luoghi remoti, ma piuttosto come uno standard fondamentale per lo sviluppo di qualsiasi app mobile.

Le app mobile sono costruite con dipendenze internet principalmente per due ragioni:

Figura 2.8: Apps mobile Offline-First

1. **Presunzione di disponibilità e velocità della connessione:** Spesso si presume che la connettività internet sia costantemente disponibile e veloce, e questa assunzione è supportata dalla diffusione di risorse basate su internet come le api restful e i backend cloud. Di conseguenza, la costruzione di app dipendenti dall'internet diventa una scelta di sviluppo conveniente e generalizzata.

2. **Mancanza di considerazione per i requisiti "Offline-First":** Vi è una tendenza a non considerare i requisiti "Offline-First" come parte integrante dello sviluppo delle app, specialmente in contesti metropolitani. Questo comporta l'errata convinzione che tali requisiti si applichino solo a scenari eccezionali e remoti, come l'esplorazione energetica, l'estrazione mineraria di roccia dura o le operazioni di salvataggio in wilderness, dove l'app deve funzionare senza accesso a internet per periodi prolungati.

Queste assunzioni hanno portato allo sviluppo di numerose app mobile che dipendono esclusivamente dalla connettività internet, causando problemi significativi agli utenti quando si trovano in situazioni in cui la rete non è disponibile.

È cruciale adottare un cambiamento di mentalità nello sviluppo, in modo che non si consideri la connettività internet come un requisito fondamentale e le interruzioni di rete come un errore, ma piuttosto ci si aspetti l'inaffidabilità intrinseca di internet e si progettino le app attorno a essa. In altre parole, per garantire app mobile più veloci e affidabili, è necessario prioritizzare la creazione della migliore esperienza offline possibile.

La progettazione di un'architettura per applicazioni mobile che operano offline richiede una pianificazione attenta e la considerazione di diversi fattori al fine di garantire un'esperienza utente fluida e affidabile. Il primo passo consiste nell'identificare le funzionalità essenziali di cui gli utenti avranno bisogno quando saranno offline.

2.2.3 Componenti critici di un app Offline-First

Una volta identificate le funzionalità essenziali, si può procedere con la progettazione dell'architettura che supporta l'accesso offline a tali funzionalità. La caratteristica più importante dello sviluppo di un'app mobile Offline-First è il modo in cui esse gestiscono i dati. Per rimuovere le dipendenze da internet, è necessario archiviare ed elaborare i dati localmente nell'app, assicurandosi che possano essere condivisi con altri utenti e che eventuali modifiche ai dati si riflettano in modo istantaneo e accurato in tutto l'ecosistema dell'app.

Pertanto, gli aspetti principali che emergono includono:

- **Elaborazione locale dei dati:** Garantire la persistenza dei dati sul dispositivo mobile per mantenere l'app disponibile in caso di assenza di rete, offrendo al contempo reattività in tempo reale.
- **Elaborazione dei dati nel cloud:** Nonostante il modello Offline-First, il cloud continua a svolgere un ruolo fondamentale come punto di aggregazione centrale per i dati dell'app.
- **Sincronizzazione dei dati:** Necessaria per garantire la coerenza e l'integrità dei dati tra i client dell'app e il cloud, assicurando che tutti i dispositivi siano aggiornati in modo corretto.
- **Gestione dei conflitti ed errori:** Importante considerare che l'applicazione funzionerà su più dispositivi con un server centralizzato, quindi è cruciale gestire i conflitti e gli errori durante la sincronizzazione dei dati tra dispositivi e il server.
- **Sicurezza:** Implementare misure di sicurezza efficaci per proteggere i dati dell'utente archiviati localmente sul dispositivo, riducendo così il rischio di furto o perdita di dati. [16]

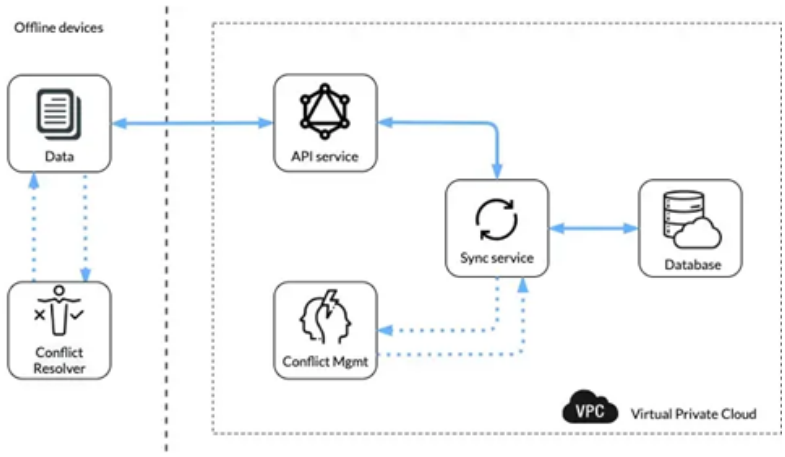

Figura 2.9: Componenti di una app Offline-First

Esaminiamo nel dettaglio gli aspetti individuati e vediamo quali potrebbero essere le migliori scelte implementative per ottimizzare l'Offline-First.

Elaborazione locale dei dati

Per garantire la disponibilità dell'app anche in assenza di connettività di rete, è fondamentale archiviare i dati localmente.

Alcuni fornitori di database cloud per soluzioni mobile offrono la memorizzazione temporanea dei dati nella cache sul dispositivo, dove le scritture vengono raccolte e mantenute in coda fino al ripristino della connettività, quando possono essere sincronizzate con il database cloud di backend. Tuttavia, questo approccio può generare problemi se l'app rimane offline per un periodo prolungato, poiché la crescita della coda può influenzare negativamente le prestazioni dell'app. Inoltre, la memorizzazione nella cache temporanea impone limiti alla quantità di dati che possono essere raccolti, con il rischio di perdita di dati se il limite viene superato.

Diversamente da una cache temporanea, un database incorporato è più resiliente, sicuro ed efficiente nell'elaborazione, in quanto è progettato per l'archiviazione rapida e a lungo termine dei dati. Pertanto, un database incorporato dovrebbe essere la scelta preferita per le prime app mobile offline. Le caratteristiche desiderabili della tecnologia di database incorporato includono:

- Supporto per il backup dei dati sul dispositivo per fornire un failover e ridurre al minimo il rischio di perdita di dati.
- La capacità di distribuire database predefiniti per un avvio più rapido delle app.
- Notifiche asincrone di modifiche al database per abilitare flussi di lavoro reattivi.
- Un SDK intuitivo e facile da usare compatibile con i linguaggi e le piattaforme di programmazione preferiti.
- Supporto completo per SQL.
- Funzionalità di ricerca integrate per facilitare l'accesso e la gestione dei dati archiviati. [17]

Elaborazione dei dati nel cloud

In un contesto offline, un database cloud continua a fungere da backend primario per le applicazioni mobile. Per garantire la massima flessibilità e scalabilità, i database di documenti JSON NoSQL rappresentano la scelta ideale. Questi database sono progettati per una distribuzione su larga scala e offrono elevate prestazioni, disponibilità e tolleranza agli errori. Inoltre, il modello di archiviazione dei documenti JSON è meno rigido rispetto ai database relazionali, consentendo modifiche più rapide e agevolando lo sviluppo delle applicazioni.

Le caratteristiche desiderabili della tecnologia di database cloud includono:

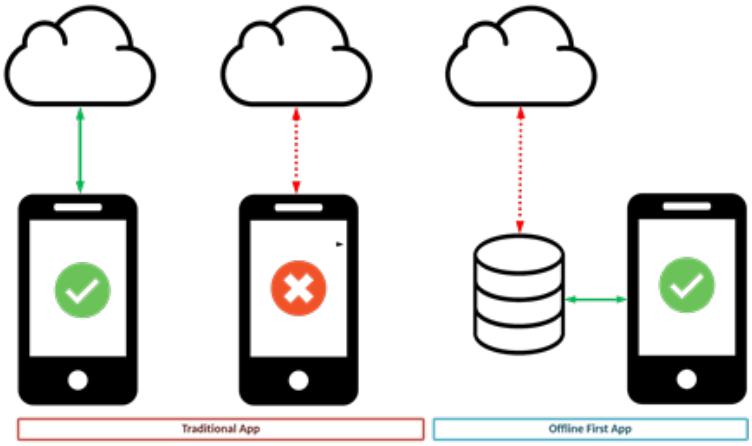

Figura 2.10: Elaborazione locale dei dati

- Supporto per funzionalità come valore-chiave, indicizzazione, ricerca full-text, analisi, serie temporali ed eventi integrati, per una gestione avanzata dei dati.
- Supporto completo per SQL, garantendo la compatibilità con le query e le operazioni di interrogazione standard.
- Funzionalità di sharding automatico, failover automatico e replica automatica, per garantire una distribuzione ottimale dei dati e la continuità operativa.
- Opzioni di hosting flessibili, con possibilità di scelta tra soluzioni ospitate o autogestite in base alle esigenze specifiche dell'applicazione e dell'organizzazione.

Sincronizzazione dei dati

È del tutto evidente che per permettere un funzionamento corretto delle applicazioni Offline-First, si deve gestire con la massima attenzione l'implementazione del sistema che garantisce la sincronizzazione dei dati tra gli utenti e il backend di tali applicazioni. Esistono diverse strade che possono essere seguite per l'implementazione di tale servizio, e ogni approccio presenta ovviamente caratteristiche proprie, e, unite ad esse, vantaggi e svantaggi:

- **Sincronizzazione manuale:** Usando tale procedura gli utenti decidono in maniera autonoma quando procedere alla sincronizzazione dei propri dati con il server esterno. Questo approccio è utile quando la disponibilità della rete non è nota o quando gli utenti desiderano controllare personalmente il momento della sincronizzazione. Tuttavia, richiede l'intervento manuale degli utenti e potrebbe portare a conflitti di sincronizzazione che devono essere risolti manualmente.
- **Sincronizzazione programmata nell'app:** In questo caso, la sincronizzazione viene programmata all'interno dell'applicazione per avviarsi in determinati orari prestabiliti. È necessario considerare diversi fattori, come lo stato del dispositivo, la disponibilità della rete e il carico del server, per garantire una sincronizzazione efficace.
- **Sincronizzazione programmata nel server:** Qui, il server invia una notifica push agli utenti per informarli della disponibilità di nuovi dati. Gli utenti possono quindi avviare manualmente il processo di aggiornamento del database locale tramite un'apposita opzione nell'applicazione.
- **Precaricamento:** Questo approccio si basa sulle previsioni di ciò che l'utente potrebbe richiedere e pre-carica i dati nella cache quando è disponibile una connessione internet ad alta velocità. In questo modo, l'applicazione è in

grado di funzionare in modalità offline, con la sincronizzazione che avviene in background in base alla disponibilità della connessione.

- **Sincronizzazione dei database:** Qui, la comunicazione tra l'app Offline-First e il server backend avviene tramite la sincronizzazione dei database. Entrambi monitorano le modifiche e si sincronizzano tra loro, garantendo che i dati su entrambi i lati siano aggiornati. Anche in assenza di connessione internet, gli utenti possono apportare modifiche che verranno sincronizzate non appena la connessione sarà ripristinata. [18]

La scelta del ciclo di sincronizzazione dipende dalla disponibilità della rete, dall'importanza dei dati e dai requisiti aziendali specifici, al fine di garantire una corretta gestione e condivisione dei dati nelle applicazioni Offline-First.

Gestione conflitti ed errori

Attualmente, è abbastanza diffuso che gli utenti utilizzino più dispositivi, rendendo la gestione delle applicazioni offline un compito più complesso e coinvolgente. Una situazione comune che può verificarsi è quando un utente sincronizza i propri dati da due dispositivi diversi sul server, potenzialmente causando conflitti di sincronizzazione. Un altro scenario che presenta conflitti è quando più utenti, ciascuno con più dispositivi, accedono contemporaneamente agli stessi servizi offerti dall'applicazione. In tali circostanze, se uno o entrambi i dispositivi perdono la connettività, continuano comunque a memorizzare localmente i dati, ma la questione critica riguarda la sincronizzazione di tali dati con il server una volta che la connessione viene ripristinata.

Infatti in caso di scarsa o totale mancanza di connettività, i dispositivi tengono traccia delle modifiche fatte ai dati localmente, nel proprio database. Quando però la connettività è ripristinata occorre far sì che tali dati siano allineati con quelli contenuti nel server e la gestione di tale procedura non è semplicissima e bisogna quindi assicurarsi che i dati siano aggiornati in maniera accurata e che vengano efficacemente risolti eventuali conflitti. In particolare nel caso non improbabile in cui lo stesso utente accede agli stessi servizi con dispositivi diversi, o anche quando più utenti utilizzano gli stessi servizi, bisogna coordinare con la massima attenzione il processo di sincronizzazione dei dati per evitare ridondanze indesiderate o conflitti all'interno dei dati stessi.

Ecco quindi sorgere come nuovo paradigma nelle scelte progettuali ed architettoniche che si andranno a prendere, la necessità di mantenere la correttezza dei dati. Il processo di gestione dei dati avviene attraverso due distinti fasi: l'identificazione dei conflitti e la risoluzione dei conflitti.

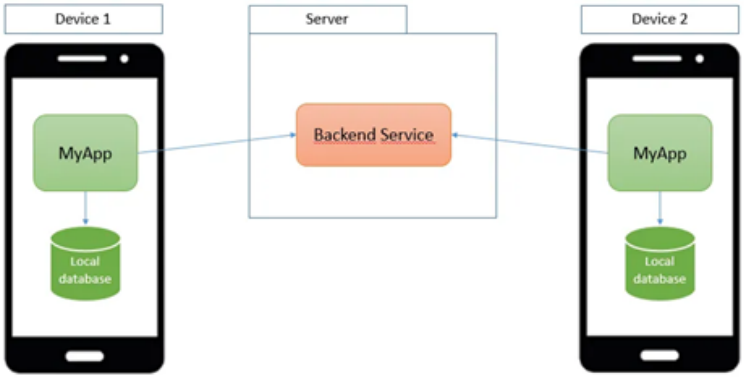

Figura 2.11: Gestione conflitti ed errori 1

- Gli identificatori di conflitto svolgono il compito di verificare la presenza di controversie tra i dati in arrivo e quelli già presenti sul cloud. Una volta individuato un conflitto, si passa alla fase successiva di risoluzione.
- La risoluzione dei conflitti può essere ottenuta attraverso diversi metodi, tra cui il "client win", il "server win", i "risolutori basati sul tempo" e i "sistemi di controllo delle versioni".
  - Nel metodo "client win", i dati presenti sul client sovrascrivono quelli già esistenti sul cloud.
  - Nel metodo "server win", invece, sono i dati sul server a sovrascrivere quelli in conflitto sul client.
  - Il "risolutore basato sul tempo" determina che i dati più recenti vengano conservati nell'applicazione, indipendentemente dai dati presenti sul client o sul server. Questo richiede che il sistema tenga traccia di tutti i timestamp per gli eventi di modifica mentre è offline.
  - Infine, il "risolutore controllato dalla versione" opera in modo simile alla strategia utilizzata in Git: in caso di conflitto, è preferibile lasciare che l'utente scelga quale versione specifica dei dati mantenere nel sistema. [19]

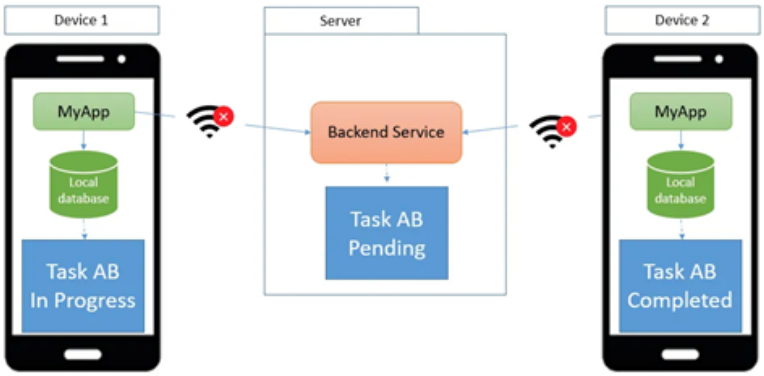

Figura 2.12: Gestione conflitti ed errori 2

Capitolo 3

Tecnologie e Scelte Progettuali

Dopo aver esplorato l'evoluzione storica delle Super App e le loro prospettive di sviluppo futuro, ci addentriamo ora in un'analisi tecnica approfondita. In questo capitolo esamineremo l'architettura attuale di queste applicazioni, analizzando le soluzioni disponibili sul mercato e le scelte adottate per la realizzazione del nostro progetto.

Partiremo dallo studio delle tecnologie chiave per la costruzione di una Super App, per poi concentrarci sulle strategie adottate per implementare funzionalità Offline-First. Successivamente, confronteremo i principali framework di sviluppo cross-platform, con particolare attenzione al confronto tra Flutter e React Native. Concluderemo illustrando le scelte tecnologiche finali adottate nel nostro progetto, giustificandole alla luce delle esigenze specifiche e delle analisi condotte.

3.1 Architetture delle Super App

In questi anni il processo di progettazione e realizzazione delle Super App si è mosso seguendo due diverse direttici: soluzioni basate sul web e soluzioni native. La scelta tra queste due alternative dipende da diversi fattori, tra cui le esigenze degli utenti, le caratteristiche del mercato di riferimento e gli obiettivi aziendali. [20]

Da un lato, le Super App basate sul web offrono una maggiore accessibilità e riducono le barriere all'adozione, consentendo agli utenti di usufruire di molteplici servizi direttamente da un'unica interfaccia web, senza la necessità di scaricare e installare applicazioni aggiuntive. Questo approccio risulta particolarmente vantaggioso nei contesti in cui l'accesso a Internet è costante e affidabile, e per aziende che desiderano ridurre i costi di sviluppo e manutenzione su diverse piattaforme. [13]

Dall'altro lato, le Super App native forniscono un'esperienza utente più fluida e integrata, sfruttando appieno le capacità dei dispositivi mobili. Grazie alla possibilità di accedere a funzionalità avanzate come il supporto offline, le notifiche push e l'integrazione con l'hardware del dispositivo, le Super App native rappresentano una scelta ideale per mercati in cui il download e l'uso intensivo delle applicazioni sono pratiche consolidate. [21]

La scelta dunque tra lo sviluppo di una Super App web piuttosto che nativa dipende dunque non solo dalle specifiche intrinseche dell'applicazione, ma anche dal pubblico target di riferimento. [22]

Se si sceglie la soluzione web si avrà di certo un'app più flessibile e quindi con più possibilità di facile distribuzione, ma magari non del tutto performante. Le applicazioni native di contro offrono di certo superiori prestazioni, e una maggiore possibilità di personalizzazione, ma proprio per tali caratteristiche sono probabilmente più rivolte a un utente esperto. Essa è progettata per garantire efficienza, scalabilità e anche facilità di manutenzione, e per far ciò si basa su una struttura modulare e integrata.

Tale modello si articola sostanzialmente in tre componenti fondamentali:

1. **Sviluppo della Super App:**La Super App è pensata e realizzata come una piattaforma centrale che ospita al suo interno diverse Mini App. La sua funzione principale in tal caso è proprio consentire il dialogo e l'interazione tra le applicazioni in essa contenute, facendo sì che gli utenti godano di una navigazione fluida e coerente all'interno dei vari servizi gestiti.

2. **Sviluppo delle Mini App:** Ciascuna Mini App è progettata come un modulo interconnesso con gli altri, ma assolutamente indipendente e che quindi offre servizi specifici e che ha la possibilità di essere aggiornata, modificata e manutenuta sena interferire con il buon funzionamento di tutta la piattaforma.

3. **Centro delle Mini App:** Questo sistema centrale è responsabile della gestione e distribuzione delle Mini App all'interno della Super App, assicurando che possano essere scalate facilmente e integrate senza soluzione di continuità. Inoltre, facilita l'espansione dell'ecosistema digitale, rendendo la piattaforma più versatile e facilmente adattabile alle esigenze degli utenti.

Per garantire che la Super App sia scalabile e facilmente manutenibile nel tempo, vengono adottate architetture avanzate, ognuna delle quali risponde a specifiche necessità:

- **Architettura Monolitica e MVC:** Sebbene sia una soluzione tradizionale, l'architettura monolitica è meno flessibile per ecosistemi complessi e in continua espansione. Tuttavia, è utile in contesti più semplici, dove la gestione centralizzata è vantaggiosa.

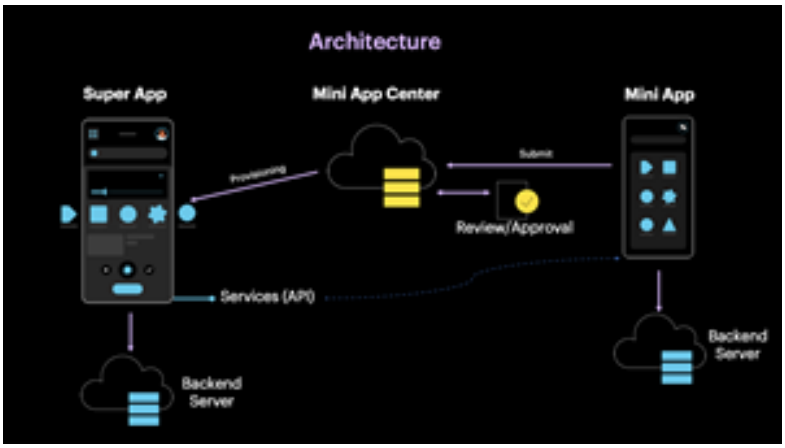

Figura 3.1: Architettura della Super App

- **Architettura a Microservizi:** Questa architettura separa le diverse funzioni della Super App in servizi indipendenti, migliorando la modularità e la scalabilità. Consente di aggiornare o modificare una singola parte senza compromettere l'intero sistema, facilitando così l'espansione delle funzionalità.

