ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA

Corso di Laurea in Ingegneria e Scienze Informatiche

FLUTTER: LA NUOVA FRONTIERA DELLE APP PLATFORM-INDEPENDENT

Tesi di Laurea in Programmazione di Sistemi Mobile

RELATORE:

Alessandro Ricci

PRESENTATA DA:

Matteo Scucchia

CORRELATORE:

Angelo Croatti

ANNO ACCADEMICO: 2018-2019

AI MIEI CARI

Introduzione

Il termine "platform-independent" in informatica si riferisce a linguaggi di programmazione, applicazioni software o dispositivi hardware che funzionano su differenti sistemi operativi e quindi su più piattaforme [45]. Le app platform-independent, anche denominate cross-platform, sono applicazioni mobile che non necessitano di differenti codifiche per essere operative su diversi sistemi. La tesi è incentrata sul framework Flutter, utilizzato per lo sviluppo della parte progettuale, che permette di scrivere un unico codice in linguaggio Dart e renderlo funzionante sia in ambiente Android che in ambiente iOS. Il mercato delle app mobile si sta espandendo esponenzialmente e sempre più spesso sono ricercate soluzioni che ne semplifichino lo sviluppo. Flutter è un progetto nato proprio per questo in quanto mette a disposizione Dart, un linguaggio molto semplice e intuitivo, ma al contempo estremamente potente, e astrazioni di Google già collaudate come la libreria Material che consentono la creazione in tempi rapidi di applicazioni fluide e ottimizzate. Sarà presentato un caso di studio realizzato con questa tecnologia che permette di gestire il modulo CRM di una azienda, aiutando e indirizzando i clienti negli acquisti.

Indice

Introduzione **i**

1 Le applicazioni mobile **1**

1.1 Categorie di app 1

1.1.1 Native 1

1.1.2 Progressive Web App 2

1.1.3 App ibride 2

1.1.4 Cross-Platform 2

1.2 Lo stato dell'arte 3

1.2.1 Perchè implementare un'applicazione cross-platform 4

1.2.2 Framework platform-independent 6

1.2.3 Xamarin 7

1.3 Alternative a JavaScript 10

1.4 Panoramica su TypeScript 10

2 Il linguaggio Dart **13**

2.1 Panoramica sul linguaggio 13

2.1.1 Vantaggi del typing 14

2.1.2 L'event loop di Dart 14

2.1.3 Gestore di pacchetti Pub 16

2.2 Principali meccanismi 17

2.2.1 dynamic 17

2.2.2 Future 17

2.2.3 Async e Await 18

2.2.4 Stream 18
2.3 La scelta di Dart 19
2.3.1 Introduzione a JavaScript 19
2.3.2 Vantaggi di Dart 19
2.4 Confronto tra Dart e TypeScript 20
2.4.1 Introduzione a TypeScript 20
2.4.2 Differenze fondamentali 21
2.4.3 Conclusioni 21
2.5 Conclusioni 22

3 Il framework Flutter 23
3.1 Concetti fondamentali 23
3.1.1 Struttura di Flutter 24
3.1.2 Motore grafico Skia 25
3.1.3 Widget 25
3.1.4 Flutter predilige la composizione 27
3.1.5 Navigator e Route 28
3.1.6 Interfaccia utente asincrona 29
3.1.7 Ciclo di vita della applicazioni 29
3.1.8 Gestione degli eventi a schermo 30
3.1.9 Libreria Material 31
3.1.10 Libreria Cupertino 31
3.1.11 pubspec.yaml 31
3.2 Widget principali 32
3.2.1 Scaffold 32
3.2.2 AppBar 33
3.2.3 Container 34
3.2.4 Row 35
3.2.5 Column 36
3.2.6 ListView 36
3.2.7 GridView 37

3.3 Differenze e analogie con Android 38
3.3.1 Le View 38
3.3.2 Gli Intents 38
3.3.3 UI asincrona 39
3.3.4 Le Activities e i Fragment 39
3.3.5 Gradle 40
3.4 Differenze e analogie con iOS 40
3.4.1 UIView 40
3.4.2 ViewController 41
3.4.3 Navigare tra le pagine 41
3.4.4 Thread e programmazione asincrona 41
3.4.5 CocoaPods 41
3.5 FireBase in Flutter 41
3.5.1 Funzionalità principali 42
3.6 Contronto tra Flutter e Xamarin 42
3.6.1 Introduzione a Xamarin 43
3.6.2 Linguaggio di programmazione 43
3.6.3 Architettura 43
3.6.4 Curva di apprendimento 44
3.6.5 Interfaccia utente 44
3.6.6 Community di supporto 44
3.6.7 Conclusione 45