- **Clean Architecture:** Un approccio che favorisce la separazione delle responsabilità, garantendo una struttura del codice pulita e facilmente manutenibile. Questo tipo di architettura permette di espandere la Super App in modo modulare, mantenendo il codice organizzato e facilmente gestibile nel lungo periodo. [23]

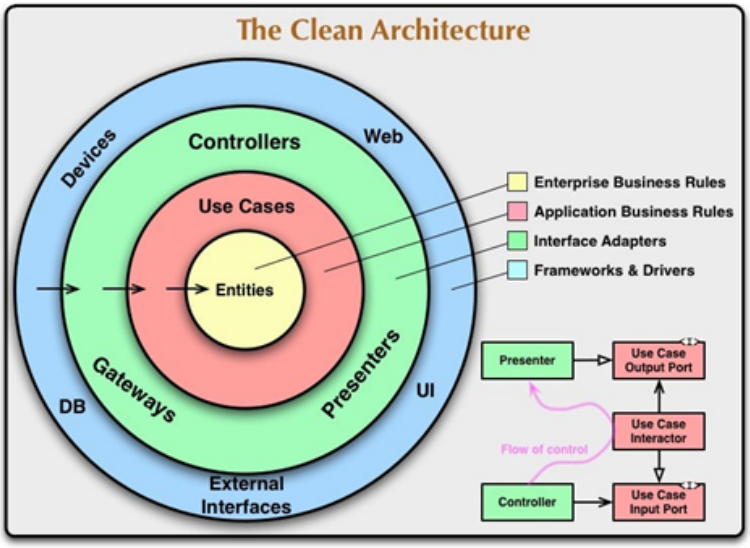

Figura 3.2: Clean Architecture

3.1.1 Integrazione delle Mini App

L'integrazione delle Mini App all'interno di una Super App è come si è cercato di far rilevare un processo fondamentale affinché si realizzi un ecosistema che sia nel contempo coerente e scalabile. Tali piccole applicazioni modulari offrono infatti specifici servizi all'interno della piattaforma ospite e la loro integrazione permette al sistema, nel suo complesso, di rimanere gestibile, estensibile e dunque performante. [24]

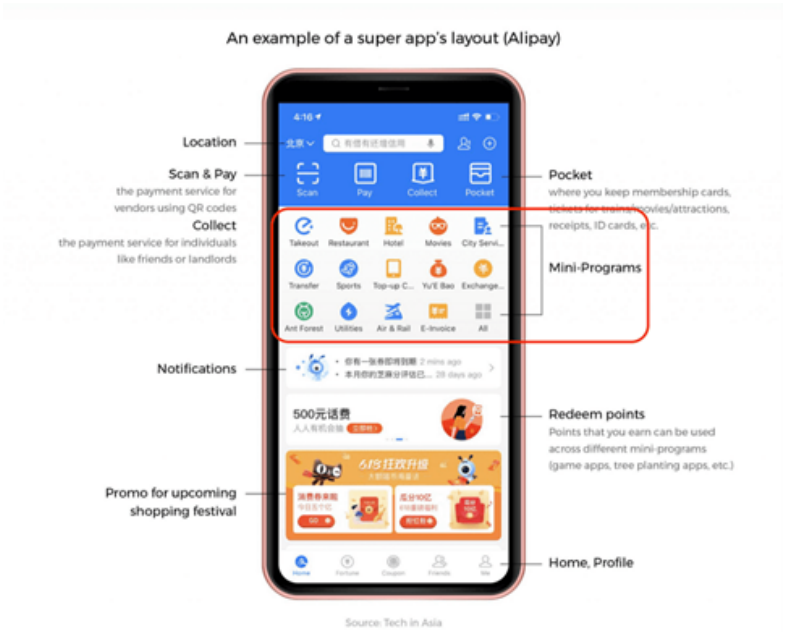

Figura 3.3: Esempio di un layout di una Super App

Come si integrano le Mini App nelle Super App?

L'integrazione di Mini App in una Super App può essere descritta attraverso un processo generale che comprende la definizione di standard e la gestione dei framework per ogni Mini App. Ecco i passaggi chiave:

1. **Creazione del Framework della Mini App:** Ogni Mini App viene sviluppata come un modulo indipendente che fornisce una funzionalità specifica. Il team di sviluppo della Mini App (MAT - Mini App Team) ha il compito di realizzare il codice sorgente e di impacchettarlo in un framework che a sua volta è progettato per poter essere facilmente integrato nella Super App madre, garantendo però al contempo l'autonomia a la funzionalità della Mini App all'interno dell'ecosistema esteso.

2. **Definizione di un'Interfaccia Standard:** Per evitare la complessità e le difficoltà legate alla gestione di numerose Mini App, la Super App definisce un'interfaccia standard che tutte le Mini App devono seguire. Questa interfaccia standard serve come un contratto che ogni Mini App deve implementare. La definizione di un'interfaccia comune permette di mantenere la coerenza nell'integrazione di diverse Mini App, semplificando la gestione delle comunicazioni tra la Super App e i vari moduli.

3. **Implementazione dell'Interfaccia da parte delle Mini App:** Ogni team di sviluppo delle Mini App (MAT) implementa l'interfaccia standard nel proprio framework. L'interfaccia stabilisce quali funzionalità devono essere implementate e come i dati devono essere scambiati tra la Mini App e la Super App. Questo approccio garantisce che ogni Mini App possa interagire con la Super App senza conflitti, indipendentemente dalla tecnologia o dalla logica interna della Mini App.

4. **Integrazione del Framework nella Super App:** Una volta che il framework della Mini App è pronto e ha implementato l'interfaccia standard, il team della Super App (SAT - Super App Team) integra il framework nel codice sorgente della Super App. La Super App utilizza le funzioni pubbliche offerte dalla Mini App per accedere e visualizzare l'interfaccia utente della Mini App, così come per gestire la logica di business specifica.

Benefici dell'integrazione delle Mini App:

- **Modularità:** Ogni Mini App è pensata e implementata come un modulo indipendente e ciò consente che essa possa essere successivamente aggiornarla o modificata senza che tale modifica si rifletta sul resto dell'app. Ciò favorisce una gestione più agile e consente una rapida introduzione di nuovi servizi.

- **Scalabilità:** L'approccio modulare e l'interfaccia standard consentono di aggiungere facilmente nuove Mini App senza compromettere la performance o la manutenibilità della Super App. Ogni nuova Mini App segue lo stesso schema e può essere integrata rapidamente.

- **Indipendenza:** Le Mini App rimangono indipendenti dal resto della Super App. Ogni team di sviluppo può concentrarsi sul miglioramento di un singolo servizio, mentre la Super App si occupa della gestione dell'integrazione e dell'interazione tra i vari moduli.

- **Semplicità nella gestione:** L'utilizzo di un'interfaccia standard evita la necessità di implementare logiche complesse per gestire diverse Mini App, riducendo il rischio di errori e facilitando la manutenzione a lungo termine.

L'integrazione delle diverse Mini App all'interno della Super App utilizzando framework modulari e interfacce standard si è dimostrata una strategia molto efficace per gestire ecosistemi complessi e in espansione. Inoltre questo tipo di approccio può consentire alla Super App di evolversi facilmente aggiungendo nuovi servizi ma mantenendo al contempo una struttura chiara, solida scalabile e facilmente manutenibile. La separazione tra la logica della Super App e quella delle Mini App offre numerosi vantaggi, tra cui una gestione semplificata degli aggiornamenti, una migliore performance e una user experience coerente e fluida.

3.1.2 Microfrontend

Le Super App stanno diventando sempre più popolari grazie alla loro capacità di integrare una vasta gamma di servizi e funzionalità in un'unica piattaforma, offrendo agli utenti un'esperienza fluida e centralizzata. Queste applicazioni, che spaziano dal pagamento ai servizi di messaggistica, dall'e-commerce ai trasporti, consentono agli utenti di soddisfare molteplici necessità all'interno di un singolo ecosistema digitale. Tuttavia, man mano che queste piattaforme crescono in complessità, diventa fondamentale gestire in modo efficace le molteplici funzionalità e garantire una buona esperienza utente. È proprio qui che entra in gioco il concetto di Microfrontend. [25]

Il modello di Microfrontend si ispira all'architettura dei microservizi, ma applicato al livello dell'interfaccia utente. Consente di scomporre applicazioni front-end complesse in moduli più piccoli e indipendenti, ognuno dei quali può essere sviluppato e distribuito autonomamente da diversi team. Questo approccio si rivela particolarmente utile nelle Super App, che integrano vari servizi e funzionalità, poiché consente di migliorare la scalabilità, la manutenzione e l'autonomia dei team di sviluppo, garantendo al contempo un'esperienza utente coerente e fluida. [26]

Il termine Microfrontend è stato introdotto nel 2016 su ThoughtWorks Technology Radar ed è un modello architettonico ispirato ai microservizi, ma applicato al front-end. Come i microservizi, il Microfrontend suddivide un'applicazione front-end monolitica in componenti più piccoli, indipendenti e facilmente distribuibili. Ogni team si concentra su una specifica area di business, sviluppando funzionalità end-to-end, dal database all'interfaccia utente. [27]

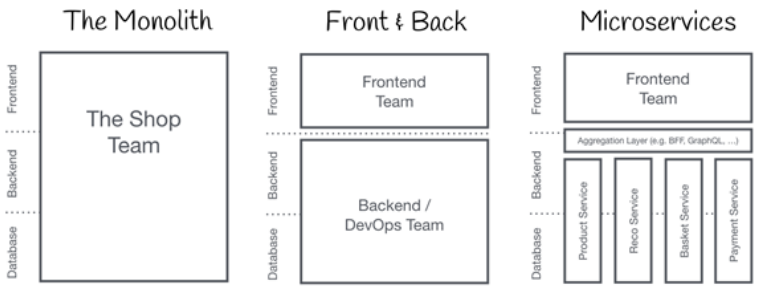

Figura 3.4: Il Microfrontend

I vantaggi del Microfrontend

I Microfrontend offrono numerosi vantaggi, tra cui la possibilità per i team di sviluppare e distribuire componenti in modo indipendente, riducendo dipendenze e colli di bottiglia. Questo porta a cicli di rilascio più rapidi, maggiore agilità e un miglior time-to-market. Inoltre, suddividendo un'applicazione frontend in componenti più piccoli, si ottiene una scalabilità mirata, con ogni componente che può essere scalato autonomamente in base alle necessità. La flessibilità tecnologica è un altro punto forte, poiché ogni componente può essere sviluppato utilizzando gli strumenti più adatti. Infine, i Microfrontend promuovono l'autonomia dei team, migliorando la collaborazione, la proprietà del codice e aumentando la produttività.[28]

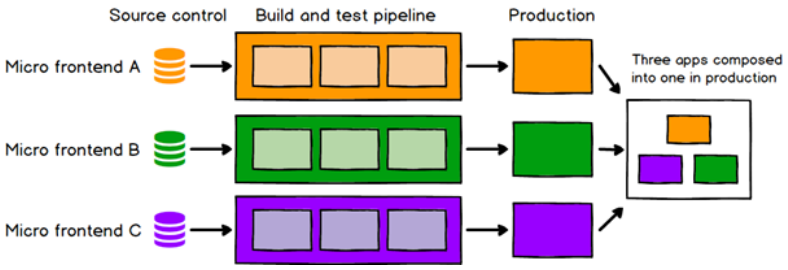

Figura 3.5: Vantaggi del Microfrontend

La comunicazione tra applicazioni

La comunicazione tra i Microfrontend dovrebbe essere minimizzata per evitare accoppiamenti troppo stretti, ma talvolta è necessaria. Gli eventi personalizzati o il passaggio di *callback* e dati espliciti (come in React) sono opzioni per la comunicazione tra componenti, evitando stati condivisi che possono creare accoppiamento e complicare le modifiche. Il routing, se utilizzato, richiede test automatizzati per garantire la corretta gestione dei percorsi.

Alcuni elementi, come immagini, CSS e dati JSON, dovrebbero essere condivisi tra i Microfrontend, ma la gestione delle librerie condivise è delicata, poiché duplicare le dipendenze può causare inefficienze. La condivisione del codice deve essere fatta con attenzione per evitare complessità non necessarie.

Il modello Backend For Frontend (BFF) è utile quando i team sviluppano indipendentemente i frontend, in quanto fornisce backend dedicati per ciascun Microfrontend, riducendo le dipendenze inter-team e migliorando l'autonomia.

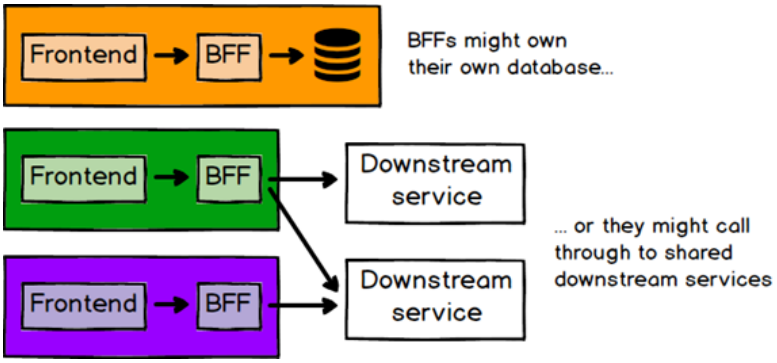

Figura 3.6: Il modello BFF

Module Federation

La Module Federation consente di caricare dinamicamente moduli tra applicazioni, favorendo una gestione modulare dell'architettura. Gli host integrano moduli remoti, mentre gli host bidirezionali possono anche esporre i loro moduli. Questo approccio facilita l'integrazione di moduli sviluppati in modo indipendente. [29]

Tuttavia, i Microfrontend, presentano anche degli svantaggi. La loro caratteristica di duplicare le dipendenze, comporta necessariamente un aumento delle dimensioni dei bundle e di conseguenza un calo delle prestazioni. Per ovviare ameno in parte a ciò si possono esternalizzare le dipendenze comuni. Inoltre il fatto che i Microfrontend si sviluppano in maniera indipendente può avere come conseguenza una certa difficoltà di integrazione soprattutto in quei casi, non rari, in cui l'ambiente di sviluppo e di produzione non sono gli stessi. E ancora la loro

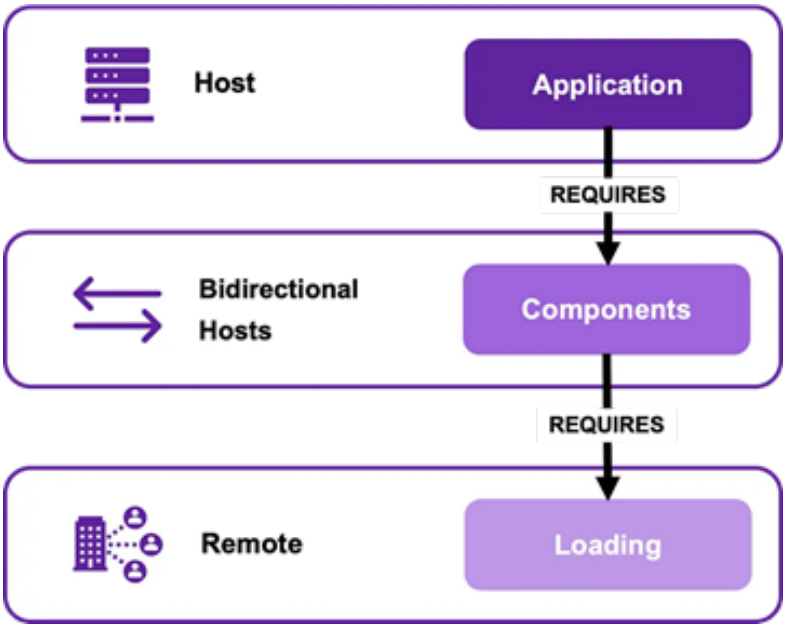

Figura 3.7: Module Federation

gestione complessa, basata sull'utilizzo di più repository, pipeline e server, richiede un'attenta e preventiva valutazione dei costi e dei benefici.

3.2 Tecnologie e strategie per la progettazione di una Super App

Come abbiamo visto, il cuore di una Super App risiede nella sua architettura modulare, un principio che trova una connessione diretta con il concetto di Microfrontend.

Tuttavia, la creazione di una Super App non si limita alla sola struttura dell'interfaccia utente, ma coinvolge anche scelte tecnologiche fondamentali che ne determinano le prestazioni, la scalabilità, la manutenibilità e l'integrazione con i vari servizi. Esploreremo, quindi, le diverse alternative tecniche disponibili per la progettazione di una Super App, con particolare attenzione alle soluzioni che supportano l'approccio modulare, come GeneXus, Ionic, Re.Pack e le Webview.

GeneXus

GeneXus è una piattaforma innovativa che consente lo sviluppo automatizzato di applicazioni multi-piattaforma, riducendo i tempi e i costi di produzione grazie all'uso di modelli ad alto livello e intelligenza artificiale.Un aspetto che caratterizza GeneXus 18 è la sua capacità di trasformare un'app nativa pre esistente in una Super App. Questo avviene tramite l'utilizzo del modulo GeneXus Render che permette di caricare in maniera dinamica le singole Mini App. Queste ultime quindi possono essere sviluppate sia internamente sia da terze parti in maniera autonoma, e caricate solo al momento dell'uso, offrendo in tal modo agli utenti finali un'esperienza dinamica e fluida.

1. Integrazione di GeneXus Render per caricare dinamicamente le Mini App.

2. Catalogazione delle Mini App nel Mini App Center, da cui verranno selezionate e autorizzate.

3. Progettazione dell'interfaccia utente per visualizzare e interagire con le Mini App.

4. Integrazione delle funzioni comuni (come pagamenti) per semplificare l'interazione tra le Mini App.

5. Sviluppo e distribuzione delle Mini App, inclusa la loro pubblicazione nel Mini App Center.

6. Revisione e test da parte degli amministratori della Super App prima che le Mini App siano disponibili per l'utente. [30]

Ionic

Ionic è una piattaforma di sviluppo app mobile che adotta un approccio "web-first", sfruttando HTML, CSS e JavaScript per creare app versatili e compatibili con iOS, Android, desktop e PWA. Con Ionic, è possibile integrare esperienze web (microapp) in app native utilizzando una suddivisione in Microfrontend. [31]Questa strategia permette una progettazione modulare delle Super App, dove ogni parte è integrata in modo dinamico, riducendo i costi e ottimizzando l'esperienza utente. [32]

Re.Pack

Re.Pack è un'alternativa al bundler predefinito di React Native, basato su Webpack, che offre maggiore controllo sul processo di creazione del bundle. [33]La sua caratteristica principale è il supporto per il code-splitting e la Module Federation. [34]Questi strumenti permettono di suddividere le funzionalità dell'app in moduli separati, caricabili su richiesta, e di integrare diverse applicazioni in una Super App. [35] Con la Module Federation, è possibile aggiornare dinamicamente i moduli senza dover ridistribuire l'intera app, ottimizzando le prestazioni e la gestione delle risorse. [36]

Webview

Nell'attuale scenario tecnologico, in cui un singolo brand può avere molteplici verticali e una Super App può servire diversi brand, una app ibrida rappresenta una scelta vantaggiosa. Questo tipo di applicazione offre numerosi benefici, tra cui aggiornamenti OTA (Over The Air), dimensioni ridotte dell'app, economicità e supporto multipiattaforma, solo per citarne alcuni.

Mentre in Occidente l'idea di una Super App è stata accolta con una certa freddezza, essa è estremamente popolare in Cina e nel Sud-Est asiatico. WeChat di Tencent, Grab (dalla Malesia) e Gojek (Indonesia) sono alcuni esempi noti di Super App in cui l'utente utilizza una singola applicazione per una vasta gamma di servizi. Ognuno di questi servizi, in effetti, è solo una web app che dipende dalla Super App per l'autenticazione, l'identità e il contesto del cliente. La web app viene visualizzata utilizzando una Webview.

Una Webview è fondamentalmente il motore di rendering di un browser caricato all'interno dell'app. Pensatela come un browser all'interno della vostra applicazione che occupa la massima area dello schermo disponibile e che è in grado di caricare qualsiasi URL. [37]

Le Webview sono una parte importante delle applicazioni mobili poiché ci permettono di accedere a un sito web all'interno della nostra applicazione stessa, anziché portarci a un browser esterno. Questo può risultare utile quando dobbiamo mostrare una risorsa web esterna nella nostra applicazione senza doverla sviluppare specificamente per l'app mobile, il che sarebbe costoso e dispendioso in termini di tempo. In questo caso è più vantaggioso utilizzare una Webview per visualizzare il contenuto della nostra app, e questa procedura è spesso utilizzata all'interno delle Super App.

Tale tipo di approccio si rileva infatti vantaggioso in quanto consente di visualizzare contenuti dinamici senza la necessità di sviluppare funzionalità native per ogni singolo servizio, risparmiando in tal modo tempo e risorse. Le app ibride che utilizzano Webview, per esempio quelle basate su Cordova, Ionic o Capacitor permettono inoltre di integrare più servizi in un unico codice base. Tuttavia, l'utilizzo di Webview può ridurre le prestazioni rispetto a soluzioni completamente native.

App Ibride

Un'app ibrida è una soluzione che combina elementi di app native e web. Queste applicazioni, come quelle sviluppate con Cordova, Ionic o Capacitor, permettono di scrivere una sola volta il codice e adattarlo a diverse piattaforme. Anche se le app ibride sono più economiche e versatili, possono avere delle limitazioni in termini di prestazioni rispetto alle app completamente native.

Tra i vantaggi delle app ibride possiamo annoverare:

- Codice riutilizzabile su più piattaforme.
- Possibilità di usare pacchetti e plugin NPM per estendere le funzionalità.
- Curva di apprendimento ridotta, soprattutto per chi conosce già framework come Angular.

Come svantaggi emergono:

- Prestazioni inferiori rispetto alle app native.
- Rischio di degrado della prestazione della Webview a causa di uso frequente di codice boilerplate.
- Aumento delle dimensioni dell'app a causa dei plugin NPM non ottimizzati.

Uno degli aspetti fondamentali per lo sviluppo di app ibride e a cui bisogna prestare massima attenzione, è la scelta del framework più adatto, perché esso

incide in maniera significativa sulle prestazioni dell'app e sulla user experience. I framework per le app ibride si suddividono in due principali categorie, in relazione a come esse gestiscono l'interfaccia utente e interagiscono con il dispositivo:

1. **UI WebView-based:** questi framework si basano su una tecnica particolare, ovvero utilizzano un "guscio" nativo per visualizzare il contenuto web all'interno dell'applicazione, e così facendo consentono lo sviluppo di soluzioni multi-piattaforma con un'unica base di codice. Di contro però, l'uso della WebView può limitare le prestazioni rispetto alle app native. Tra i principali framework di questa categoria troviamo **Cordova**, **Ionic** e **Capacitor**.

2. **UI Native-based:** in questo caso, l'interfaccia utente è costruita utilizzando componenti nativi della piattaforma, garantendo prestazioni superiori e un'esperienza utente più fluida. Tuttavia, questo approccio introduce una maggiore complessità nello sviluppo rispetto alle soluzioni WebView-based. Alcuni tra i framework più noti di questa categoria includono **React Native**, **Xamarin**, **NativeScript** e **Flutter**. [38]

3.3 L'Offline-First nelle Super App: le tecnologie coinvolte

Nel contesto delle Super App, l'approccio Offline-First si presenta come una funzionalità interessante da esplorare per offrire esperienze utente più fluide e robuste. Questo approccio consente alle applicazioni di funzionare senza interruzioni, anche quando la connessione di rete è instabile o assente, salvando e sincronizzando i dati localmente. In questo paragrafo, esploreremo le soluzioni tecniche che rendono possibile l'implementazione dell'Offline-First nelle Super App, dalle **Progressive Web Apps (PWA)** a l'uso di database locali come **Couchbase**, **PouchDB** e di tecnologie come le **Shared Preferences**. Discuteremo anche delle strategie di caching e sincronizzazione, evidenziando come queste possano essere integrate per migliorare l'esperienza complessiva degli utenti.

3.3.1 Progressive Web Apps (PWA)

Le Progressive Web Apps rappresentano uno dei principali strumenti per implementare un'esperienza Offline-First. Queste applicazioni combinano il meglio delle applicazioni web e native, sfruttando le potenzialità dei browser moderni e delle API più recenti. Le PWA sono progettate per offrire un'esperienza simile a quella di un'app nativa, ma senza la necessità di scaricare un'app dal negozio, rendendo il processo di installazione estremamente veloce e privo di frizioni. [39]

Le principali caratteristiche delle PWA che le rendono particolarmente adatte a un approccio Offline-First includono:

- **HTTPS:** Le PWA sono sempre servite tramite connessioni sicure, garantendo che i dati siano trasmessi in modo sicuro e privato.
- **Service Workers:** I Service Worker sono script che girano in background, indipendenti dalle pagine web, e consentono la gestione della cache, l'intercettazione delle richieste di rete e l'elaborazione dei dati in modo indipendente dalla connessione di rete. Grazie a questi, una PWA può funzionare perfettamente anche offline, rispondendo alle richieste con risorse precedentemente memorizzate. [40]
- **Web App Manifest:** Questo file JSON permette di definire le informazioni chiave sull'applicazione, come l'icona, il nome e le versioni, permettendo agli utenti di "installare" la PWA direttamente sulla schermata principale del dispositivo, senza dover passare da un app store.

L'utilizzo di Service Worker e CacheStorage consente alle PWA di memorizzare nella cache risorse essenziali, come file CSS, immagini e script JavaScript. Quando l'utente è offline, la PWA può ancora fornire contenuti e risorse memorizzate nella cache, ottimizzando così l'esperienza anche in assenza di connessione internet.

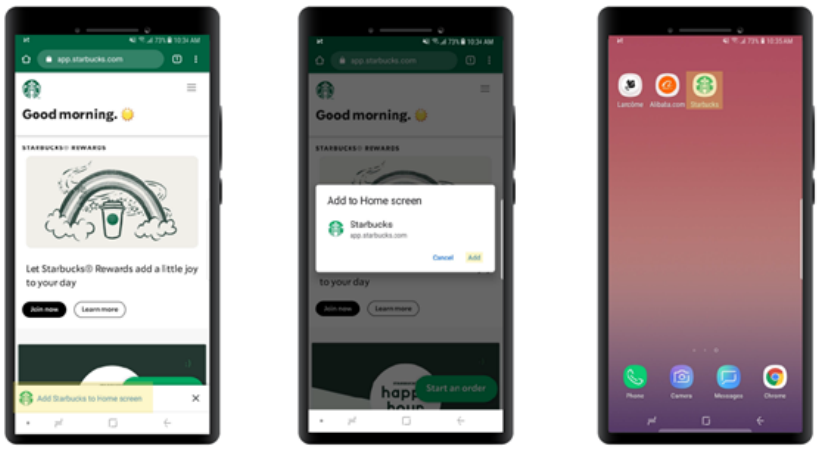

Figura 3.8: Esempio di PWA

3.3.2 Database locali

Affinché un'applicazione funzioni correttamente in modalità offline, è necessario disporre di un sistema di archiviazione dati locale che consenta di memorizzare

temporaneamente i dati sul dispositivo dell'utente e successivamente sincronizzarli con il server quando la connessione è ripristinata. Esistono diverse tecnologie che possono essere utilizzate per implementare questa funzionalità.

Couchbase Mobile

Couchbase Mobile è una suite di strumenti che supporta l'archiviazione locale e la sincronizzazione dei dati in modo efficiente. Couchbase Lite è il componente che consente di archiviare i dati localmente, sia su piattaforme iOS che Android, mentre Couchbase Sync Gateway gestisce la sincronizzazione dei dati tra il database locale e un server Couchbase remoto. Questo sistema è particolarmente utile per le applicazioni mobili, dove la connessione di rete può essere intermittente.

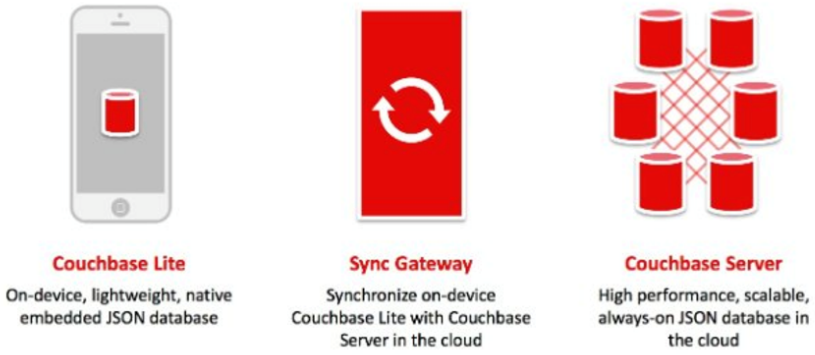

Figura 3.9: Couchbase Mobile

PouchDB

PouchDB è una libreria JavaScript che fornisce un potente sistema di archiviazione locale su dispositivi mobili e browser web. PouchDB consente di memorizzare e sincronizzare i dati in modo asincrono. La sua caratteristica distintiva è la capacità di replicare i dati con qualsiasi server che implementi il protocollo CouchDB, come ad esempio Couchbase Server o Cloudant. PouchDB è particolarmente utile quando si desidera garantire che i dati siano sempre disponibili, anche in assenza di rete, e che possano essere sincronizzati una volta che la connessione è ripristinata.

3.3.3 Strumenti per il Caching e la memorizzazione dei dati

Oltre a pensare a soluzioni di archiviazioni avanzate, è fondamentare implementare sistemi di caching idonei a ottimizzare l'esperienza offline.

Service Workers e Cache Storage

I Service Workers, di cui abbiamo parlato in precedenza, sono il cuore dell'implementazione Offline-First. [41] Considerato che essi lavorano in background, essi sono in grado di intercettare le richieste di rete e possono provvedere mediante utilizzo di risorse memorizzate nella cache. L'uso della Cache Storage API inoltre consente di memorizzare in modo duraturo risorse web, come immagini, file JavaScript e CSS, rendendo disponibili le applicazioni anche in assenza di una connessione di rete. [42]

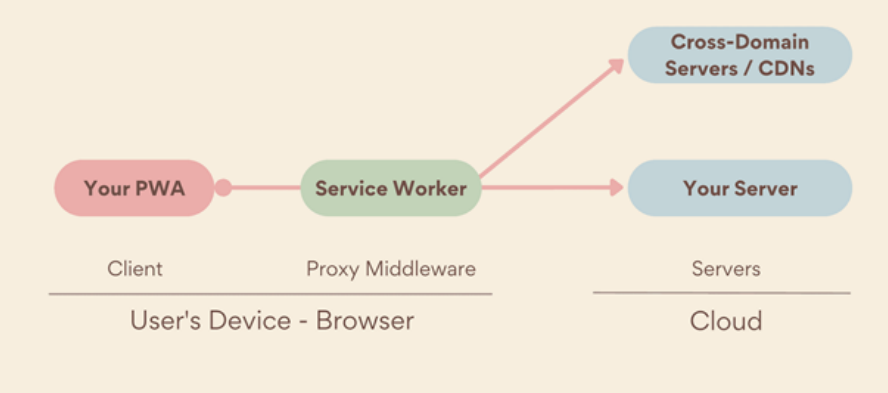

Figura 3.10: Service Worker

IndexedDB e LocalStorage

Per applicazioni che richiedono un'archiviazione più complessa, IndexedDB è una soluzione potente. Si tratta di un database NoSQL che consente di archiviare grandi volumi di dati in modo persistente nel browser. Sebbene più complesso da implementare rispetto a LocalStorage, IndexedDB offre una maggiore flessibilità e capacità di archiviazione. [43]

D'altra parte, LocalStorage è un'opzione più semplice per memorizzare piccole quantità di dati direttamente nel browser. Sebbene abbia limitazioni di capacità (circa 5 MB), è comunque utile per applicazioni leggere che non necessitano di funzionalità avanzate come la ricerca o le transazioni.

3.3.4 Shared Preferences

Le Shared Preferences sono un sistema di archiviazione che memorizza dati in una coppia chiave-valore ed è particolarmente utile nelle applicazioni mobili per gestire piccole quantità di informazioni, come le preferenze dell'utente, le credenziali o i dati temporanei. In un contesto Offline-First, le Shared Preferences sono una risorsa

fondamentale per garantire che l'app continui a funzionare anche senza connessione internet, grazie al fatto che possono salvare e sincronizzare i dati localmente. [44]

Le Shared Preferences rappresentano quindi una soluzione al contempo semplice ed efficace per memorizzare dati in maniera persistente all'interno di un'applicazione. Sebbene esse non siano progettate per gestire ingenti volumi di dati, il loro utilizzo risulta particolarmente vantaggioso nell'ambito di applicazioni Offline-First, che hanno la necessità di memorizzare informazioni chiavi in locale al fine di garantire all'utente un'esperienza semplice e fluida anche senza connessione a internet. Di seguito vengono elencati alcuni scenari in cui l'uso delle Shared Preferences è particolarmente vantaggioso:

1. **Memorizzazione delle preferenze utente:** Le Shared Preferences sono ideali per salvare configurazioni leggere e preferenze dell'utente, come il tema dell'app, la lingua e altre impostazioni generali. Queste informazioni non richiedono una sincronizzazione complessa e possono essere recuperate rapidamente anche quando l'app è offline.

2. **Gestione delle credenziali e dei token di accesso:** Le Shared Preferences sono comunemente usate per memorizzare informazioni di autenticazione, come i token di accesso, per mantenere l'utente autenticato anche quando non è disponibile una connessione di rete. Questo è fondamentale per le Super App, che spesso supportano molteplici funzionalità, come social, acquisti e pagamenti.

3. **Gestione della cache dei dati:** Per un'app Offline-First, è necessario avere un sistema per memorizzare temporaneamente i dati, come articoli o informazioni scaricate da un'API. Sebbene le Shared Preferences non siano progettate per archiviare grandi volumi di dati, sono utili per salvare piccole quantità di informazioni che devono essere disponibili anche in modalità offline.

4. **Memorizzazione dello stato delle operazioni in corso:** Le Shared Preferences sono anche adatte per gestire lo stato di operazioni in corso, come transazioni o processi che devono essere completati quando la connessione Internet viene ripristinata. Possono memorizzare informazioni temporanee relative a un'operazione, come il suo stato o eventuali errori, per garantire che l'utente possa riprendere da dove aveva lasciato.

5. **Sincronizzazione dei dati al ritorno online:** In un'app Offline-First, la sincronizzazione dei dati è un aspetto cruciale. Le Shared Preferences possono essere usate per salvare temporaneamente i dati che devono essere inviati al server una volta che la connessione viene ripristinata. Questo consente all'app di continuare a funzionare offline e di inviare i dati non appena la rete è disponibile.

Vantaggi e limitazioni delle Shared Preferences nell'Offline-First

L'utilizzo delle Shared Preferences in un'architettura Offline-First offre diversi vantaggi, ma presenta di contro anche alcune limitazioni. Analizziamo di seguito i principali pro e contro di tale scelta di archiviazione.

Vantaggi:

- **Semplicità:** Le Shared Preferences sono di facile utilizzazione e configurazione, ideali dunque per dati semplici come per esempio le preferenze utente.
- **Velocità:** Nelle Shared Preference il recupero dei dati è immediato e questo le rende la scelta ideale per la memorizzazione di informazioni leggere e per accessi rapidi.
- **Persistenza:** I dati memorizzati nelle Shared Preferences rimangono disponibili anche dopo che l'app viene chiusa o riavviata, e questo garantisce una buona conservazione delle informazioni locali.

Limitazioni:

- **Capacità limitata:** Le Shared Preferences sono pensate per piccole quantità di dati. Non sono ideali per memorizzare grandi volumi di dati, come file multimediali o grandi database.
- **Non strutturate:** Le Shared Preferences sono più adatte per dati semplici (come stringhe o numeri) e non per dati complessi che richiedono query avanzate. [45]

3.4 La scelta del Framework

Nel contesto delle Super App, l'approccio Offline-First è importante per garantire esperienze utente fluide e performanti, anche quando la connessione di rete è instabile o addirittura assente. Come abbiamo visto, la chiave per implementare un'app Offline-First risiede nell'utilizzo di tecnologie che permettano la memorizzazione dei dati localmente, la gestione della sincronizzazione con i server e la cache dei contenuti, in modo da assicurare una fruizione continua delle funzionalità senza dipendere dalla connessione a Internet.

Abbiamo inoltre visto, come diverse tecnologie contribuiscono a questo obiettivo.

Tuttavia, tutte queste tecnologie devono essere integrate in un framework di sviluppo che possa supportare la costruzione dell'intera applicazione. La scelta del framework giusto è cruciale, non solo per la gestione dell'approccio Offline-First, ma

anche per la scalabilità, la manutenibilità e la facilità di sviluppo dell'applicazione nel lungo periodo.

Dopo aver esplorato le diverse tecnologie Offline-First, la domanda successiva diventa: quale framework scegliere per sviluppare la nostra Super App? In particolare, due dei framework più popolari per lo sviluppo di app mobili cross-platform sono Flutter e React Native. Entrambi offrono soluzioni robuste e potenti, ma ciascuno ha delle peculiarità che lo rendono adatto a determinate necessità.

La scelta tra Flutter e React Native dipenderà quindi da una serie di fattori, tra cui il tipo di applicazione da sviluppare, le esigenze di performance, la familiarità con i linguaggi e le librerie, e la strategia di gestione dei dati offline. Entrambi i framework supportano perfettamente la costruzione di applicazioni che utilizzano le tecnologie di caching, sincronizzazione e archiviazione locale, ma l'approccio di ciascuno a questi aspetti potrebbe variare, influenzando l'implementazione delle soluzioni Offline-First.

Ora siamo pronti per esaminare più nel dettaglio il confronto tra Flutter e React Native, tenendo conto delle tecnologie Offline-First che potrebbero essere integrate nelle Super App, per determinare quale sia la scelta migliore per le esigenze specifiche del progetto.

React Native

React Native è un framework open source basato su JavaScript per la creazione di app mobili native per iOS e Android. Si tratta di un framework multi-piattaforma perché, partendo dalla stessa codebase, genera applicazioni per piattaforme diverse (iOS, Android, web, Windows).

React Native si basa su React, che a sua volta si basa su JavaScript. Quindi è la scelta più semplice e veloce per molti sviluppatori React e web per entrare nel mondo nativo.

React Native sfrutta un ponte tra l'applicazione e la piattaforma per gestire la loro comunicazione. Se un utente preme il pulsante, React Native traduce questo evento nell'evento che JavaScript può gestire.

Figura 3.11: React Native

I framework nativi principalmente compilano i componenti dell'interfaccia utente in componenti UI nativi. Non "convertono" effettivamente la codebase nel codice della piattaforma nativa. Sono solo i componenti dell'interfaccia utente del codice JavaScript che vengono effettivamente convertiti nei corrispondenti componenti UI nativi nella piattaforma di destinazione.

Figura 3.12: React Native 2

Una delle principali forze di React Native è il superamento di Webview. Webview è un componente principale di iOS e Android in grado di visualizzare contenuti web richiamando un vero browser. Quindi gli sviluppatori possono scrivere codice web e vedere il risultato nelle applicazioni native.

Tuttavia, le prestazioni non sono ottimizzate perché non si stanno utilizzando le capacità native, ma si sta richiedendo il lavoro a un browser. Invece, React Native ha trovato un modo per sfruttare i vantaggi nativi senza perdere la funzionalità del codice web.

Nello sviluppo Android, si scrivono viste in Kotlin o Java; nello sviluppo iOS, si utilizza Swift o Objective-C. Con React Native, è possibile richiamare queste viste con JavaScript utilizzando i componenti React.

Durante l'esecuzione, React Native crea le viste Android e iOS corrispondenti per quei componenti. Poiché i componenti React Native sono supportati dalle stesse viste di Android e iOS, le app React Native hanno l'aspetto e le prestazioni di qualsiasi altra app. Questi componenti supportati dalla piattaforma vengono chiamati Native Components.

React Native viene fornito con un set di Native Components essenziali, pronti per l'uso, che è possibile utilizzare per iniziare a costruire la propria app. Questi sono i Componenti Principali di React Native.

Un'altra importante caratteristica è il codice specifico della piattaforma. React Native fornisce il modulo Platform per rilevare in quale piattaforma e versione

il codice è in esecuzione. Quindi gli sviluppatori possono discriminare parte del codice se l'app è su iOS o Android. Estendendo questa funzionalità, React Native supporta estensioni specifiche della piattaforma; un singolo file JavaScript può essere fornito in diverse estensioni, ad esempio *Container.ios.js*, *Container.android.js*, *Container.native.js* e *Container.js*. React Native selezionerà il file più specifico in base alla piattaforma corrente.

Flutter

Flutter è un framework open source per lo sviluppo di applicazioni mobile multi-piattaforma sviluppato da Google. Consente agli sviluppatori di creare app native di alta qualità per dispositivi iOS e Android con un'unica base di codice. Flutter ha il pregio di consentire agli sviluppatori di poter creare in maniera rapida e con facilità app gradevoli e potenti e che funzionano in maniera nativa su entrambe le piattaforme maggiormente diffuse, ovvero Android e Ios.

Esso utilizza il linguaggio di programmazione open source Dart e presenta uno stile di programmazione accattivante e reattivo e ciò rende lo sviluppo delle app particolarmente facile e anche rapido. Risulta inoltre caratterizzato da un layout avanzato che consente la creazione di componenti UI anche complessi con una certa facilità.

Ancora, Flutter contiene al suo interno una vasta gamma di set di API e di librerie specifiche per la piattaforma e questo agevola gli sviluppatori che possono creare app per le due piattaforme citate utilizzando lo stesso linguaggio di programmazione. Per concludere Flutter, basandosi su Dart è una risorsa open source e quindi completamente gratuita e questo lo rende particolarmente appetibile nei confronti di quegli sviluppatori che sono alla ricerca di uno strumento semplice ed economico per realizzare le loro app mobili. Esso, grazie ai suoi veloci cicli di iterazioni, al suo ricco set di funzionalità e alla svariata gamma di piattaforme supportate, fornisce agli sviluppatori una piattaforma potente e semplice per la creazione di app mobili belle e funzionali.

Quanto è maturo Flutter? Un rapido flashback sull'evoluzione di Flutter: Un team di Google ha creato Flutter come progetto open source. Pertanto, sia Google che la comunità Flutter contribuiscono al suo sviluppo. La Figura 3.13 mostra le principali milestones di Flutter. [46]

3.4.1 Flutter vs React Native

La rivalità tra Flutter e React Native affonda le sue radici nel lancio di React Native da parte di Facebook nel 2015 e di Flutter da parte di Google nel 2017.