4 Flutter e la programmazione reattiva 47
4.1 Programmazione reattiva 47
4.2 Programmazione reattiva in flutter 48
4.3 Pattern BLoC 49
4.3.1 Funzionamento di BLoC 49
4.3.2 Bloc Provider 49
4.3.3 StreamBuilder 49
4.3.4 Integrazione in Flutter 50

4.4 Pro e contro della programmazione reattiva 51

5 Caso di studio: Active CRM 53

5.1 Active CRM 53
5.2 Requisiti 53
5.2.1 Clienti 54
5.2.2 Contatti 54
5.2.3 Azioni 54
5.2.4 Target 55
5.2.5 Casi d'uso 55
5.3 Progettazione dell'applicazione 57
5.3.1 Widget dell'applicazione 58
5.3.2 Implementazione delle web API 58
5.3.3 Classi dell'applicazione 59
5.3.4 Autorizzazioni 59
5.3.5 Notifiche push 60
5.4 Architettura dell'applicazione 61
5.4.1 HomePage 62
5.4.2 Form di modifica 62
5.4.3 Dettagli 63
5.4.4 Classi di visualizzazione 63
5.5 Implementazione prototipale 64
5.5.1 Classi principali 64
5.5.2 View dell'applicazione 69
5.6 Valutazione dell'applicazione 73
5.6.1 Criteri di valutazione 73
5.6.2 Cosa migliorare 74

6 Conclusione 75

6.1 Considerazioni sulle app platform-independent 75
6.2 Considerazioni su Dart 76
6.3 Considerazioni su Flutter 76

6.4 Considerazioni personali 78
6.5 Sviluppi futuri 78

Elenco delle figure

1.1 Percentuale delle applicazioni native e ibride presenti sul mercato. . . 3
1.2 Alternative a Flutter: punti di debolezza 5

2.1 Architettura event loop 14
2.2 Modello a singolo thread di Dart 15
2.3 Funzionamento dell'event loop in una applicazione Dart 16
2.4 Confronto con Node.js nell'ambito della velocità di esecuzione di alcuni programmi di problem solving. 20

3.1 Struttura a strati di Flutter 24
3.2 Metodo corretto per modificare l'albero delle Widget mediante l'uso di un Container vuoto. 26
3.3 Gerarchia delle Widget e Widget composte. 27
3.4 Navigator push 28
3.5 Navigator pop 28
3.6 Inserimento di una dipendenza 32
3.7 Widget Scaffold 33
3.8 Widget AppBar 34
3.9 Widget Container 35
3.10 Widget Row 35
3.11 Widget Column 36
3.12 Widget ListView 37
3.13 Widget GridView 37

3.14 Crescita della diffusione di Flutter calcolata come numero di domande inerenti all'argomento presenti su StackOverflow. 43

4.1 Implementazione del pattern BLoC 50

5.1 Caso d'uso per l'inserimento e la modifica di un cliente 56
5.2 Caso d'uso per il completamento di una azione 56
5.3 Caso d'uso per la modifica delle impostazioni 57
5.4 Architettura dell'applicazione 61
5.5 Struttura principale di HomePage 64
5.6 Struttura di DettaglioCliente 65
5.7 Struttura di DettaglioContatto 66
5.8 Struttura di DettaglioAzione 67
5.9 Struttura di DettaglioTarget 68
5.10 Dettagli di due tipologie differenti 69
5.11 PageView della homepage 70
5.12 Form di inserimento e modifica del cliente 71
5.13 Drawer laterale 72

Capitolo 1

Le applicazioni mobile

Le applicazioni mobile sono software applicativi realizzati appositamente per essere eseguiti su dispositivi mobile, come smartphone o tablet. Oggigiorno è possibile trovare sul mercato diverse soluzioni per la programmazione di applicazioni mobile. Stanno prendendo sempre più piede le soluzioni ibride e platform-independent, in quanto la loro versatilità riduce notevolmente i tempi di produzione e di reingegnerizzazione di una applicazione per trasportarla su diverse piattaforme.

1.1 Categorie di app

Il settore delle applicazioni mobile è nato in tempi relativamente recenti eppure ha già invaso il mercato. In base alle esigenze dei produttori, sono nate diverse soluzioni che permettono di implementare tipi diversi di applicazione, ognuna presentando punti di forza e di debolezza. Le app sono suddivisibili in diverse categorie in base alla modalità con cui sono state progettate e realizzate.