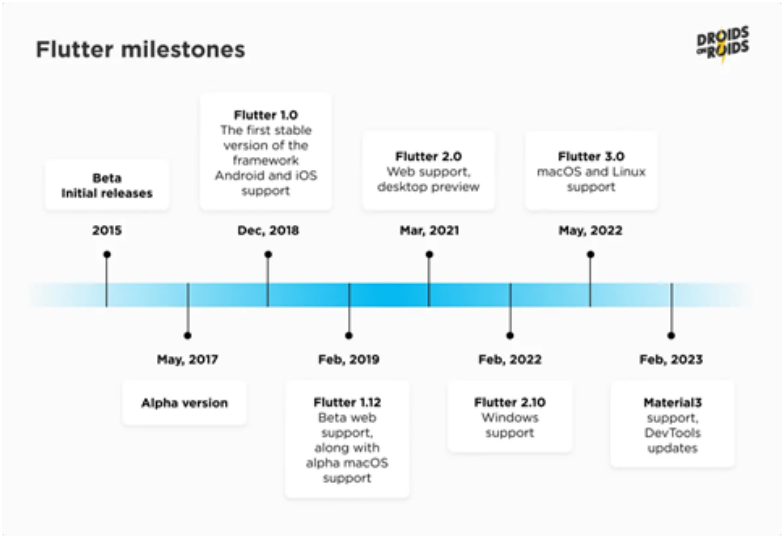

Figura 3.13: Le milestones di Flutter

React Native, basandosi sulla popolarità di JavaScript, ha rapidamente guadagnato un forte punto d'appoggio tra gli sviluppatori.

Flutter è entrato in scena più tardi, offrendo una nuova prospettiva con il suo linguaggio di programmazione Dart e il suo motore di rendering unico. Nel corso degli anni, entrambe le piattaforme si sono evolute, competendo per il dominio nell'arena dello sviluppo di app multipiattaforma. Questa rivalità ha portato a progressi significativi in entrambi i framework, avvantaggiando sia gli sviluppatori che le imprese.

Cercando i migliori linguaggi per lo sviluppo cross-platform mobile, è possibile trovare molti framework, quindi il nostro obiettivo è sceglierne il migliore. Secondo il grafico seguente fornito da Statista, sembra che negli ultimi anni gli sviluppatori abbiano perso interesse in ogni framework tranne Flutter. Possiamo notare una convergenza verso due principali concorrenti: React Native e Flutter.

Quindi confrontiamoli per trovare le loro differenze, applicazioni, vantaggi e svantaggi.

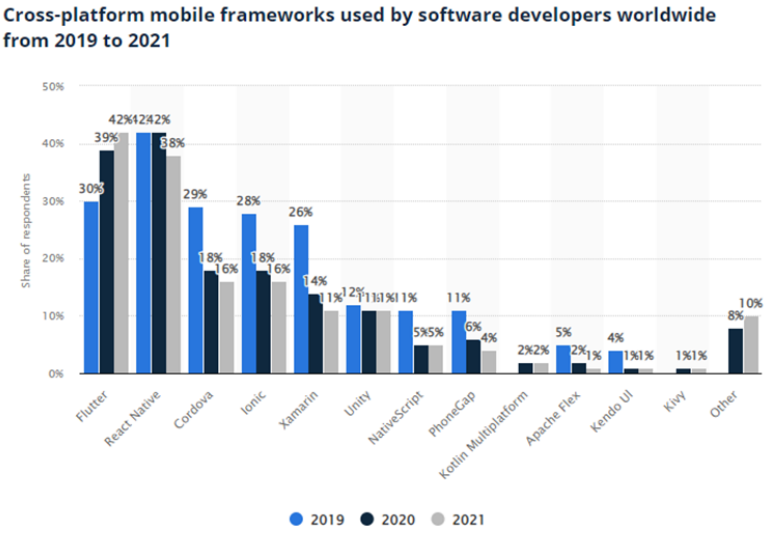

Figura 3.14: Popolarità dei cross-platfrom frameworks

Punti in comune: Ci sono certamente delle differenze tra Flutter e React Native, ma al contempo non possiamo non far notare alcune similarità. Entrambi sono framework open source ed entrambi hanno il pregio di essere progettati per lo sviluppo di app senza essere vincolati a una specifica piattaforma, consentendo quindi la creazione di app sia per Android sia per iOS, partendo da un'unica base di codice. Entrambi evidenziano processi di sviluppo molto efficienti e performance elevate grazie a funzionalità come **hot reload** e **fast refresh**. Inoltre, entrambi

i framework godono di un forte sostegno da parte della comunità, fornendo una vasta gamma di risorse, librerie e strumenti. Questa comunanza sottolinea il loro obiettivo condiviso: semplificare il processo di sviluppo e ridurre la complessità della creazione di applicazioni multipiattaforma. [47]

Feature	Flutter	React Native
Developed by	Google	Facebook
Programming language	Dart	JavaScript
Rendering	Native	JavaScript Bridge
Cross-platform	Yes, for mobile, web, desktop and embedded devices	Yes, but it's only for mobile iOS & Android
UI flexibility	High	Medium
Learning curve	Easy	Medium
Ecosystem	Large	Large
Performance	Faster	Slower
Productivity	High	Medium
Cost-effectiveness	More cost-effective because it's for all platforms	Cost-effective only for mobile
Hot-Reload / Fast Refresh	Works with all types of changes	Only code changes

Figura 3.15: Tabella di confronto: React Native vs Flutter

Performance: Quando si parla delle prestazioni di Flutter e React Native ci sono molti fattori da considerare. Sebbene entrambe le tecnologie possano ottenere buone prestazioni, Flutter si distingue ampiamente per l'utilizzo di Dart e per l'approccio di rendering distintivo. Compilando il codice Dart in codice macchina nativo, Flutter offre animazioni rapide, tempi di avvio delle app più brevi e un miglioramento complessivo delle prestazioni.

D'altra parte, React Native si basa su un bridge per facilitare la comunicazione tra JavaScript e i componenti nativi. Questo bridge introduce un sovraccarico, che può potenzialmente influire sulle prestazioni dell'app. A differenza di Flutter, dove il codice Dart viene compilato in anticipo, rendendolo simile al 100% nativo, React Native deve tradurre il codice JavaScript in codice nativo durante il runtime. Di conseguenza, React Native potrebbe non raggiungere lo stesso livello di ottimizzazione delle prestazioni di Flutter.

Condivisione del codice: Dalla stessa base di codice, Flutter genera app native per *iOS, Android, Windows, macOS, Linux e web.*

React Native, invece, supporta nativamente solo *iOS e Android.* Tuttavia, alcune librerie di terze parti consentono di ottenere compatibilità anche con *web e Windows 10.* React Native permette di scrivere codice specifico per una singola piattaforma, come Android, iOS o web, separando le implementazioni quando necessario.

Hot-reload vs Fast-refresh:

- **Fast-refresh:** React Native fornisce agli sviluppatori un modo conveniente per vedere le modifiche al codice riflesse in tempo reale durante il processo di sviluppo. Tuttavia, questo meccanismo funziona solo per le modifiche al codice.

- **Hot-reload:** Flutter consente agli sviluppatori di vedere l'impatto delle modifiche al codice quasi istantaneamente. La ricarica a caldo in Flutter non si limita alle sole modifiche al codice, ma supporta anche aggiornamenti alle risorse, come immagini, caratteri e stringhe localizzate. Questo approccio garantisce iterazioni più rapide ed efficienti nello sviluppo dell'interfaccia utente e delle funzionalità dell'app.

Test: Flutter supporta test unitari, di widget e di integrazione, accompagnati da una documentazione dettagliata. Fornisce istruzioni chiare per la creazione e il rilascio di app su *Play Store* e *App Store*, documentando anche il processo di distribuzione.

React Native supporta test a livello di unità (*Jest*), ma non ha un supporto ufficiale per i test UI, che devono essere eseguiti con librerie di terze parti come *Appium* o *Detox.* Anche il processo di compilazione e rilascio si basa su librerie esterne.

Automazione della compilazione e del rilascio: React Native non fornisce un processo automatizzato per la distribuzione delle app iOS su App Store. Sebbene esista documentazione a riguardo, il processo rimane in gran parte manuale. Tuttavia, strumenti di terze parti come *fastlane* possono essere utilizzati per distribuire le app React Native.

Flutter, invece, offre un'interfaccia a riga di comando robusta, che permette la creazione di un binario dell'app seguendo le istruzioni ufficiali per il rilascio su *Android* e *iOS.* Inoltre, il processo di distribuzione con *fastlane* è documentato ufficialmente.

Comunità e Manutenibilità: React Native può contare su una comunità più vasta in termini di progetti su GitHub. D'altro canto però la fase di debugging in React Native, considerato che si basa su diversi componenti nativi che dipendono da librerie di terze parti, che col tempo possono divenire obsolete e non più aggiornate, può spesso risultare un'operazione complessa.

Questo non accade invece con Flutter che offre una maggiore convenienza in termini di manutenibilità grazie alla sua architettura più semplice e alla gestione centralizzata dei componenti.

User Interface (UI):

- **Flutter:** L'app Flutter avrà un aspetto e un comportamento naturale su ogni piattaforma, grazie all'uso di due set di widget: *Material Design*, che implementa il design di Google, e *Cupertino*, che imita il design iOS.
- **React Native:** I componenti dell'app React Native assomigliano esattamente a quelli nativi. Ad esempio, un pulsante su un dispositivo iOS avrà lo stesso aspetto di un pulsante nativo iOS, e lo stesso vale per Android. L'uso di componenti nativi garantisce che eventuali aggiornamenti all'interfaccia utente del sistema operativo vengano riflessi automaticamente nell'app.

Componenti UI: React Native dipende fortemente da librerie di terze parti per accedere ai moduli nativi. Flutter, invece, include nativamente un set di componenti di rendering UI, accesso alle API dei dispositivi, navigazione, testing e gestione dello stato. Questa caratteristica elimina la necessità di affidarsi a librerie esterne per lo sviluppo dell'applicazione.

Efficienza dei costi: Quando si considera l'ottimizzazione del budget, il primo fattore da esaminare è il tipo di applicazione da sviluppare.

Se l'obiettivo è creare un'applicazione multipiattaforma compatibile con *Android, iOS, Web, Linux, macOS, Windows* e dispositivi integrati (*CarPlay, ecc.*), Flutter rappresenta la scelta migliore. Grazie alla possibilità di utilizzare un'unica base di codice per tutte le piattaforme, Flutter permette di ridurre significativamente i tempi e i costi di sviluppo.

React Native, invece, è più efficiente nello sviluppo di app mobile, offrendo un'unica base di codice per *iOS* e *Android*. Tuttavia, nei progetti complessi e ricchi di funzionalità, potrebbe presentare delle limitazioni.

Linguaggio di Programmazione: Uno dei principali fattori che differenzia fortemente Flutter e React native è ovviamente il linguaggio di programmazione che essi utilizzano:

- Come abbiamo già avuto modo di dire Flutter utilizza il linguaggio open source *Dart*, sviluppato da Google all'inizio solo per la realizzazione di interfacce utente. Esso coniuga al proprio interno peculiarità di *JavaScript* e di *C#*, dando luogo a un linguaggio di programmazione altamente bilanciato, molto intuitivo e che ben si presta allo sviluppo moderno.

- React Native si basa invece soltanto su *JavaScript*, linguaggio diffusissimo, apprezzato e ampiamente adottato che contiene un vasto ecosistema interno di librerie e framework. Proprio la diffusione di JavaScript, unita alla sua particolare flessibilità rendono React Native fortemente apprezzato nella community open source e di conseguenza un'ottima scelta per la progettazione e realizzazione di applicazioni web e mobile. [48]

Popolarità e Crescita dei Framework: Negli ultimi anni, sia Flutter che React Native hanno registrato una crescita significativa all'interno delle rispettive comunità di sviluppatori.

Secondo i dati di *Google Trends* e *GitHub Stars*, Flutter ha guadagnato costantemente popolarità, superando React Native nel tempo. La sua rapida ascesa suggerisce una crescente preferenza tra gli sviluppatori per lo sviluppo di applicazioni multipiattaforma.

React Native, d'altro canto, continua a mantenere una solida base di utenti nel settore mobile, grazie alla sua compatibilità con JavaScript e alla vasta gamma di librerie disponibili.

Figura 3.16: Il livello di interesse nel tempo

I dati di Google Trends dimostrano che Flutter ha costantemente guadagnato popolarità, con un numero crescente di query di ricerca che indicano una crescente curiosità riguardo alle sue capacità. Al contrario, l'interesse di ricerca di React

Native è rimasto relativamente stabile o è cresciuto a un ritmo più lento. Ciò suggerisce che Flutter ha catturato con successo l'attenzione degli sviluppatori e generato una curiosità maggiore rispetto a React Native.

Inoltre, GitHub Stars, una misura della popolarità di un progetto e del coinvolgimento della comunità sulla piattaforma GitHub, supporta ulteriormente la crescente importanza di Flutter. I dati storici mostrano un chiaro schema in cui Flutter supera React Native in termini di stelle accumulate nel tempo. Questa metrica riflette il numero di sviluppatori che esprimono il proprio interesse e supporto per un particolare progetto pubblicandolo su GitHub.

3.5 Scelte progettuali

Nel processo di sviluppo della nostra Super App Offline-First, sono state fatte delle scelte cruciali che definiscono l'architettura e le funzionalità principali dell'applicazione. Queste decisioni riguardano l'adozione di Flutter come framework di sviluppo, l'utilizzo delle Webview per integrare le Mini App all'interno della Super App e l'impiego delle Shared Preferences per garantire il corretto funzionamento offline.

3.5.1 Flutter nelle Super App: un approccio differente

La scelta fatta di utilizzare Flutter come framework di sviluppo per la nostra Super App si è rivelata strategica per svariati motivi. Flutter in particolare si differenzia da qualsiasi altro prodotto simile per il suo approccio unico nel rendering dell'interfaccia utente. Esso infatti, anziché affidarsi a widget nativi basati su OEM (Original Equipment Manufacturer), utilizza il proprio motore di rendering per disegnare direttamente i pixel sull'interfaccia utente, sfruttando il canvas nativo. Questo approccio consente a Flutter di garantire prestazioni superiori e una maggiore affidabilità sulle piattaforme native, anche nelle situazioni di carico elevato e nell'esecuzione di operazioni intensive.

Vantaggi principali dell'utilizzo di Flutter:

- Maggiore performance e affidabilità: Poiché il rendering è gestito direttamente da Flutter, l'applicazione è più performante rispetto ad altri approcci che dipendono da componenti nativi.

- Esperienza utente coerente: Flutter assicura che l'interfaccia utente sia la stessa su tutte le piattaforme *(iOS, Android)*, grazie al motore di rendering uniforme. Questo è un vantaggio considerevole nelle Super App, dove l'utente si aspetta una esperienza fluida e senza interruzioni, indipendentemente dal dispositivo utilizzato.

Tuttavia, ci sono anche alcuni svantaggi:

- Nuovo linguaggio di programmazione **(Dart)**: Flutter richiede l'utilizzo di Dart, un linguaggio di programmazione meno diffuso rispetto a JavaScript, che può rappresentare una barriera iniziale per alcuni sviluppatori.

- Impossibilità di riutilizzare il codice web esistente: A differenza di altre soluzioni che potrebbero consentire una maggiore condivisione di codice con le versioni web delle app, Flutter richiede che il codice per la parte mobile sia scritto appositamente per il framework.

- Dimensioni dell'applicazione: Le app sviluppate con Flutter possono risultare più pesanti, in quanto includono il motore di rendering Flutter stesso e altre risorse necessarie per il suo corretto funzionamento.

3.5.2 Webviews nelle Super App: integrazione delle Mini App

Un'altra scelta chiave per la nostra Super App è stata l'adozione delle Webview per integrare Mini App all'interno della struttura principale. Le Webview consentono di visualizzare contenuti web direttamente nell'interfaccia nativa dell'app, permettendo di includere funzionalità complesse e servizi aggiuntivi senza dover aprire una nuova finestra del browser. Questo tipo di approccio permette all'utente di avere un'esperienza coerente, non essendoci interruzioni quando si passa dall'utilizzo della Super App alle Mini App.

Quando un'applicazione è realizzata con Flutter la gestione delle Webview è particolarmente semplice, in quanto avviene tramite widget dedicati che consentono di integrare i contenuti web senza intaccare la performance nativa. Le Super App e le Mini App comunicano tramite i canali Javascript, che consentono il dialogo tra il codice nativo (Flutter) e quello web (HTML, CSS, JavaScript).

Ad esempio, nella nostra Super App, una Mini App che gestisce l'autenticazione utente potrebbe utilizzare Webview per caricare la pagina di login, mentre la Super App Flutter gestirebbe la memorizzazione delle credenziali utente tramite un *JS-Native Bridge*, che garantisce sicurezza e coerenza nell'esperienza utente.

Vantaggi e Svantaggi dell'uso delle WebView

L'uso delle **WebView** rappresenta una soluzione efficace per integrare contenuti web all'interno di un'applicazione mobile, evitando la necessità di sviluppare app native separate. Tuttavia, presenta sia vantaggi che svantaggi, che devono essere considerati attentamente.

Vantaggi:

- **Integrazione fluida:** Le WebView permettono di integrare facilmente servizi web all'interno dell'app senza la necessità di creare app native separate.
- **Flessibilità:** Possibilità di modificare e aggiornare il contenuto web delle Mini App senza dover rilasciare nuove versioni della Super App.
- **Esperienza utente unificata:** Mantenimento di una UI coerente, senza dover aprire un'applicazione esterna.

Svantaggi:

- **Dipendenza dalla connessione internet:** Se la Mini App non è ottimizzata per l'utilizzo offline, la WebView potrebbe presentare limitazioni quando non è disponibile una connessione.

3.5.3 L'approccio Offline-First: le Shared Preferences

Nel contesto di un'app Offline-First, una delle scelte più rilevanti è stata l'utilizzo delle Shared Preferences per gestire la memorizzazione dei dati in modo persistente sul dispositivo dell'utente.

Come Flutter supporta l'approccio Offline-First con Shared Preferences
In un'app Offline-First, le Shared Preferences possono essere utilizzate per:

1. **Memorizzare dati temporanei**: Per esempio, i dati ricevuti dalle API quando l'utente è online, e poi visualizzati quando l'utente è offline.
2. **Gestire la cache locale**: Le Shared Preferences possono memorizzare le chiamate API e le risorse scaricate nella cache locale, migliorando, in tal modo, la velocità di caricamento e riducendo di conseguenza la necessità di accessi frequenti e ripetuti alla rete.
3. **Sincronizzazione dei dati**: È probabilmente la fase più delicata dell'intero processo. Una volta che la connessione sia stata ripristinata, i dati immagazzinati nelle Shared Preferences sono pronti a essere sincronizzati con il server remoto, cosicché l'utente possa continuare a lavorare su tali dati anche offline.

Notiamo che le Shared Preferences sotto questo aspetto si integrano alla perfezione con il modello Offline-First in quanto offrono un metodo affidabile e veloce per memorizzare piccole quantità di dati. Questa circostanza è, per le Super App, di cruciale importanza, perché consente loro di gestire correttamente i dati temporanei delle Mini App e al contempo garantire, al ripristino della connessione, che tutte le modifiche eventualmente fatte vengano correttamente sincronizzate con il server.

3.5.4 Conclusione

Flutter, con la sua architettura unica e la gestione avanzata delle interfacce utente, offre la base ideale per applicazioni complesse e multi-funzionali, come quelle delle Super App. Le Webview permettono di integrare i contenuti web in modo fluido, mentre le Shared Preferences rendono possibile un'esperienza Offline-First, fondamentale per garantire la continuità dell'uso dell'applicazione anche senza connessione.

Questa combinazione di tecnologie ha permesso di realizzare un'app mobile che non solo risponde alle esigenze moderne di accessibilità e performance, ma che si inserisce perfettamente nel concetto di Super App, rendendo l'esperienza utente più ricca, interattiva e stabile, indipendentemente dalle condizioni di rete.

Capitolo 4

Il Prototipo di Super App: Struttura e Comunicazione

Le Super App rappresentano una nuova frontiera nel mondo delle applicazioni digitali, offrendo un ecosistema integrato in cui diversi servizi convivono all'interno di un'unica piattaforma. Il nostro progetto si concentra sullo sviluppo di un prototipo di Super App, con l'obiettivo di esplorare le potenzialità dell'approccio scelto e analizzare le dinamiche di comunicazione tra la Super App e le Mini App che ne fanno parte.

Nei prossimi paragrafi approfondiremo gli aspetti chiave del progetto, dall'architettura adottata alle soluzioni implementate per garantire un'integrazione efficiente tra i vari moduli. Particolare attenzione sarà dedicata ai meccanismi di interazione e scambio di dati tra la Super App e le Mini App, elementi essenziali per assicurare un'esperienza utente coerente e senza interruzioni.

4.1 Introduzione

Lo sviluppo di questo prototipo di Super App prende origine da un'applicazione e-commerce esistente, inizialmente concepita con un approccio Offline-First. Partendo da questa base, il progetto è stato ampliato per integrare nuove funzionalità e supportare la comunicazione tra la Super App e le Mini App.

Lo scopo di tale progetto non è tanto andare a creare una Super App fatta e finita, quanto più andare a studiare e toccare con mano i vari aspetti di una Super App, ovvero come andare ad aprire Mini App, del tutto indipendenti e scollegate dalla Super App, come avviene la comunicazione tra la Super App e le sue Mini App; senza tralasciare le limitazioni e difficoltà di tale approccio.

4.2 Obiettivi

Pertanto, ci siamo concentrati soprattutto su aspetti tecnici delle Super App, partendo dal *come importare una Mini App*, come avviene il passaggio di dati sia in una direzione che nell'altra, fino a implementare un sistema di login automatico, in quanto volevamo che il tutto fosse il più user-friendly possibile.

Infatti, quando si entra in un'app ed effettua l'operazione di login, se tale applicazione mette a disposizione diversi servizi, indipendentemente dal fatto che tali servizi siano esterni o meno all'app stessa, l'utente finale si aspetta di non dover inserire le credenziali ogni volta che deve accedere a un servizio.

4.3 Modello Architetturale e Tecnologie

Come detto precedentemente, per la realizzazione della Super App, abbiamo optato per le *Webviews*, fondamentalmente un motore di rendering di un browser caricato all'interno dell'app, che permettesse appunto di replicare il meccanismo di apertura di una Mini App all'interno dell'applicazione stessa, prerogativa imprescindibile delle Super App.