1.1.1 Native

Le applicazioni native sono app che si installano e si utilizzano interamente sul dispositivo mobile e sono create appositamente per uno specifico sistema operativo. Tali applicazioni essendo ottimizzate per un determinato sistema garantiscono prestazioni ottimali e sono assolutamente complete, ossia il linguaggio fornisce il pieno controllo sul dispositivo e i suoi sensori. Essendo tali app sviluppate per un sistema

operativo specifico, nel caso si voglia trasportarle su altre piattaforme è necessario riscrivere completamente l'applicazione. Le app native comunemente sono realizzate in Java o Kotlin per Android e in Swift per iOS.

1.1.2 Progressive Web App

Una Progressive Web App non è una vera e propria applicazione mobile, in quanto sul dispositivo non è installato nessun applicativo. Si può considerare una Web App come un ponte tra il dispositivo dell'utente e un applicativo remoto [46]. A tale scopo vengono utilizzati linguaggi cross-platform come ad esempio HTML5 e JavaScript, per realizzare la medesima Web App sia su dispositivo mobile che su computer. Nella pratica si può considerare una Web App esattamente come un sito web che quando è aperto su device si comporta come app nativa. In questo modo non si incide minimamente sulla memoria o le prestazione del dispositivo mobile, ma per poter utilizzare app di questo tipo è sempre necessario il collegamento internet e le prestazioni dipendono dalla qualità del collegamento stesso.

1.1.3 App ibride

Le applicazioni ibride sono l'anello evolutivo intermedio tra le app native e le Web App [46], ovvero sono realizzate mediante linguaggi adibiti allo sviluppo di applicazioni web, ma per funzionare necessitano di un software applicativo che venga scaricato e installato su device. Questa soluzione garantisce all'utente un'esperienza molto simile alle Progressive Web App, ma intacca l'utilizzo di memoria del dispositivo e di conseguenza le sue performance. Anche questa tipologia fa largamente uso di linguaggi cross-platform come HTML5 e JavaScript.

1.1.4 Cross-Platform

Le applicazioni cross-platform, anche dette platform-independent, sono applicazioni a tutti gli effetti, installabili su device, che, come spiega il nome stesso, funzionano su più piattaforme senza il bisogno di riscrivere l'applicazione per adattarla a diversi sistemi operativi, al contrario delle app native [40]. L'utilizzo di tale soluzione

abbassa notevolemente i tempi di realizzazione di un'applicazione, garantendo buone performance. Tuttavia alcune funzionalità dei dispositivi mobile sono difficilmente accessibili senza utilizzare codice nativo, come ad esempio l'utilizzo di tutti i sensori, può quindi essere necessaria un'integrazione nativa.

1.2 Lo stato dell'arte

Nel passato recente le applicazioni più utilizzate erano realizzate nativamente, ma il trend a partire già dal 2015 sembra essersi invertito [25]. Anche Flutter, il framework utilizzato per la realizzazione del caso di studio, è nato e si è diffuso negli ultimi due anni e ha già conquistato una fetta di mercato molto ampia se rapportata alla sua breve vita [14]. Il motivo di ciò è la necessità di avere sviluppi più rapidi e non legati ad un unico sistema, ricercando maggiormente la portabilità dell'applicazione, l'adattamento a piattaforme differenti.