Partendo, come detto prima, la Super App risulta scritta in Flutter (Dart) e grazie alle Webviews, permette di inglobare al suo interno Mini App di qualsiasi tipo, quindi sia web app che app native, senza limitazioni. Tuttavia, come vedremo, la scelta della tecnologia della Mini App può influire sul risultato finale e le difficoltà d'implementazione.

Le Webviews, pertanto, permettono di aprire Mini App all'interno dell'app stessa e lo fanno grazie all'URL pubblico delle Mini App. Pertanto, questa è una prima limitazione: le Mini App, per poter essere accedute dalla Super App, devono 'vivere' nel web ed avere un URL pubblico.

Per utilizzare la Webview, impieghiamo il pacchetto `Webview_flutter` per utilizzare il `WebviewWidget`, che mostrerà i contenuti del sito web ospitato sull'URL fornito. Per aggiungere il widget Webview, dobbiamo prima istanziare un `WebviewController` che sarà poi passato al `WebviewWidget`. Questo controller è molto importante per la Webview poiché lo utilizzeremo per caricare l'URL, ascoltare gli eventi, gestire la navigazione e, in sostanza, per qualsiasi operazione vogliamo compiere con il `WebviewWidget`. [49]

La Super App comunica con le Mini App tramite un ponte *Js-Native*, che permette dal lato Mini App di lasciare messaggi su tale canale e lato Super App riceverli ed agire di conseguenza. Prende il nome di *Js-Native Bridge* perché tale oggetto vive nel contesto globale dell'applicazione e pertanto è visibile sia alla Super App che alla Mini App tramite codice JavaScript.

La nostra architettura risultante sarà pertanto come mostrato nella Figura 4.1.

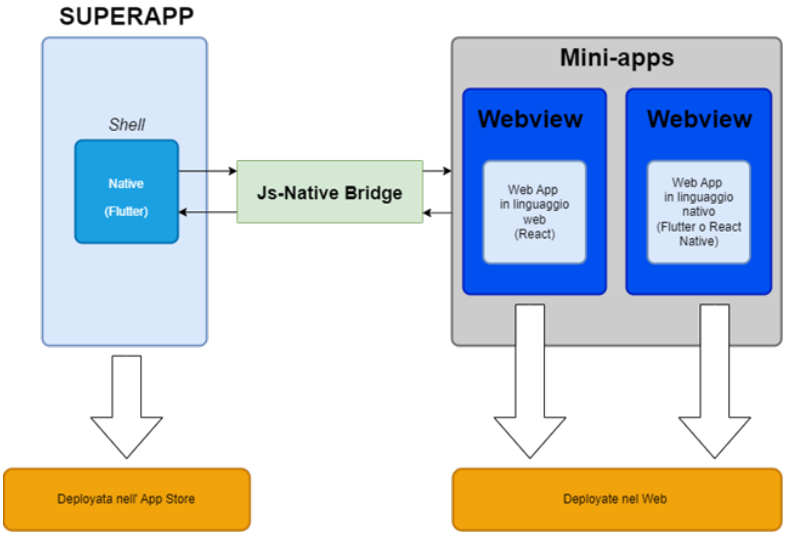

Figura 4.1: Architettura Super App

4.4 Implementazione

4.4.1 Mini App Store

Avviato il progetto, siamo dunque partiti dall'implementazione del setup iniziale: ovvero, abbiamo realizzato, nell'app di e-commerce, una sezione apposita ad ospitare le Mini App, cercando di mantenere uno stile grafico coerente con il resto dell'applicazione, inserendo due web app e un'app nativa in Flutter.

Le due web app non sono altro che i siti web di Flutter e React Native e servivano solo per testare il collegamento tra la Super App e la Webview che ospitava il sito. La Mini App in Flutter è stata appositamente realizzata con lo scopo di testare le funzionalità offerte dalle Webviews ed esplorare le possibili comunicazioni tra Super App e Mini App. Essendo quindi un'app di testing, non ci siamo spesi molto in fattore estetico.

Come detto in precedenza, le Mini App vengono caricate a partire dall'URL pubblico. Nel nostro caso, la Mini App, non essendo pubblica ma realizzata solo con lo scopo di test, veniva deployata in locale, agli indirizzi `localhost:numeroPorta`; ciò la rendeva ovviamente inaccessibile all'esterno e alla Super App, che necessitava di un indirizzo pubblico per funzionare correttamente.

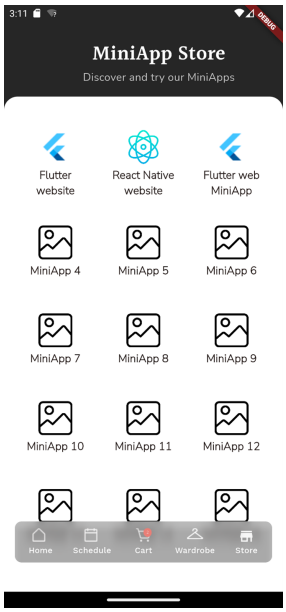

Figura 4.2: Mini App Store

Per ovviare a questo problema, ci siamo serviti di *Ngrok*.

Ngrok

Ngrok è uno strumento di tunneling che consente di esporre server locali a Internet in modo sicuro e semplice. Utilizzando Ngrok, è possibile creare un tunnel dal proprio server locale (ad esempio, in esecuzione su `localhost:8000`) a un indirizzo pubblico temporaneo fornito da Ngrok. Questo è particolarmente utile nello sviluppo e nel debugging di applicazioni web, poiché permette di condividere facilmente l'accesso a un server in esecuzione localmente senza dover configurare manualmente firewall o modifiche al DNS.

Nel nostro caso, Ngrok è stato essenziale per rendere la Mini App accessibile alla Super App, fornendo un URL pubblico che la Super App può utilizzare per interagire con la Mini App. Questo ha facilitato l'integrazione e il testing tra i due componenti, permettendo una comunicazione fluida e immediata tra l'ambiente di sviluppo locale e le risorse accessibili su Internet.

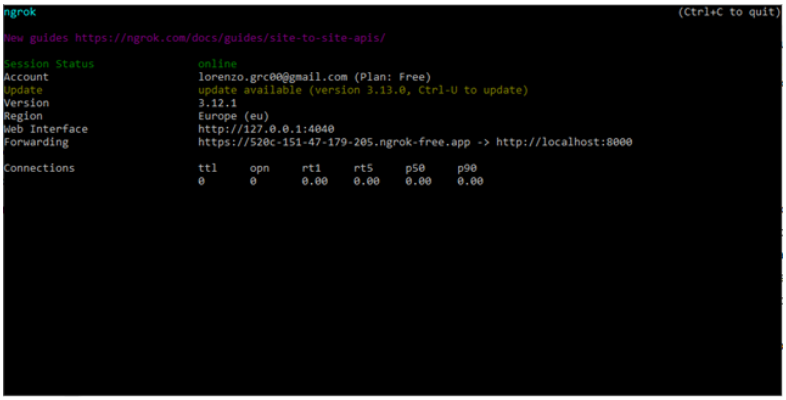

Figura 4.3: Ngrok

4.4.2 Mini App

Dato gli obiettivi che ci siamo posti, abbiamo realizzato la Mini App di conseguenza. Tale Mini App prevede un form di login basato su email e password e una sezione profilo che contiene tale email ricavata dall'account loggato e qualche bottone usato per interagire con la Super App.

Per quanto riguarda il sistema di autenticazione, ci siamo affidati a Firebase.

4.4.3 Login automatica

Come già detto, quando un utente loggato entra in una Mini App, avendo già in precedenza effettuato il login, si aspetta che la sua autenticazione sia valida per tutti i servizi offerti dalla Super App. Essendo la Mini App un'app scollegata e

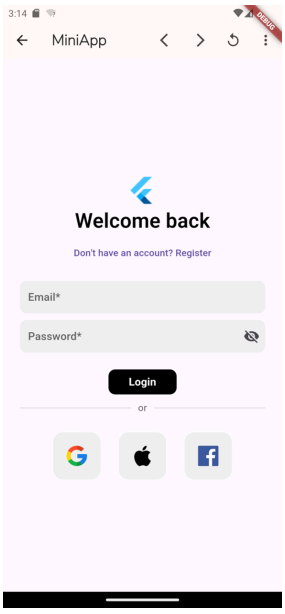

Figura 4.4: Login Mini App

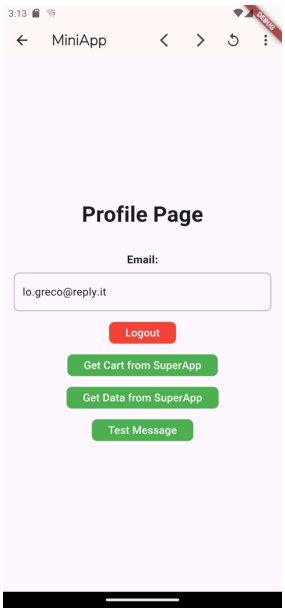

Figura 4.5: Mini App

indipendente rispetto alla Super App, abbiamo riprodotto questo comportamento tramite un meccanismo di login automatico.

All'avvio della Mini App, la Super App si premura di condividere le informazioni per effettuare il login nella Mini App, rendendo disponibile già da subito la pagina profilo all'utente che ha appena effettuato l'accesso. [50]

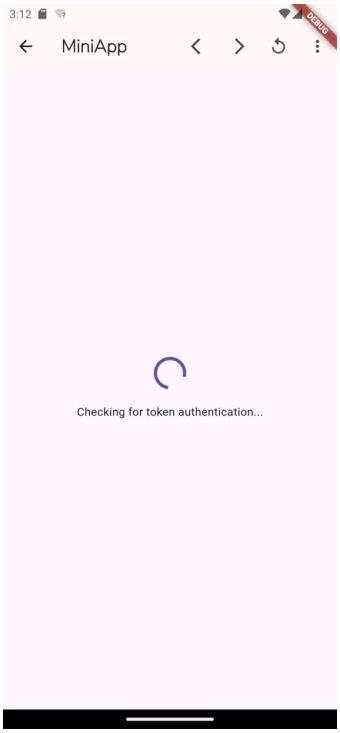

Figura 4.6: Login automatica

Per passare tali informazioni, nel nostro caso email e password, la Super App le inserisce come parametri, opportunamente codificati, in coda all'URL pubblico della Mini App, che si premura dall'altro lato di riceverli e usarli per effettuare il login.

```
1 class MiniappStore extends StatelessWidget {
2   final PageController pageController;
3   MiniappStore(this.pageController);
4
5   void _openMiniApp(BuildContext context, String url, String name)
      async {
6     SharedPreferences prefs = await SharedPreferences.getInstance
    ();
7     String? email = prefs.getString('email');
8     String? password = prefs.getString('password');
```

```
9       print('email: $email, password: $password');
10
11      if (email != null && password != null) {
12        String encodedEmail = Uri.encodeQueryComponent(email);
13        String encodedPassword = Uri.encodeQueryComponent(password);
14        String fullUrl = '$url?email=$encodedEmail&password=
      $encodedPassword';
15        Navigator.push(
16          context,
17          MaterialPageRoute(
18            builder: (context) =>
19                MiniAppPage(miniAppUrl: fullUrl, miniAppName: name),
20          ),
21        );
22      } else {
23        // Gestisci il caso in cui email e password non sono
      disponibili
24        print('Failed to get email and password');
25      }
26    }
27  }
```

Le **email** e **password** vengono salvati nelle **Shared Preferences**, usate anche nel contesto dell'*Offline-First*, in fase di login e opportunamente recuperate in fase di apertura della Mini App. All'apertura della Mini App, pertanto, vi è una fase intermedia di inizializzazione della Mini App, in cui essa si recupera le credenziali e tenta l'accesso.

```
1   Future<void> _checkForCredentials() async {
2       print('Checking for credentials');
3
4       final Uri? uri = Uri.base;
5       final String? encodedEmail = uri?.queryParameters['email'];
6       final String? encodedPassword = uri?.queryParameters['password
      '];
7
8       await Future.delayed(
9           const Duration(seconds: 3)); // Simulazione di una
      richiesta asincrona
10
11      if (encodedEmail != null && encodedPassword != null) {
12        // Prova ad autenticarsi con email e password
13        String email = Uri.decodeQueryComponent(encodedEmail);
14        String password = Uri.decodeQueryComponent(encodedPassword);
15
16        try {
17          await _auth.signInWithEmailAndPassword(
```

```
18              email: email, password: password);
19
20          setState(() {
21            _isLoading = false;
22          });
23        } catch (e) {
24          print('Authentication error: $e');
25
26          // Se l'autenticazione fallisce, crea un nuovo utente e
    autenticati
27          try {
28            await _auth.createUserWithEmailAndPassword(
29                email: email, password: password);
30            await _auth.signInWithEmailAndPassword(
31                email: email, password: password);
32
33            setState(() {
34              _isLoading = false;
35            });
36          } catch (e) {
37            print('Failed to create user and sign in: $e');
38
39            setState(() {
40              _isLoading = false;
41            });
42          }
43        }
44      } else {
45        setState(() {
46          _isLoading = false;
47        });
48      }
49  }
```

Se qualcosa va storto durante il recupero delle credenziali, allora l'utente verrà reindirizzato verso la pagina di login. Un'altra cosa che può succedere è che, al primo avvio della Mini App, l'utente non risulta registrato nel database di **Firebase** della Mini App e pertanto l'autenticazione mediante **email** e **password** fallisce. Per ovviare a questo problema, vi è un meccanismo che, se l'utente non risulta ancora registrato, lo registra automaticamente e lo logga, come se lo fosse sempre stato. Similmente al caso precedente, se qualcosa va storto durante questa fase, l'utente verrà reindirizzato verso la pagina di login. Pertanto, finché non termina questa fase, l'utente vedrà un `CircularProgressIndicator` che lo informa di tale fase transitoria, al termine della quale, se l'autenticazione è avvenuta con successo, verrà reindirizzato alla pagina del profilo, altrimenti alla pagina di login.

4.4.4 Js-Channel

Il canale `Js`, come detto precedentemente, è un oggetto che vive nel contesto globale dell'app, accedibile pertanto sia dalla Super App che dalla Mini App tramite codice `javascript`. Grazie a questa sua caratteristica, permette di ascoltare messaggi provenienti dalla Mini App per poi rispondere ed agire di conseguenza. Viene instanziato dal `controller` della `Webview`, un oggetto che si occupa di gestire tutte le funzionalità delle Webviews nella Super App, quindi aprire la Mini App, fornire un supporto alla navigazione interna e instanziare appunto uno o più canali `Js`. [51]

```
1  WebViewController createWebViewController(
2    BuildContext context,
3    String url,
4    ValueChanged<int> onLoadingPercentageChanged,
5  ) {
6    final controller = WebViewController();
7
8    void handleMessage(String message) async {
9      if (message == "getCart") {
10       SharedPreferences prefs = await SharedPreferences.
     getInstance();
11       String? cartJson = prefs.getString('cachedCart');
12
13       if (cartJson != null) {
14         Map<String, dynamic> cart = jsonDecode(cartJson);
15         print(cart);
16
17         String totalPrice = (cart['totalPrice']['centAmount'] /
     100)
18             .toStringAsFixed(2)
19             .replaceAll('.', ',');
20         String currencyCode = cart['totalPrice']['currencyCode'];
21
22         List<dynamic> lineItems = cart['lineItems'];
23
24         String formattedCart = '''Total Price: $totalPrice
     $currencyCode''';
25
26         if (lineItems.isNotEmpty) {
27           formattedCart += "\nLine Items:\n";
28           for (var item in lineItems) {
29             String productName = item['name']['en'];
30             int quantity = item['quantity'];
31             String price = (item['price']['value']['centAmount'] /
      100)
32                 .toStringAsFixed(2)
```

```
33                  .replaceAll('.', ',');
34              formattedCart +=
35                  "Product: $productName, Quantity: $quantity, Price
    : $price\euro\n";
36            }
37          } else {
38            formattedCart += "\nCart is empty\n";
39          }
40
41          // Mostra il carrello all'utente
42          [...]
43        } else {
44          // Il carrello è vuoto
45          [...]
46        }
47      } else if (message == "getData") {
48        await controller.runJavaScript('onMessageReceived("Hello
    from SuperApp")');
49      }
50    }
51
52    controller
53      ..setNavigationDelegate(NavigationDelegate(
54        onPageStarted: (url) => onLoadingPercentageChanged(0),
55        onProgress: (progress) => onLoadingPercentageChanged(
    progress),
56        onPageFinished: (url) => onLoadingPercentageChanged(100),
57        onNavigationRequest: (navigation) => NavigationDecision.
    navigate,
58      ))
59      ..setJavaScriptMode(JavaScriptMode.unrestricted)
60      ..addJavaScriptChannel(
61        'JsChannel',
62        onMessageReceived: (message) {
63          handleMessage(message.message);
64        },
65      )
66      ..loadRequest(Uri.parse(url));
67
68    return controller;
69  }
```

4.4.5 Comunicazione tra Super App e Mini App

La comunicazione tra **Super App** e **Mini App** si basa pertanto su tale canale, che offre il metodo `onMessageReceived` per rispondere ad eventuali messaggi depositati sul canale dalla Mini App, tramite il metodo `NomeCanale.postMessage(message)`.

Possiamo distinguere due tipi di comunicazione: una basata sull'interazione dell'utente con elementi a schermo, che sia una **appbar**, un **FAB**, gestiti dalla Super App, e una comunicazione invece scaturita dall'interazione dell'utente con elementi della Mini App stessa. Nel primo caso, la comunicazione risulta più semplice, in quanto essendo elementi controllati dalla Super App, hanno accesso più facilmente al canale per lasciare messaggi.

Nel nostro caso, alla ricezione di tali messaggi, il canale, tramite `handleMessage`, che differenzia opportunamente le azioni da eseguire in base al messaggio ricevuto, viene mostrato a schermo tramite un `alert` il carrello corrente dell'utente.

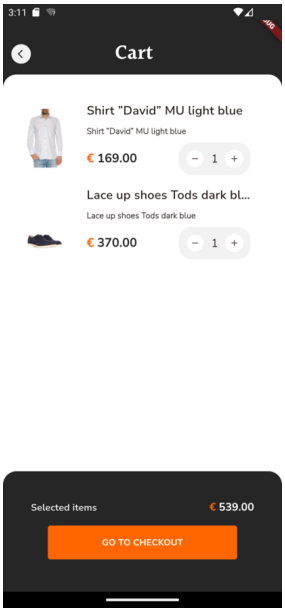

Figura 4.7: Carrello Super App

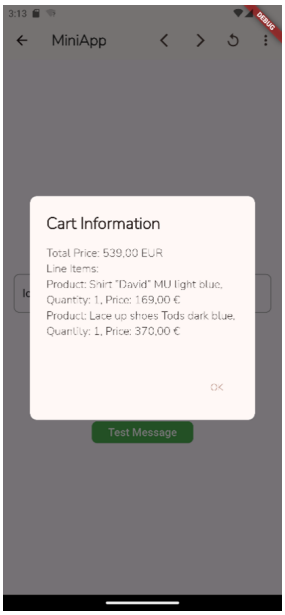

Figura 4.8: Carrello ricevuto nella Mini App

Nel secondo caso, la comunicazione tramite un'interazione diretta con la **Mini App**, il tutto risulta un po' più complesso, in quanto la Mini App non ha accesso diretto alle variabili e strutture della **Super App**. Ci siamo serviti pertanto di una libreria esterna, `JS`, importata tramite `import 'package:js/js.dart';`.

package:js

La libreria `package:js` di Dart è una potente soluzione per l'interoperabilità tra Dart e JavaScript, offrendo una serie di funzionalità che permettono a questi due mondi di comunicare senza soluzione di continuità. Questo è particolarmente utile

nello sviluppo di applicazioni web moderne, dove spesso è necessario integrare librerie JavaScript.

Interoperabilità (Interop): L'interoperabilità è al cuore della libreria `package:js`. Essa consente al codice Dart di chiamare funzioni JavaScript e viceversa. Questa capacità è cruciale quando si sviluppano applicazioni web complesse che richiedono l'integrazione di diverse tecnologie e librerie.

Annotazioni @JS: Le annotazioni `@JS` sono strumenti fondamentali per facilitare l'interazione tra Dart e JavaScript. Ecco alcune delle principali annotazioni:

- `@JS`: Utilizzata per esporre funzioni, oggetti e classi Dart al contesto JavaScript, permettendo così al codice JavaScript di invocare direttamente funzioni Dart.
- `@anonymous`: Questa annotazione viene utilizzata per creare classi anonime JavaScript direttamente in Dart. È utile per rappresentare oggetti JavaScript che non hanno una struttura rigida.
- `@staticInterop`: Consente di interagire con membri statici di classi JavaScript, facilitando l'accesso a costanti e metodi statici.

Esportazione di Funzioni Dart a JavaScript: Le funzioni Dart possono essere esposte al contesto JavaScript utilizzando funzioni come `allowInterop` o `allowInteropCaptureThis`. Questo permette di definire funzioni Dart che possono essere chiamate direttamente dal codice JavaScript.

Importazione di Funzioni JavaScript in Dart: Analogamente, le funzioni JavaScript possono essere importate e invocate nel codice Dart. Utilizzando le annotazioni `@JS`, è possibile dichiarare funzioni esterne in Dart che mappano direttamente alle funzioni JavaScript.

Nel nostro caso, ciò è stato estremamente utile per scrivere del codice JavaScript, che avesse quindi accesso al contesto delle variabili globali, tra cui appunto il canale JS della Super App, e lasciare quindi dei messaggi in base all'interazione dell'utente con la Mini App stessa.

```
1 <script>
2     function sendMessageToFlutter(message) {
3       if (window.JsChannel) {
4         window.JsChannel.postMessage(message);
5       } else {
6         console.error('JsChannel is not available.');
7       }
```

```
8       }
9
10      function onMessageReceived(message) {
11        console.log('Message Received from SuperApp: ', message);
12        if (window.updateMessageReceived && typeof window.
      updateMessageReceived === 'function') {
13          window.updateMessageReceived(message);
14        } else {
15          console.error('Dart function is not available.');
16        }
17      }
18  </script>
```

Il meccanismo di per sé funziona così: all'input dell'utente, tramite un bottone della Mini App, viene richiamata una funzione che a sua volta invoca la funzione `sendMessageToFlutter` dello script JS. Questa funzione, avendo accesso al canale JS, è in grado di lasciare messaggi che verranno poi ricevuti dalla `handleMessage` del controller delle Webviews.

```
1   class _ProfilePageState extends State<ProfilePage> {
2     final User? user = FirebaseAuth.instance.currentUser;
3
4     String? messageReceived;
5
6     Future<void> signOut() async {
7       await FirebaseAuth.instance.signOut();
8     }
9
10    void sendMessageToSuperApp() {
11      sendMessageToFlutter('getCart');
12    }
13
14    void getDataFromSuperApp() {
15      sendMessageToFlutter('getData');
16    }
17
18    void updateMessageReceived(message) {
19      setState(() {
20        messageReceived = message;
21      });
22    }
23
24    @override
25    void initState() {
26      super.initState();
27      // Espone la funzione Dart al contesto JavaScript
```

```
28      updateMessageReceivedFunction = allowInterop(
        updateMessageReceived);
29    }
30  }
```

```
1  // Definisce una funzione esterna per comunicare con Flutter
2  @JS('sendMessageToFlutter')
3  external void sendMessageToFlutter(String message);
4
5  // Utilizza @JS() per esporre la funzione al contesto globale
6  @JS('updateMessageReceived')
7  external set updateMessageReceivedFunction(Function(String) f);
```

`onMessageReceived`, come vedremo più avanti, implementa la comunicazione nell'altro senso, ovvero viene invocata dalla Super App, sempre dalla `handleMessage`, quindi in seguito a un'interazione dell'utente con la Mini App, per scatenare un evento nella Mini App passando al testo stesso dei dati.

4.4.6 Inizializzazione asincrona delle variabili

Come ultima cosa, abbiamo deciso di implementare un meccanismo di inizializzazione delle variabili in modo asincrono.

Quando un utente naviga all'interno della Mini App, può esserci l'esigenza di ottenere dei dati dalla Super App, come nel caso precedente, ma anche di ottenere dei dati in base ad alcune variabili o filtri impostati tramite la Mini App, che però abbiano un effetto diretto sui risultati da ottenere dalla Super App stessa.

Per fare ciò è necessario pertanto implementare un meccanismo che permetta, in seguito a un'interazione con la Mini App, di ricevere dei dati dalla Super App come nel caso precedente, con l'aggiunta però della possibilità di riutilizzare tali dati in futuro e pertanto salvarli nel contesto della Mini App. Il tutto si può riassumere pertanto con l'inizializzazione di variabili locali della Mini App in modo asincrono con dati provenienti dalla Super App.

Per fare ciò bisogna espandere il meccanismo del canale `js` precedentemente utilizzato con la possibilità di mandare alla Mini App dei dati effettivi e non limitarsi a mostrarli a schermo.

Quindi il meccanismo comincia con la solita interazione dell'utente con un bottone della Mini App, che richiama una funzione che a sua volta richiama la funzione esterna dello script `js`, la quale deposita un messaggio nel canale `js`.

La Super App si occuperà invece di ricevere tale messaggio ed elaborarlo, ovvero eseguire anch'essa del codice `javascript` che richiami la funzione dello script `js` della Mini App `onMessageReceived`, passando opportunamente i dati interessati.

Questa funzione richiama una funzione interna della Mini App che viene esposta al contesto `javascript` grazie all'ausilio della libreria `js`.

La funzione interna della Mini App si occuperà pertanto di aggiornare localmente lo stato di una variabile con i dati ricevuti dalla Super App.

Nel nostro caso, passiamo solamente una stringa che, salvata localmente e aggiornando lo stato della variabile locale, aggiorna il contesto della Mini App e quindi viene mostrata a schermo.

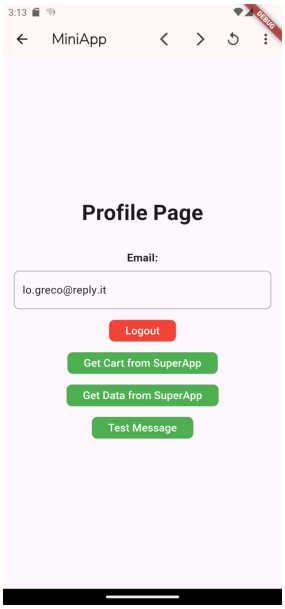

Figura 4.9: Inizializzazione delle variabili in modo asincrono 1

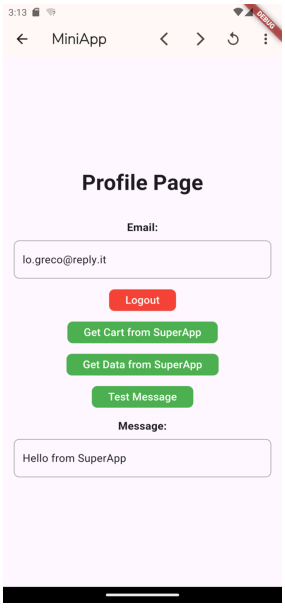

Figura 4.10: Inizializzazione delle variabili in modo asincrono 2

Capitolo 5

Implementazione Offline-First

5.1 Introduzione

Il passaggio all'approccio **Offline-First** rappresenta un'evoluzione strategica nell'ambito dello sviluppo della Super App, offrendo un'esperienza utente più resiliente e affidabile.

Sebbene l'implementazione di questa funzionalità sia stata condotta come un progetto separato su un'applicazione preesistente, è utile presentarla come una naturale estensione delle caratteristiche della Super App.

Questo consente di enfatizzare il valore aggiunto di un sistema capace di funzionare senza interruzioni anche in condizioni di connettività limitata, rafforzando la percezione di solidità e completezza del prodotto complessivo.

Nel presente capitolo, verranno illustrati i risultati ottenuti da questa implementazione, analizzati in relazione agli obiettivi prefissati e discussi nel contesto delle *best practice* per applicazioni mobile moderne.

5.2 Sfide e criticità del progetto Offline-First

Per garantire un'implementazione efficace dell'applicazione, è stato necessario definire una serie di **requisiti funzionali**, sia per la modalità online che offline.

Questi requisiti emergono dall'analisi delle esigenze dell'utente finale e dalle caratteristiche proprie di un sistema *Offline-First*, che deve assicurare continuità d'uso anche in assenza di connettività.

In particolare, l'attenzione si è concentrata sulla **persistenza dei dati locali**, sulla **gestione dei conflitti** al ripristino della connessione e sulle strategie per garantire un'esperienza utente fluida e coerente.

Di seguito sono elencati i principali requisiti funzionali dell'applicazione:

- **Navigazione e acquisto online:** L'utente deve poter navigare tra i prodotti, visualizzarne i dettagli, aggiungerli al carrello ed effettuare il pagamento. Deve inoltre poter accedere al proprio profilo, monitorare lo stato degli ordini e usufruire di altre funzionalità tipiche di un'app di e-commerce.
- **Supporto alla modalità offline:** L'app deve consentire all'utente di accedere a dati precedentemente caricati, come la lista dei prodotti visualizzati, il proprio profilo e il carrello. Inoltre, deve essere possibile aggiungere prodotti al carrello anche senza connessione.
- **Persistenza dei dati locali:** Le informazioni essenziali per il funzionamento offline devono essere salvate in locale. Questo include i dettagli dei prodotti, il contenuto del carrello, le informazioni utente e lo stato dell'ordine. Deve essere prevista una cache per le chiamate API, in modo da ridurre le richieste non necessarie quando la connessione è disponibile.
- **Gestione dei conflitti alla riconnessione:** Quando l'app torna online, i dati locali devono essere sincronizzati con il server, risolvendo eventuali conflitti. Ad esempio, se un prodotto è stato rimosso dal catalogo mentre l'utente era offline, il sistema deve notificare il problema e aggiornare il carrello di conseguenza.
- **Strategie di risoluzione dei conflitti:** I conflitti possono essere gestiti in modi diversi a seconda del contesto. Alcuni dati possono essere sovrascritti automaticamente con quelli più recenti dal server, mentre per altri potrebbe essere necessaria un'interazione con l'utente.
- **Gestione degli errori e delle notifiche:** L'app deve fornire all'utente un feedback chiaro sulle operazioni offline e sulla loro sincronizzazione. In caso di successo, deve mostrare un messaggio di conferma; in caso di errore, deve indicare il problema e suggerire possibili azioni correttive.

5.3 Struttura e funzionalità dell'applicazione con supporto Offline-First

Struttura dell'applicazione:

- Homepage con il catalogo

- Pagina del dettaglio prodotto con possibilità di aggiungere al carrello
- Area utente con informazioni personali e storico ordini e lista preferiti
- Carrello per fare gli acquisti

Offline-First features:

- Poter sfogliare il catalogo e vedere i dettagli dei prodotti precedentemente scaricati
- Poter aggiungere/rimuovere prodotti dai preferiti
- Poter visionare l'area utente e modificare le informazioni personali
- Poter aggiungere prodotti al carrello

Conflitti:

- Prodotto aggiunto al carrello che non è più disponibile

5.3.1 Descrizione dell'applicazione

L'applicazione fornitami è una demo di un'app di e-commerce sviluppata utilizzando il framework **Flutter** e integrando diverse librerie esterne. Il backend dell'applicazione è gestito tramite **Firebase**,una piattaforma abbastanza completa messa a punto e offerta da Google per lo sviluppo e la realizzazione di app web e mobile.

L'applicazione presenta una **homepage** dal design moderno e pulito, caratterizzato da una combinazione di elementi grafici e testuali ben bilanciati. L'uso di una **palette di colori scuri**, principalmente neri e grigi, conferisce un aspetto elegante e professionale. Le immagini di alta qualità e le icone chiare aggiungono interesse visivo e facilitano la navigazione dell'utente attraverso le varie sezioni dell'app. Gli elementi sono disposti in modo intuitivo, con pulsanti e icone chiaramente evidenziati per indicare le azioni disponibili per l'utente.

Le schermate scrollabili rappresentano le diverse **collezioni** dell'applicazione, come la '*Collezione estiva*', '*Collezione autunnale*' e '*Collezione invernale*'. Per accedere al catalogo di queste collezioni, l'utente può fare clic su un pulsante posto in alto a sinistra. Un pulsante al centro della schermata fornisce una breve descrizione della collezione e la possibilità di visualizzare alcuni prodotti. Attualmente, solo la *collezione estiva* è stata implementata, quindi tutti i pulsanti portano a questa collezione.

Oltre ai pulsanti per accedere al catalogo, nella **homepage** è presente un'icona posta in alto a destra che reindirizza verso il **profilo utente**. Una **bottom bar**

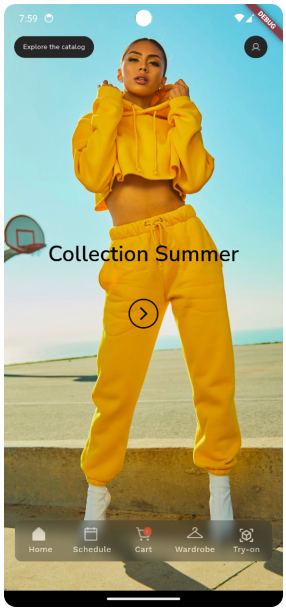

Figura 5.1: Homepage

permette agli utenti di passare rapidamente tra le varie sezioni dell'applicazione, come '*Home*', '*Schedule*', '*Cart*', '*Wardrobe*' e '*Try-on*'.

Nel catalogo, le diverse categorie di prodotti, come '*Uomo*', '*Donna*' e '*Accessori*', presentano una lista di prodotti tra cui l'utente può scegliere per visualizzarne i dettagli.

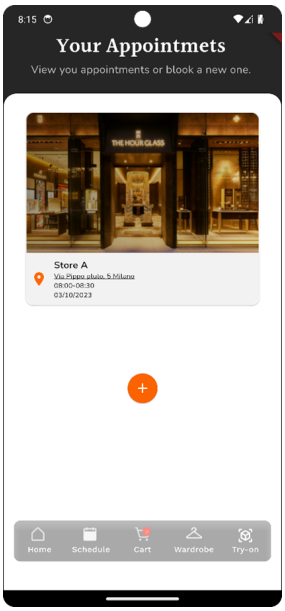

Figura 5.2: Sezione *Schedule*

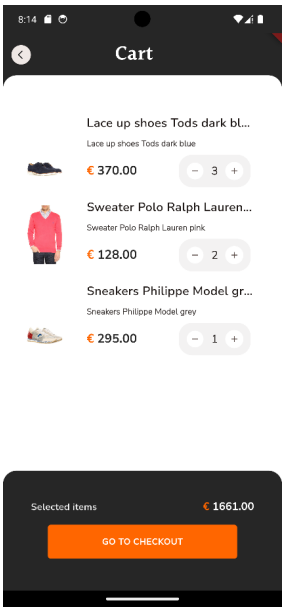

Figura 5.3: Sezione *Cart*

5.4 Implementazione

5.4.1 Monitoraggio della connessione

Dopo aver definito la struttura dell'applicazione, ho iniziato il mio lavoro focalizzandomi sull'implementazione dell'approccio **Offline-First** nelle schermate accessibili dalla tab *home* della *bottom bar*, che comprendono tutte le sezioni relative al catalogo e alla sua esplorazione.

Per prima cosa, ho sviluppato un meccanismo per monitorare la connettività dell'applicazione, consentendo di rilevare i cambiamenti nella connessione. Questo sistema è in grado di individuare passaggi dalla connessione dati al Wi-Fi, nonché la perdita e il ripristino della connessione.

Nel dettaglio, il codice che ho implementato fa largo uso della libreria `get` per l'iniezione delle dipendenze e la gestione dello stato dell'applicazione. La

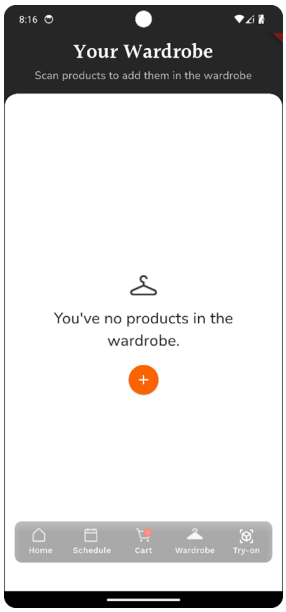

Figura 5.4: Sezione *Wardrobe*

Figura 5.5: Sezione *Try on*

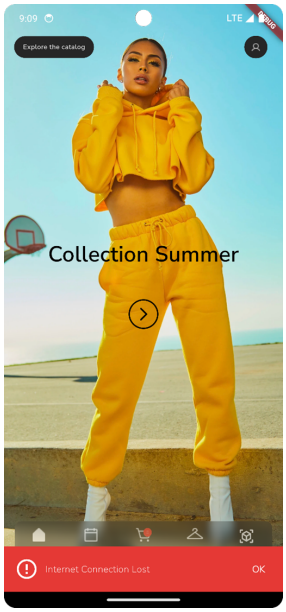

Figura 5.6: Connessione persa

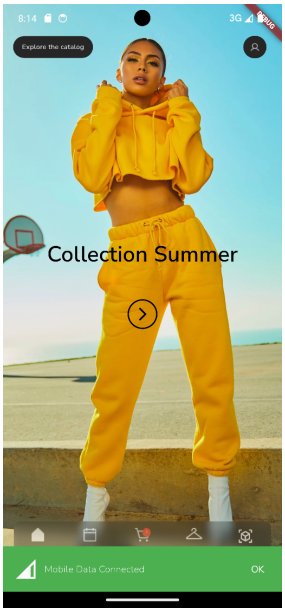

Figura 5.7: Collegato tramite dati mobili

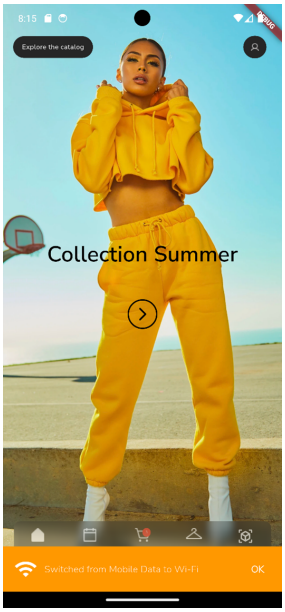

Figura 5.8: Passaggio da connessione Wi-Fi a dati mobili

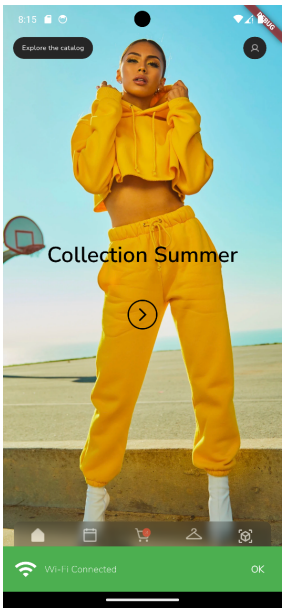

Figura 5.9: Collegato tramite Wi-Fi

funzione `main()` si occupa di inizializzare l'applicazione Flutter, configurando `Firebase` e avviando l'applicazione con il widget principale `MyApp()`. Inoltre, viene eseguita l'inizializzazione delle dipendenze dell'applicazione attraverso la classe `DependencyInjection`, la quale definisce un metodo `init()` per inizializzare il controller di rete e integrarlo nell'`InheritedWidget` di `GetX`.

Il controller di rete (`NetworkController`) sfrutta la potenza della libreria `GetX` per monitorare lo stato della connessione. Questo controller utilizza la classe `Connectivity` della libreria `connectivity_plus` per tenere traccia dello stato della connessione di rete. La classe `Connectivity` offre vari metodi per rilevare se il dispositivo è connesso a Internet e per ascoltare gli eventi di cambiamento di connessione.

Quando il controller di rete rileva una modifica nello stato della connessione di rete, esso aggiorna lo stato dell'applicazione in modo appropriato tramite `GetX`. Questo stato può essere utilizzato in qualsiasi punto dell'applicazione per regolare la visibilità di specifiche funzionalità o fornire feedback agli utenti in base alla loro connessione di rete.

```
1 import 'package:get/get.dart';
2 import 'controller_network/network_controller.dart';
3
4 class DependencyInjection {
5   static void init() {
6     Get.put<NetworkController>(NetworkController(), permanent:
    true);
7   }
8 }
```

```
1 void main() async {
2   WidgetsFlutterBinding.ensureInitialized();
3   await Firebase.initializeApp(options: DefaultFirebaseOptions.
    currentPlatform);
4   setupNotification();
5   cleanCachedPreferences();
6   runApp(MyApp());
7   DependencyInjection.init();
8 }
```

```
1 class NetworkController extends GetxController {
2   final Connectivity _connectivity = Connectivity();
3
4   @override
```

```
5   void onInit() {
6     super.onInit();
7     _connectivity.onConnectivityChanged.listen(
     _updateConnectionStatus);
8
9   static Future<bool> isOnline() async {
10    var connectivityResult = await Connectivity().
     checkConnectivity();
11    return connectivityResult != ConnectivityResult.none;
12  }
13 }
```

5.4.2 Gestione del Catalogo

Dopo aver completato l'implementazione del monitoraggio della connettività, ho proseguito con lo sviluppo effettivo dell'**Offline-First** nelle pagine relative al catalogo dell'applicazione.

L'idea fondamentale dietro l'approccio **Offline-First** è quella di consentire agli utenti di accedere alle pagine già visualizzate anche in assenza di connessione internet. Per raggiungere questo obiettivo, esistono diverse tecniche supportate da varie librerie, ma ho scelto quella che mi è sembrata più veloce, efficiente e adatta al mio caso, che non richiede l'aggiunta di database locali. Per implementare questa soluzione, ho utilizzato le `Shared Preferences`.

Ho implementato un sistema che sfrutta le `Shared Preferences` per memorizzare localmente le risposte alle chiamate API, consentendo all'applicazione di utilizzarle quando l'utente non è connesso a Internet. Prima di effettuare una chiamata API, il sistema verifica la connettività dell'utente: se è online, l'applicazione procede normalmente, salvando i dati ottenuti dal server tramite `Shared Preferences` per un accesso futuro. Se l'utente è offline, il sistema controlla se ci sono dati salvati localmente relativi alla pagina richiesta: in caso affermativo, li utilizza per fornire una risposta immediata all'utente; altrimenti, mostra un messaggio di avviso e invita l'utente a controllare la connessione.