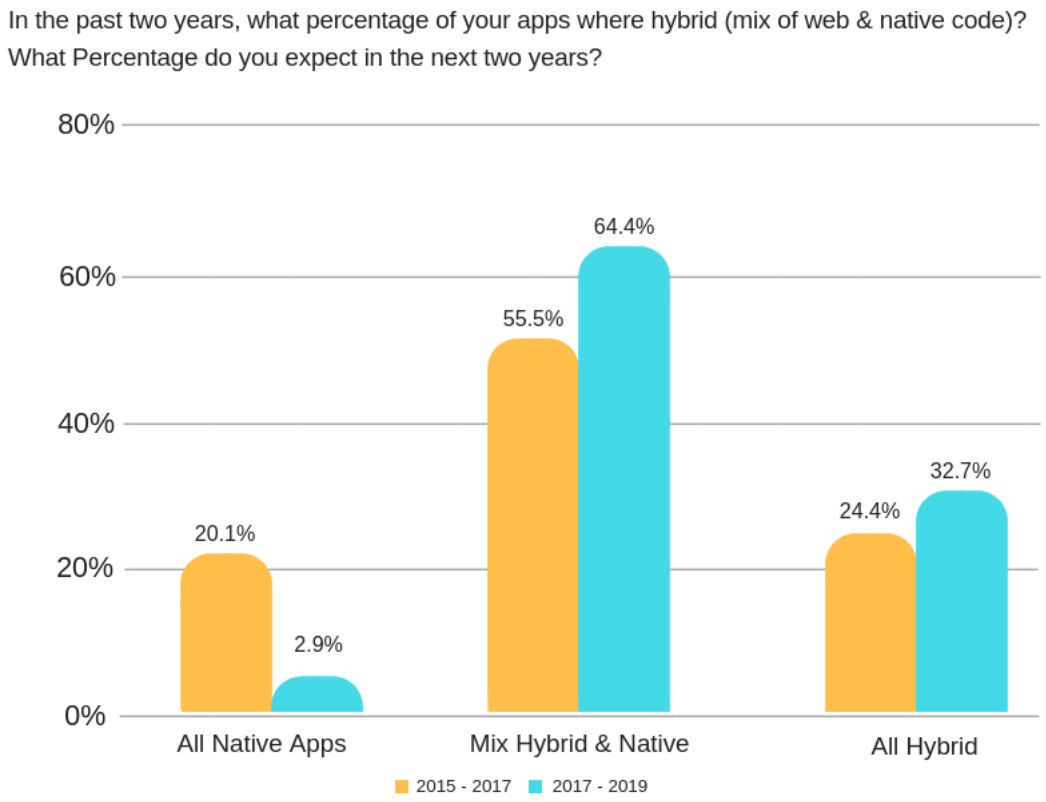


Figura 1.1: Percentuale delle applicazioni native e ibride presenti sul mercato.

1.2.1 Perchè implementare un'applicazione cross-platform

I motivi per cui orientarsi verso lo sviluppo di applicazioni utilizzando un'implementazione platform-independent sono molteplici:

- manutenibilità: è sicuramente il vantaggio più importante per quanto riguarda le risorse di tempo ma anche di costo risparmiate nel realizzare un'applicazione cross-platform. Realizzando infatti un unico codice per l'applicazione è più veloce mantenerlo, correggerlo, identificare malfunzionamenti
- rapidità di sviluppo: i tempi di realizzazione, dovendo scrivere un unico codice per tutte le piattaforme, si riducono notevolmente
- app ottimizzate quasi come le native: non si hanno notevoli perdite per quanto riguarda l'efficienza e la fluidità dell'applicazione
- investimento sul futuro: in ottica futura, per quanto rigurda gli sviluppi del mercato delle app, può essere una scelta vincente acquisire competenze rigurdanti i linguaggi cross-platform

Ulteriori motivazioni a supporto di questa scelta sono gli svantaggi causati dagli sviluppi implementativi delle altre tipologie di applicazioni [1]:

- per quanto rigurda le Progressive Web App, il rischio è quello di andare incontro ad utenti che utilizzano browser o sistemi operativi che non sono in grado di supportarle
- le app ibride hanno performance peggiori poichè richiedono l'installazione dell'applicazione sul dispositivo, ma l'effettivo funzionamento dell'app dipende dalla connessione internet.

Con Flutter nello specifico, è possibile ottenere una vera e propria app nativa con un unico linguaggio di programmazione. Alibaba, la più grande società di commercio

online del mondo, ha utilizzato Flutter per creare una app per iOS e Android, che ha 50 milioni di download [4].