Questo approccio migliora le prestazioni dell'applicazione in assenza di connessione Internet, riducendo i tempi di attesa e migliorando l'esperienza utente complessiva. Grazie a questa implementazione, l'**Offline-First** è stato integrato con successo in tutte le pagine relative al catalogo e alla home dell'applicazione.

```
1 Future<List<Categories>?> setup() async {
2     try {
3       print("Chiamo il setup");
4       isOnline = await NetworkController.isOnline();
5       if (isOnline) {
```

```
6           print("Sono online");
7           return await getCategory();
8         } else {
9           print("Sono offline");
10          return await loadLocalData();
11        }
12      } catch (e) {
13        print('error');
14      }
15    }
```

```
1  Future<List<Categories>?> getCategory() async {
2      final pref = await SharedPreferences.getInstance();
3      String accessToken = pref.getString("accessToken") ?? "";
4      print("Chiamo getCategory");
5      try {
6        final response = await http.get(
7          Uri.parse("$commerceToolApiBaseURL/categories?limit=20&
       offset=0"),
8          headers: {
9            'authorization': 'Bearer $accessToken',
10         },
11       );
12       final responseData = await jsonDecode(response.body);
13       final categoriesList = CategoriesList.fromJson(responseData)
       ;
14       categoryList = categoriesList.results ?? [];
15       categoryMap =
16           categoryList.groupListsBy((element) => element.name?.en
       ?? '');
17       categoryFilteredMap.addAll(categoryMap);
18       print("categoryFilteredMap length: ${categoryFilteredMap.
       length}");
19
20       // Salva i dati localmente
21       print("Aggiorno i dati salvati localmente: Categories");
22       await LocalStorage.saveData('cachedCategories', response.
       body);
23
24       return categoriesList.results;
25     } catch (e) {
26       print("error $e ");
27     }
28     return null;
29 }
```

```
1  Future<List<Categories>?> loadLocalData() async {
2      final pref = await SharedPreferences.getInstance();
3      String accessToken = pref.getString("accessToken") ?? "";
4      print("Chiamo loadData");
5      try {
6        final cachedData = await LocalStorage.loadData('
       cachedCategories');
7        if (cachedData != null) {
8          final responseData = jsonDecode(cachedData);
9          final categoriesList = CategoriesList.fromJson(
       responseData);
10           categoryList = categoriesList.results ?? [];
11           categoryMap =
12               categoryList.groupListsBy((element) => element.name?.
       en ?? '');
13           categoryFilteredMap.addAll(categoryMap);
14           print("categoryFilteredMap length: ${categoryFilteredMap.
       length}");
15
16           return categoriesList.results;
17         } else {
18           print(
19               "Sorry, no content available! You are offline, please
       go back and make sure to connect to the internet!");
20         }
21       } catch (e) {
22         print("error $e ");
23       }
24       return null;
25 }
```

La metodologia **Offline-First** implementata implica la capacità di visualizzare le pagine in modalità offline nel caso in cui siano state precedentemente visitate online. Tuttavia, nel caso in cui questa premessa non sia soddisfatta e si tenti di accedere a una pagina che non è stata precedentemente visitata senza una connessione Internet attiva, verrà visualizzato un messaggio di errore.

A livello di implementazione, indipendentemente dalla connettività, viene mostrata l'icona classica di caricamento finché i dati della pagina non sono disponibili. Tuttavia, in caso di mancanza di connessione Internet e tentativo di accesso a una nuova pagina, i dati non saranno disponibili. Per gestire questa situazione, è stata implementata una logica che consiste in un timer di 3 secondi. Al termine di questo intervallo, se i dati non sono stati recuperati con successo, viene visualizzato un messaggio di errore. Tale messaggio offre la possibilità all'utente di tornare alla pagina precedente.

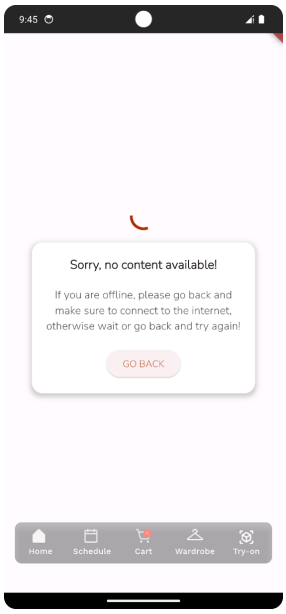

Figura 5.10: Messaggio d'errore

5.4.3 Gestione del Carrello

Dopo essermi occupato della gestione di base delle varie pagine in modalità offline, sono passato ad occuparmi della gestione del carrello in tale contesto, ovvero in assenza di connessione. Tale procedura è stata da me identificata come cruciale e complessa. Essa comprende operazioni quali la visualizzazione e la modifica del carrello.

In particolare la modifica del carrello va distinta in due scenari diversi: incremento e decremento del numero di articoli di un dato prodotto già presente nel carrello e aggiunta di nuovi prodotti, precedentemente non presenti al carrello. Entrambe tali operazioni sono particolarmente delicate in quanto necessitano di una sincronizzazione accurata con il server quando la connessione è resa nuovamente disponibile. Questo aspetto è di cruciale importanza, in quanto è necessario che le eventuali modifiche apportate offline si ritrovino poi in ambiente online, come gli utenti ovviamente si aspettano che avvenga.

Lista delle azioni offline

Per realizzare il meccanismo di sincronizzazione, ho adottato l'utilizzo delle `Shared Preferences`, utilizzando una lista specifica per memorizzare le azioni da eseguire una volta ripristinata la connessione.

Pertanto, ho apportato modifiche alle funzioni esistenti di gestione del carrello per integrare questo meccanismo. In particolare, ogni volta che queste funzioni vengono eseguite offline, aggiungono l'azione corrispondente alla lista delle `Shared Preferences`. Successivamente, l'applicazione continua a mostrare localmente le modifiche apportate.

Dopo aver correttamente aggiunto le azioni offline alla lista delle `Shared Preferences`, è stato necessario implementare un meccanismo per eseguire tali modifiche una volta che il dispositivo torna online, sincronizzando i dati con il server. A tal fine, ho sfruttato il meccanismo esistente per monitorare la connessione.

Quando viene rilevata la ripresa della connessione, oltre a informare l'utente mediante un componente visuale, viene avviata una chiamata alla funzione responsabile di eseguire le richieste API che normalmente l'utente avrebbe effettuato se fosse stato sempre online, garantendo così la sincronizzazione dei dati con il server.

```
1 // Funzione per eseguire le azioni offline quando l'applicazione
     torna online
2 Future<void> executeOfflineActions() async {
3     print("Connessione ripristinata: eseguo azioni offline");
4     final pref = await SharedPreferences.getInstance();
5     List<String> offlineActions = pref.getStringList('
     offlineActions') ?? [];
6
```

```
7       // Itera su ogni azione offline e eseguila
8       for (String offlineAction in offlineActions) {
9         // Decodifica la stringa JSON per ottenere l'azione e l'
        indice
10        Map<String, dynamic> actionData = jsonDecode(offlineAction);
11        String action = actionData['action'];
12        String data = actionData['data'];
13
14        try {
15          // Esegui la chiamata API corrispondente all'azione
16          await executeAction(action, data);
17        } catch (e) {
18          // Gestisci eventuali errori durante l'esecuzione dell'
        azione
19          print('Errore durante l\'esecuzione dell\'azione offline:
        $e');
20        }
21      }
22
23      // Dopo aver eseguito tutte le azioni, svuota la lista delle
        azioni offline
24      await pref.setStringList('offlineActions', []);
25  }
```

```
1   // Funzione per eseguire un'azione (chiamata API) in base al tipo
        di azione
2   Future<void> executeAction(String action, String data) async {
3       CartModel cartModel = CartModel();
4       ProductDetailModel productDetailModel = ProductDetailModel();
5       final pref = await SharedPreferences.getInstance();
6
7       if (action == 'addQuantity') {
8           print("Eseguo AZIONE: $action, con index $data");
9           await cartModel.setup();
10          print("Eseguo addQuantity");
11          await cartModel.addQuantity(int.parse(data));
12      } else if (action == 'removeQuantity') {
13          print("Eseguo AZIONE: $action, con index $data");
14          await cartModel.setup();
15          print("Eseguo removeQuantity");
16          await cartModel.removeQuantity(int.parse(data));
17      } else if (action == 'addToCart') {
18          print("Eseguo AZIONE: $action, con productId $data");
19          // Logica per aggiunta al carrello post-offline
20          pref.setString('selectedProduct', data);
21          await productDetailModel.setup();
22          print("Eseguo addToCart");
```

```
23          await productDetailModel.addToCartPostOffline(data);
24      }
25  }
```

Aggiunta di un prodotto al carrello in assenza di connessione

Una volta implementato il meccanismo di sincronizzazione, è diventato essenziale fornire all'utente un feedback immediato sulle modifiche apportate al carrello anche quando si trova offline. È fondamentale che l'utente percepisca la completezza delle funzionalità anche in assenza di connessione internet. Ad esempio, se modifica la quantità di un prodotto nel carrello, deve vedere tale modifica riflessa nell'interfaccia.

Per ottenere questo risultato, ho modificato le funzioni preesistenti, come `addQuantity`, `removeQuantity` e `addToCart`, affinché possano gestire anche tali operazioni offline. Modificare il carrello offline significa intervenire direttamente sul JSON del carrello salvato localmente nelle `Shared Preferences`. Ho quindi adeguato le funzioni affinché possano agire su questo JSON del carrello locale.

Quando queste funzioni vengono chiamate offline, eseguono le seguenti operazioni in sequenza:

1. Aggiungono l'azione alla lista delle azioni offline.
2. Caricano il carrello salvato localmente.
3. Accedono alle proprietà desiderate del JSON del carrello.
4. Sovrascrivono tali proprietà con le modifiche desiderate.
5. Salvano il nuovo carrello localmente e lo restituiscono alla funzione chiamante, consentendo così di visualizzare immediatamente le modifiche apportate.

```
1  Future<Cart?> addQuantity(int index) async {
2      final pref = await SharedPreferences.getInstance();
3      cartId = pref.getString('cartId');
4      token = pref.getString('accessToken');
5
6      final quantity = (cart?.lineItems?[index].quantity ?? 1) + 1;
7
8      bool isOnline = await NetworkController.isOnline();
9
10     if (isOnline) {
11         [...]
12     } else {
13         print("Sono offline");
```

```
14
15          // Aggiungi l'azione alla lista delle azioni offline
16          await NetworkController.addActionToOfflineList(
17              'addQuantity', index.toString());
18
19          // Carica il Cart localmente
20          print("Carico CachedCart");
21          final cachedData = await LocalStorage.loadData('cachedCart
    ');
22
23          if (cachedData != null) {
24              Map<String, dynamic> json = jsonDecode(cachedData);
25
26              // Accedi alle proprietà desiderate della mappa json
27              var lineItems = json['lineItems'] as List<dynamic>;
28              var lineItem = lineItems[index];
29              var oldQuantity = lineItem['quantity'];
30              var priceItem = lineItem['price']['value']['centAmount
    '];
31              var oldLineItemsTotalPrice = lineItem['totalPrice']['
    centAmount'];
32              var oldTotalPrice = json['totalPrice']['centAmount'];
33
34              // Stampa le informazioni ottenute
35              print("Old quantity: $oldQuantity");
36              print("Price item: $priceItem");
37              print("Old lineItemsTotalPrice:
    $oldLineItemsTotalPrice");
38              print("Old TotalPrice: $oldTotalPrice");
39
40              var newQuantity = quantity;
41              var newLineItemsTotalPrice = oldLineItemsTotalPrice +
    priceItem;
42              var newTotalPrice = oldTotalPrice + priceItem;
43
44              print("New quantity: $newQuantity");
45              print("New lineItemsTotalPrice:
    $newLineItemsTotalPrice");
46              print("New totalPrice: $newTotalPrice");
47
48              json['lineItems'][index]['quantity'] = newQuantity;
49              json['lineItems'][index]['totalPrice']['centAmount'] =
50                  newLineItemsTotalPrice;
51              json['totalPrice']['centAmount'] = newTotalPrice;
52
53              cart = Cart.fromJson(json);
54
55              // Sovrascrivi i dati locali con il nuovo carrello
    JSON
```

```
56              await LocalStorage.saveData('cachedCart', jsonEncode(
    json));
57          }
58
59          notifyListeners();
60          return cart;
61      }
62  }
```

L'ultimo scenario da affrontare riguarda l'aggiunta di un prodotto al carrello, con i due casi distinti precedentemente menzionati. L'azione di aggiunta di un prodotto al carrello viene gestita dalla funzione `addToCartPostOffline`, che a differenza della sua controparte, una volta eseguita l'azione non mostra a schermo un alert di avvenuta aggiunta al carrello, in quanto tale funzione viene richiamata in modo desincronizzato rispetto all'azione in sé di aggiunta al carrello, che è quando l'utente si aspetta di ricevere il feedback dell'alert di avvenuta aggiunta al carrello.

```
1  Future<Cart?> addToCartPostOffline(String productID) async {
2      Cart? cart;
3
4      print("Chiamo addToCartOfflineCart");
5
6      try {
7        if (cartAlreadyCreated) {
8          [...]
9        }
10
11       final pref = await SharedPreferences.getInstance();
12       pref.setInt("cartVersion", cart?.version ?? 1);
13       cartVersion = cart?.version ?? 1;
14       print("cartVersion post addToCart: ${cart?.version}");
15       return cart;
16     } catch (e) {
17       print("error $e ");
18     }
19     return cart;
20 }
```

La controparte di `addToCartPostOffline`, denominata `AddToCart`, assume la responsabilità di gestire l'aggiunta al carrello sia quando è disponibile la connettività di rete, sia in assenza di connessione. In particolare, si occupa di gestire l'aggiunta al carrello quando l'utente si trova offline, intervenendo direttamente sul `json` del carrello precedentemente salvato.

In pratica, `AddToCart` si adatta dinamicamente alle condizioni di connettività, garantendo un'esperienza fluida all'utente indipendentemente dalla presenza di una

connessione internet. Quando l'utente è offline, questa funzione agisce sul carrello locale, consentendo comunque di eseguire operazioni come l'aggiunta di prodotti al carrello senza dipendere dalla connettività di rete.

Nel primo caso, l'aggiunta di un prodotto già presente nel carrello, comporta semplicemente un aumento della quantità del prodotto esistente nel carrello. Questo processo avviene in modo trasparente per l'utente, poiché l'interfaccia mostra solo il risultato finale, cioè la quantità aggiornata del prodotto nel carrello.

Nel secondo caso, sebbene simile al primo, si presenta una differenza sostanziale. Qui, quando si aggiunge un prodotto al carrello che non era precedentemente presente, è necessario intervenire manualmente sul `json` del carrello per inserire il nuovo oggetto. Questo processo richiede un'attenta manipolazione dei dati per garantire che il nuovo oggetto venga correttamente inserito nel carrello esistente senza compromettere l'integrità dei dati.

Ogni elemento nel carrello è rappresentato da una `LineItem` che contiene tutte le informazioni necessarie per quel singolo prodotto. Costruire correttamente una nuova `LineItem` richiede attenzione ai dettagli e una conformità rigorosa al formato atteso dal client.

Ogni `LineItem` deve essere strutturata correttamente e contenere le informazioni pertinenti al prodotto, come il nome, l'immagine, il prezzo e la quantità. Qualsiasi deviazione dal formato previsto potrebbe causare malfunzionamenti o errori nell'applicazione.

Pertanto, quando si aggiunge un nuovo prodotto al carrello, è fondamentale assicurarsi che la nuova `LineItem` sia costruita correttamente, rispettando tutti i requisiti di formattazione e contenendo tutte le informazioni richieste. Questo garantisce un'esperienza utente senza problemi e un funzionamento ottimale dell'applicazione.

```
1  Future<Cart?> addToCart(BuildContext context, String productID)
       async {
2      Cart? cart;
3
4      print("Chiamo addToCart");
5
6      bool isOnline = await NetworkController.isOnline();
7
8      try {
9        if (cartAlreadyCreated) {
10         [...]
11       } else {
12         print("Sono offline");
13
14         // Aggiungi l'azione alla lista delle azioni offline
15         await NetworkController.addActionToOfflineList(
```

```
16              'addToCart', productID);
17
18          // Carica il Cart localmente
19          print("Carico CachedCart");
20          final cachedData = await LocalStorage.loadData('cachedCart
    ');
21
22          if (cachedData != null) {
23              Map<String, dynamic> json = jsonDecode(cachedData);
24
25              // Cerca il prodotto nella lista lineItems
26              bool productExists = false;
27              int existingProductIndex = -1;
28              for (int i = 0; i < json['lineItems'].length; i++) {
29                  if (json['lineItems'][i]['productId'] == productID
    ) {
30                      productExists = true;
31                      existingProductIndex = i;
32                      break;
33                  }
34              }
35
36              if (productExists) {
37                  // Se il prodotto è già presente, aggiorna la
    quantità
38                  print("Prodotto già nel carrello: $productExists")
    ;
39
40                  var lineItem = json['lineItems'][
    existingProductIndex];
41
42                  var oldQuantity = lineItem['quantity'];
43                  var priceItem = lineItem['price']['value']['
    centAmount'];
44                  var oldLineItemsTotalPrice = lineItem['totalPrice'
    ]['centAmount'];
45                  var oldTotalPrice = json['totalPrice']['centAmount
    '];
46
47                  var newQuantity = oldQuantity + 1;
48                  var newLineItemsTotalPrice =
    oldLineItemsTotalPrice + priceItem;
49                  var newTotalPrice = oldTotalPrice + priceItem;
50
51                  json['lineItems'][existingProductIndex]['quantity'
    ] = newQuantity;
52                  json['lineItems'][existingProductIndex]['
    totalPrice']
53                      ['centAmount'] = newLineItemsTotalPrice;
```

```
54                  json['totalPrice']['centAmount'] = newTotalPrice;
55              } else {
56                  // Se il prodotto non è presente, aggiungilo al
    carrello
57                  print("Prodotto Non presente nel carrello:
    $productExists");
58
59                  final cachedData =
60                      await LocalStorage.loadData('
    cachedProduct$productID');
61
62                  if (cachedData != null) {
63                      final Map<String, dynamic> cachedProduct =
64                          await jsonDecode(cachedData);
65
66                      final responseData = await jsonDecode(
    cachedData);
67                      final products = ProductByID.fromJson(
    responseData);
68
69                      var priceItem = products.masterData?.current?.
    masterVariant
70                              ?.prices?[0].value?.centAmount! ??
    10000;
71                      var nameEn =
72                          cachedProduct['masterData']['current']['
    name']['en'];
73                      var nameDe =
74                          cachedProduct['masterData']['current']['
    name']['de'];
75                      var imageUrl = product
76                              ?.masterData?.current?.masterVariant?.
    images?[0].url ?? "";
77
78                      var newLineItem = {
79                          'productId': productID,
80                          'name': {'en': nameEn, 'de': nameDe},
81                          'quantity': 1,
82                          'price': {
83                              'value': {'centAmount': priceItem}
84                          },
85                          'totalPrice': {'centAmount': priceItem},
86                          'variant': {
87                              'images': [
88                                  {'url': imageUrl}
89                              ]
90                          }
91                      };
92
```

```
93                      json['lineItems'].add(newLineItem);
94                      json['totalPrice']['centAmount'] += priceItem;
95                  }
96              }
97
98              cart = Cart.fromJson(json);
99
100             // Sovrascrivi i dati locali con il nuovo carrello
    JSON
101             await LocalStorage.saveData('cachedCart', jsonEncode(
    json));
102         }
103
104         return cart;
105       }
106     } catch (e) {
107       print("error $e ");
108     }
109     return cart;
110 }
```

Gestione del conflitto

Grazie all'implementazione di una serie di funzioni, sia nuove che modificate, ho raggiunto un efficace livello di gestione del carrello in modalità offline. Tuttavia un ultimo caso, la cui gestione si presentava particolarmente ostica, ha meritato la mia attenzione specifica. Sto parlando della circostanza, non del tutto improbabile, in cui un utente cerca di aggiungere al carrello un prodotto che nel frattempo risulta esaurito e dunque non disponibile.

In questo caso è necessario notificare tempestivamente al cliente l'inconveniente e impedire l'aggiunta del prodotto nel carrello. Per vincere questa sfida, ho apportato delle modifiche alla funzione `addToCartPostOffline`.

All'interno di questa funzione, dopo aver eseguito la chiamata all'API, è attesa una risposta con codice 200 per confermare l'avvenuta aggiunta al carrello. Tuttavia, nel caso in cui il prodotto diventi indisponibile nel frattempo, la chiamata all'API non restituirà il codice 200. Sfruttando questa informazione, ho implementato una gestione apposita per questo scenario.

Nel caso di una risposta negativa dall'API, ho introdotto la visualizzazione di una `snackbar`, la stessa utilizzata per segnalare le variazioni nella connettività. Questo messaggio informa l'utente dell'errore nell'aggiornamento del carrello, consentendo così una chiara comunicazione dell'evento anomalo.

Capitolo 6

Risultati e considerazioni finali

6.1 Implementazione della Super App

Tramite quindi le **Webviews** e l'ausilio di librerie come `js`, siamo riusciti a riprodurre una demo di una Super App perfettamente funzionante, con funzionalità di base ma che illustrano perfettamente le potenzialità di tale approccio.

Come considerazioni, però, bisogna mettere in luce come tale approccio necessiti di alcune condizioni affinché sia fattibile:

- Le Mini App devono *vivere* nel web e avere un URL pubblico;
- Per quanto le Mini App siano completamente scisse e indipendenti dalla Super App, vi deve essere una sorta di comunicazione tra i team della Super App e i team delle Mini App, soprattutto qualora si volessero farle comunicare come abbiamo fatto noi.

Bisognerebbe concordare previamente quali messaggi usare per comunicare gli eventi alla Super App, quali parametri la Mini App si aspetta di ricevere durante la fase di avvio e, se si aspetta dati in modo asincrono, in che formato essi vengano ricevuti.

Riguardo a questo punto, una considerazione importante da fare è che la necessità di definire un protocollo di comunicazione tra la Super App e le Mini App non rappresenta un ostacolo significativo. Sin dagli albori dello sviluppo software, i programmatori hanno sempre dovuto conformarsi a specifiche tecniche per garantire l'integrazione tra tecnologie diverse. L'utilizzo di API, protocolli standard e best practice consolidate fa parte della quotidianità dello sviluppo software moderno.

Anche nelle vere Super App attualmente sul mercato, è altamente probabile che esistano linee guida precise per la sottoscrizione e l'integrazione di Mini App, definendo standard chiari per la comunicazione e l'interoperabilità. In questo senso, il modello proposto non si discosta concettualmente dalle soluzioni adottate su larga scala, ma anzi si inserisce in una prassi consolidata nel settore dello sviluppo di ecosistemi digitali integrati.

Nonostante ciò, risulta un ottimo approccio, soprattutto per le aziende che intendono sviluppare una Super App proprietaria, avendo quindi il pieno controllo sui team della Super App e delle varie Mini App.

Qualora non vi sia una particolare esigenza di comunicazione tra la Super App e le varie Mini App, risulta ancora più comodo ed immediato, regalando una user experience elevata all'utente, che si trova potenzialmente sommerso di microservizi utili, tutti all'interno della stessa applicazione.

6.2 Implementazione dell'approccio Offline-First

L'implementazione dell'approccio *Offline-First* nell'applicazione ha richiesto un'accurata progettazione e gestione di diverse funzionalità, con particolare attenzione alla persistenza dei dati e alla sincronizzazione con il server.

Attraverso l'uso delle `Shared Preferences`, è stato possibile garantire un'esperienza utente fluida anche in assenza di connessione Internet, migliorando la reattività e riducendo i tempi di attesa. In particolare, la strategia adottata di salvare localmente la risposta delle API ha dimostrato di essere una scelta efficace per migliorare la disponibilità dei dati e garantire una navigazione senza interruzioni anche in condizioni di connettività instabile.

Uno degli esempi più rilevanti di questa implementazione riguarda la gestione del catalogo prodotti. Salvando le risposte delle API nelle `Shared Preferences`, siamo riusciti a ridurre significativamente i tempi di caricamento nelle sessioni offline. In scenari di test, il caricamento del catalogo da API ha richiesto in media **350–500 ms**, mentre il recupero dei dati dalle `Shared Preferences` ha permesso di ottenere gli stessi risultati in meno di **10 ms**, migliorando sensibilmente la percezione di velocità dell'applicazione da parte dell'utente.

Allo stesso modo, la gestione del carrello offline, con il monitoraggio delle modifiche e la corretta sincronizzazione delle operazioni una volta ristabilita la connessione, ha rappresentato una delle sfide principali. Tuttavia, grazie a un'accurata gestione dei dati locali e delle azioni in sospeso, siamo riusciti a garantire un'esperienza coerente e affidabile per l'utente, anche in caso di connessione intermittente. Inoltre, la gestione di casi particolari, come l'indisponibilità di un prodotto al momento dell'aggiunta al carrello, ha contribuito a mantenere la trasparenza dell'applicazione e a migliorare la UX complessiva.

Nonostante i vantaggi offerti, è importante considerare alcune limitazioni delle `Shared Preferences`, specialmente quando l'applicazione cresce in complessità e richiede una gestione più avanzata dei dati.

6.2.1 Limitazioni dell'uso delle Shared Preferences

Le `Shared Preferences` sono uno strumento utile per memorizzare dati semplici sotto forma di coppie chiave-valore, garantendo un accesso rapido e una gestione semplificata delle informazioni persistenti. Come già detto, la loro leggerezza e semplicità le rendono ideali per la memorizzazione di preferenze utente o impostazioni di base, oltre che per la memorizzazione locale di risposte API quando l'app necessita di un caching leggero. Tuttavia, quando vengono utilizzate per gestire dati complessi o voluminosi, emergono alcune limitazioni significative che devono essere prese in considerazione.

Capacità massima di storage: Uno dei principali problemi delle `Shared Preferences` è la loro capacità di memorizzazione. Anche se non esiste un limite ufficiale, nella pratica Google **sconsiglia di superare i 100 KB** per file di preferenze, poiché file di grandi dimensioni possono rallentare l'accesso ai dati e compromettere le prestazioni dell'applicazione.

Nel nostro caso, il salvataggio di dati JSON compressi nelle `Shared Preferences` si è rivelato efficace per garantire l'accesso rapido alle informazioni principali, come il catalogo prodotti e lo stato del carrello. Tuttavia, per dataset più ampi, questa soluzione potrebbe risultare limitante, rendendo necessario l'utilizzo di database più strutturati.

Assenza di supporto per query avanzate e sincronizzazione: Un'altra limitazione rilevante riguarda l'assenza di supporto per query avanzate. Poiché i dati vengono archiviati in un semplice file XML, non è possibile eseguire operazioni di filtraggio o interrogazioni strutturate come avviene nei database relazionali o NoSQL.

Inoltre, le `Shared Preferences` non offrono un meccanismo di *sincronizzazione automatica* tra più dispositivi. Questo può causare problemi di **incoerenza dei dati** tra versioni locali e remote, specialmente in applicazioni che richiedono un costante aggiornamento delle informazioni.

Problemi di sicurezza e privacy: Come già evidenziato, un ulteriore svantaggio delle `Shared Preferences` è la **mancanza di crittografia automatica**. I dati memorizzati sono accessibili in chiaro e vulnerabili ad attacchi di *reverse engineering*, rendendoli inadatti per archiviare informazioni sensibili come credenziali utente o dati finanziari.

Per mitigare questo problema, è necessario adottare tecniche di crittografia manuale, come l'uso dell'API `EncryptedSharedPreferences` di Android, sebbene ciò introduca ulteriore complessità nello sviluppo.

Impatto sulla scalabilità dell'applicazione: Come già discusso, le `Shared Preferences` non sono progettate per la gestione di **grandi quantità di dati dinamici**. Un utilizzo eccessivo può comportare problemi quali:

- *Rallentamenti nelle operazioni di lettura e scrittura*, soprattutto se il file XML cresce eccessivamente.
- *Errori di out-of-memory*, poiché l'intero file di preferenze viene caricato in memoria ogni volta che viene richiamato.
- *Possibile corruzione dei dati*, specialmente in scenari in cui l'applicazione si chiude inaspettatamente mentre i dati vengono aggiornati.

In conclusione, le `Shared Preferences` si sono dimostrate un'ottima soluzione per il nostro scopo, garantendo un caching locale efficiente per i dati critici e migliorando la reattività dell'app. Tuttavia, per scenari più avanzati che richiedono maggiore scalabilità, sincronizzazione tra dispositivi o query avanzate, potrebbe essere opportuno valutare soluzioni più robuste come database locali ottimizzati per il mobile.

Capitolo 7

Conclusione

7.1 Conclusioni

Il lavoro svolto ha dimostrato che un'architettura basata su *Super App*, Webviews e *Offline-First* può essere efficace e applicabile in scenari reali. Le soluzioni adottate hanno garantito **scalabilità**, **modularità** e una migliore esperienza utente, confermando la validità dell'approccio.

L'integrazione delle *Webviews* ha reso la gestione delle Mini App semplice e flessibile, mentre l'adozione di un modello *Offline-First* ha migliorato la reattività e l'affidabilità del sistema. Nonostante alcune limitazioni, le scelte fatte hanno permesso di raggiungere gli obiettivi prefissati e pongono solide basi per futuri sviluppi.

7.2 Lavori futuri

Le scelte progettuali adottate in questo lavoro di tesi si sono rivelate efficaci per gli obiettivi prefissati. I risultati ottenuti evidenziano la validità delle tecnologie utilizzate e aprono la strada a sviluppi futuri che potrebbero rendere il sistema ancora più robusto e scalabile, adattandolo a scenari reali e di larga scala.

7.2.1 Evoluzione della Super App e delle Mini App

Uno degli aspetti più interessanti emersi da questo studio è l'efficacia delle **Webviews** nella realizzazione di una Super App modulare. Questa scelta tecnica si è dimostrata un valido alleato per garantire flessibilità e semplicità d'integrazione, caratteristiche essenziali anche in un progetto di produzione su larga scala. Tuttavia, mentre nel nostro caso l'ecosistema della Super App è stato costruito partendo da un'applicazione esistente (un e-commerce) e adattato per accogliere Mini App, un

passo evolutivo naturale per un progetto futuro potrebbe essere quello di progettare una Super App fin dall'inizio con questa logica.

Un'architettura concepita fin dall'inizio come una Super App permetterebbe di progettare un ecosistema più coerente e ottimizzato, evitando l'adattamento di un'app preesistente – come nel nostro caso con l'e-commerce – e costruendo invece una piattaforma pensata esclusivamente per ospitare e gestire Mini App. In questo scenario, un e-commerce potrebbe essere solo una delle Mini App all'interno della Super App, anziché rappresentare il nucleo principale su cui vengono integrate altre funzionalità. Questo approccio consentirebbe di realizzare una Super App più flessibile, capace di accogliere Mini App con servizi differenti e complementari, rafforzando il valore dell'ecosistema digitale nel suo complesso.

Un altro sviluppo rilevante riguarderebbe le **Mini App** stesse. Nel nostro caso, l'implementazione è stata finalizzata principalmente a testare i meccanismi di comunicazione tra Super App e Mini App, senza sviluppare funzionalità reali. Un'estensione futura potrebbe consistere nello sviluppo di Mini App con servizi completi e diversificati, che diano un vero valore aggiunto all'ecosistema della Super App, integrando funzionalità che spaziano dall'e-commerce ai servizi finanziari, fino a strumenti per la produttività o il social networking.

7.2.2 Miglioramenti nell'approccio Offline-First

L'approccio *Offline-First* si è rivelato una strategia vincente, migliorando la fruibilità dell'applicazione e garantendo continuità operativa in assenza di connessione. Tuttavia, per un utilizzo in scenari reali e su larga scala, sarebbero necessarie alcune ottimizzazioni.

Una delle principali evoluzioni riguarda la **gestione dei dati locali**. Sebbene l'uso delle `Shared Preferences` si sia dimostrato efficace per il nostro scopo, garantendo un accesso rapido ai dati e migliorando la reattività dell'applicazione, questa soluzione presenta limiti evidenti quando il volume dei dati cresce. Per una Super App complessa, un'architettura più solida potrebbe basarsi su database ottimizzati per dispositivi mobili, come:

- **SQLite**, per la gestione strutturata dei dati con supporto a query avanzate.
- **Hive**, un database NoSQL altamente performante e ottimizzato per Flutter.
- **Room** (per Android), un'astrazione di SQLite che semplifica la gestione dei dati offline.

L'adozione di una di queste soluzioni permetterebbe una gestione più efficiente dei dati locali, garantendo sia la persistenza delle informazioni che la sincronizzazione ottimizzata con il backend.

Un ulteriore miglioramento riguarderebbe la progettazione dell'approccio Offline-First fin dalle prime fasi di sviluppo. Nel nostro caso, la funzionalità offline è stata integrata in un'applicazione già esistente, adattando il sistema per supportare la memorizzazione locale dei dati. Tuttavia, in un contesto produttivo, una pianificazione anticipata di questa strategia consentirebbe di ottimizzare l'intera architettura, sfruttando appieno i benefici dell'Offline-First non solo per gestire l'assenza di connessione, ma anche per migliorare l'esperienza utente in ogni condizione di utilizzo.

Adottare un sistema di caching intelligente permetterebbe di ridurre il numero di chiamate ridondanti al server, garantendo un accesso più rapido ai dati e migliorando le prestazioni complessive dell'applicazione anche in presenza di connessione. In questo modo, l'Offline-First non sarebbe solo un meccanismo di emergenza in caso di disconnessione, ma diventerebbe parte integrante dell'ottimizzazione delle sessioni online, riducendo il consumo di rete e migliorando la fluidità dell'interazione utente.

Inoltre, prevedere fin dall'inizio una gestione strutturata della sincronizzazione dei dati tra Super App e Mini App consentirebbe di evitare inconsistenze e garantire una maggiore coerenza tra le informazioni locali e quelle sul server, assicurando un'esperienza uniforme indipendentemente dallo stato della rete.

Bibliografia

[1] Alessandro De Bartolo e Pierluigi Stifanelli. *Super App*. 2022. URL: https://it.nttdata.com/insights/blog/super-app (cit. a p. 1).

[2] Alessandra Boraso. *Cosa sono le Super App e perché ne sentiremo parlare*. 2022. URL: https://www.fintastico.com/it/blog/cosa-sono-le-super-app-e-perche-ne-sentiremo-parlare/ (cit. a p. 1).

[3] NTD Italia. *Cosa sono le Super App?* 2023. URL: https://www.ntditalia.com/cosa-sono-le-super-app/ (cit. a p. 2).

[4] Gennaro Cuofano. *Super App*. 2024. URL: https://fourweekmba.com/it/super-app/ (cit. a p. 2).

[5] Dan Prud'homme, Guoli Chen e Tony W. Tong. *Le Super App stanno arrivando sul mercato USA*. 2023. URL: https://www.hbritalia.it/homepage/2023/05/10/news/le-super-app-stanno-arrivando-sul-mercato-usa-15529/ (cit. a p. 3).

[6] Lori Perri. *What is a Superapp?* 2022. URL: https://www.gartner.com/en/articles/what-is-a-superapp (cit. a p. 3).

[7] Alessandra Colarizi. *WeChat: una super app per conquistare il mercato cinese*. 2022. URL: https://www.china-files.com/wechat-una-super-app-per-conquistare-il-mercato-cinese/ (cit. a p. 7).

[8] Chinese Business Academy. *WeChat: introduzione alla super app cinese*. URL: https://chinesebusinessacademy.com/china-export-kit/wechat-introduzione-alla-super-app-cinese/ (cit. a p. 7).

[9] Daniel Wang. *WeChat Product Design Case Study: How to Build a Super App and Why Western Companies Struggle*. 2023. URL: https://medium.com/qmind-ai/wechat-product-design-case-study-how-to-build-a-super-app-and-why-western-companies-struggle-acf40f49eed5 (cit. a p. 8).

[10] Marcus Law. *Top 10 Super Apps*. 23. URL: https://technologymagazine.com/top10/top-10-super-apps (cit. a p. 9).

[11] Yeeply. *6 migliori esempi di Super App*. URL: https://www.yeeply.com/it/blog/sviluppo-app-mobile/6-migliori-esempi-super-app/#6 (cit. a p. 10).

[12] Niketan Sharma. *Super App Development*. 2022. URL: https://www.nimbleappgenie.com/blogs/super-app-development/ (cit. a p. 12).

[13] Grand View Research. *Super Apps Market Report*. 2023. URL: https://www.grandviewresearch.com/industry-analysis/super-apps-market-report#:~:text=The%20global%20super%20apps%20market%20size%20was%20estimated%20at%20USD,USD%20426.01%20billion%20by%202030. (cit. alle pp. 12, 14, 28).

[14] Vikas I. *Designing a Mobile Application Architecture for Offline Use*. 2023. URL: https://medium.com/@i.vikas/designing-a-mobile-application-architecture-for-offline-use-e03e7731f4f2 (cit. a p. 18).

[15] Mark Gamble. *Offline-First: More Reliable Mobile Apps*. 2023. URL: https://www.couchbase.com/blog/offline-first-more-reliable-mobile-apps/ (cit. a p. 18).

[16] Manikandan Palani e Kirubhakar Thangaraj. *Offline-First Application Insights*. 2022. URL: https://www.bigthinkcode.com/insights/offline-first-application (cit. a p. 21).

[17] Muzammil K. *Offline App Architecture: Building Offline-First Apps*. 2024. URL: https://www.aalpha.net/blog/offline-app-architecture-building-offline-first-apps/ (cit. a p. 22).

[18] Arnold Parge. *What is Offline-First Architecture?* 2023. URL: https://blog.nonstopio.com/what-is-offline-first-architecture-af5537095a9c (cit. a p. 25).

[19] Bernardo Iribarne. *Offline-First Application: What Does It Mean and How to Implement This Concept in Your Next Flutter App*. 2023. URL: https://medium.com/nerd-for-tech/offline-first-application-what-does-it-mean-and-how-to-implement-this-concept-in-your-next-flutter-ff40f4b62c6c (cit. a p. 27).

[20] Prachi Palkhiwala. *Super App Development Guide*. 2024. URL: https://radixweb.com/blog/super-app-development-guide#Super (cit. a p. 28).

[21] Jacklin Altman. *How the Web Enables Building SuperApps*. 2023. URL: https://ionic.io/blog/how-the-web-enables-building-superapps#:~:text=A%20superapp%20is%20essentially%20a,creating%20a%20unified%20end%20product. (cit. a p. 29).

[22] Prashant Krishnan. *Super App Development: Features and Cost*. 2022. URL: https://medium.com/mqos-technologies/super-app-development-features-and-cost-1232341c1423 (cit. a p. 29).

[23] Wildang Budhi. *Microservices, Clean Architecture, and Kafka in Gojek*. 2020. URL: https://wildangbudhi.medium.com/microservices-clean-architecture-and-kafka-in-gojek-c5fbe60dea9c (cit. a p. 31).

[24] Tuan Hoang. *How to Make a Super App*. 2021. URL: https://hoangatuan.medium.com/how-to-make-a-super-app-75c2588dac42 (cit. a p. 31).

[25] Michael Geers. *Micro Frontends*. URL: https://riccardogmoschetti.github.io/micro-frontends/ (cit. a p. 34).

[26] Martin Fowler e James Lewis. *Microservices*. 2014. URL: https://martinfowler.com/articles/microservices.html#footnote-etymology (cit. a p. 34).

[27] Cam Jackson. *Micro Frontends*. 2019. URL: https://martinfowler.com/articles/micro-frontends.html (cit. a p. 35).

[28] Vivek Shukla. *A Comprehensive Guide to Micro Frontend Architecture*. 2023. URL: https://medium.com/appfoster/a-comprehensive-guide-to-micro-frontend-architecture-cc0e31e0c053 (cit. a p. 35).

[29] Luis Cameroon. *Micro Frontends with Module Federation*. 2023. URL: https://luiscameroon.medium.com/micro-frontends-with-module-federation-d5d9e135b0f1 (cit. a p. 36).

[30] Wina Arambule. *How to Transform a Native App into a Super App*. 2022. URL: https://genexus.blog/en_US/genexus-platform/how-to-transform-a-native-app-into-a-super-app/ (cit. a p. 39).

[31] Ionic. *Superapp Starter Guide*. URL: https://ionic.io/docs/superapp-starter (cit. a p. 39).

[32] Christine Perez. *Superapps and Composable Mobile App Development*. URL: https://ionic.io/resources/articles/superapps-composable-mobile-app-development#h-make-your-app-modular (cit. a p. 39).

[33] Jakub Romańczyk. *Step-by-Step Guide to Super App Development*. 2023. URL: https://www.callstack.com/blog/step-by-step-guide-to-super-app-development (cit. a p. 39).

[34] Jakub Romańczyk. *A Step-by-Step Guide to Super App Development*. 2023. URL: https://dev.to/callstackengineers/a-step-by-step-guide-to-super-app-development-1gmm (cit. a p. 39).

[35] Jakub Romańczyk. *Case Study: Super App Template*. 2023. URL: https://www.callstack.com/blog/case-study-super-app-template (cit. a p. 39).

[36] Candra Aji. *Re.Pack in React Native*. 2022. URL: https://medium.com/aia-sg-techblog/re-pack-in-react-native-7d84e963f815 (cit. a p. 39).

[37] Codestax AI. *Of Webviews and Superapps*. 2023. URL: https://www.codestax.ai/technologies-dark/of-webviews-and-superapps (cit. a p. 39).

[38] Parth Mahajan. *Hybrid Application: Native-Like Experience with WebView*. 2022. URL: https://medium.com/1mgofficial/hybrid-application-native-like-experience-with-webview-9896f61881cb (cit. a p. 41).

[39] Google. *Progressive Web Apps*. URL: https://web.dev/explore/progressive-web-apps?hl=it (cit. a p. 41).

[40] Louis Japheth Kouassi. *Make Your PWA Work Offline with a Service Worker*. 2023. URL: https://medium.com/@louisjaphethkouassi/make-your-pwa-work-offline-with-a-service-worker-f95f16d93029 (cit. a p. 42).

[41] Google Web Dev Team. *Service Workers*. URL: https://web.dev/learn/pwa/service-workers?hl=it (cit. a p. 44).

[42] Google Web Dev Team. *Caching in Progressive Web Apps*. URL: https://web.dev/learn/pwa/caching?hl=it (cit. a p. 44).

[43] Google Web Dev Team. *Offline Data in Progressive Web Apps*. URL: https://web.dev/learn/pwa/offline-data?hl=it (cit. a p. 44).

[44] Android Developers. *Shared Preferences API*. URL: https://developer.android.com/training/data-storage/shared-preferences?hl=it (cit. a p. 45).

[45] Punith S Uppar. *Offline-First Approach with Flutter*. 2025. URL: https://medium.com/@punithsuppar7795/offline-first-approach-with-flutter-3468906eb01a (cit. a p. 46).

[46] Bartosz Skuza, Jakub Janiec e Inez Bartosińska. *Flutter vs React Native Comparison*. 2024. URL: https://www.thedroidsonroids.com/blog/flutter-vs-react-native-comparison (cit. a p. 49).

[47] Mauricio Pastorini. *Flutter vs React Native*. 2024. URL: https://somniosoftware.com/blog/flutter-vs-react-native (cit. a p. 52).

[48] IT Creative Labs. *Flutter vs React Native 2023: Which is Better for Mobile App?* 2023. URL: https://www.linkedin.com/pulse/flutter-vs-react-native-2023-which-better-mobile-app (cit. a p. 55).

[49] Marvel Apps. *Everything About WebView in Flutter*. 2024. URL: https://medium.com/@MarvelApps_/everything-about-webview-in-flutter-ab56a2315f0f (cit. a p. 61).

[50] Ashish Garg. *Working with WebView in Flutter*. 2021. URL: https://medium.com/@ashishgarg1998/working-with-webview-in-flutter-287ec492ec2a (cit. a p. 66).

[51] Tomer. *How to Build a Native Communication Bridge in Flutter with WebView and Javascript*. 2024. URL: https://www.freecodecamp.org/news/how-to-build-a-native-communication-bridge-in-flutter-with-webview-and-javascript/ (cit. a p. 69).