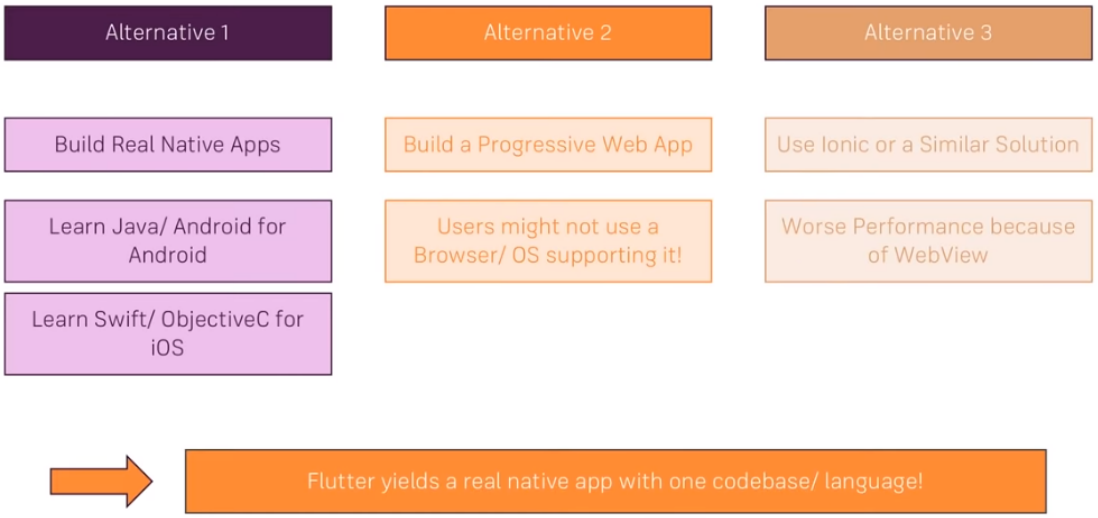


Figura 1.2: Alternative a Flutter: punti di debolezza

La figura 1.2 mostra i principali punti di debolezza delle applicazioni native, ibride e progressive web app, rispetto a quelle cross-platform.

1.2.2 Framework platform-independent

Ad oggi esistono già molti framework per lo sviluppo di applicazioni ibride che utilizzano linguaggi platform-independent come JavaScript o applicazioni cross-platform vere e proprie, ognuno con i suoi punti di forza. Essendo un mercato in espansione sono molti gli sviluppatori e le aziende che si interrogano verso quale framework sia meglio orientarsi. Di seguito verranno esposti i framework maggiormente utilizzati, facendone una panoramica ed esponendone pro e contro.

Apache Cordova

Apache Cordova [9] è uno dei framework più utilizzati nel mercato per lo sviluppo di applicazioni cross-platform [2]. Esso raggruppa HTML, CSS e JavaScript in un pacchetto unico lato client, quindi il software esegue e rende il codice personalizzato all'interno di una app ibrida nativa [10]. L'approccio ibrido fornisce una soluzione Write-Once-Run-Anywhere (WORA). Con il wrapping del codice Web in un pacchetto Cordova nativo, Cordova può fornire l'accesso alle API native e incorporando plugin sviluppati dalla community, le app possono connettersi a qualsiasi numero di queste API utilizzando semplicemente JavaScript. Punti di forza:

- essendo una app ibrida, vale il principio Write-Once-Run-Anywhere
- permette di ricevere sempre nuovi aggiornamenti dal team di Apache che ottimizzano il sistema operativo

Punti di debolezza:

- lascia tutte le decisioni di progettazione e architettura allo sviluppatore. Questo potrebbe essere un problema se è la prima volta che si lavora su un'applicazione mobile. Un sondaggio tra gli sviluppatori di StackOverflow mostra come molti si sentano insicuri a lavorare con Apache Cordova piuttosto che con altri framework [35].

1.2.3 Xamarin

Xamarin [33] è oggi largamente utilizzato per realizzare applicazioni platform-independent e permette di creare app con design pressochè identico a quelle native. Esso utilizza il C#, che è un linguaggio di programmazione molto conosciuto poichè utilizzato da molti anni da Microsoft. Essenziale nello sviluppo di un'applicazione cross-platform è il supporto nativo, che consente di rendere l'applicazione pressochè identica alla corrispettiva nativa. Xamarin supporta piattaforme multiple, come iOS, Android, Forms, macOS, watchOS, tvOS, e altre, che a loro volta dispongono di molti componenti e moduli per quanto riguarda l'interfaccia utente [31]. Punti di forza:

- utilizza il linguaggio C#, molto diffuso e molto conosciuto
- si basa su un sistema collaudato e molto efficiente che è Mono [44]
- essendo di proprietà di Microsoft ha una community molto forte in suo sostegno

Punti di debolezza:

- Xamarin fa uso di strumenti .NET i quali possono risultare molto ostici per i programmatori alle prime armi

Adobe PhoneGap

PhoneGap era all'origine la base per Cordova, ma ora è un prodotto standalone [39]. Adobe PhoneGap [21] è un wrapper che contiene e arricchisce le funzionalità di Cordova, ad esempio offre un sistema di costruzione GUI che astrae il caos della riga di comando di Apache Cordova. Fornendo un'interfaccia per creare applicazioni Cordova, PhoneGap semplifica il processo e consente agli sviluppatori di lavorare in maniera più efficiente diminuendo il time-to-market della applicazioni. Punti di forza:

- semplifica Apache Cordova fornendo un sistema di costruzione GUI e modelli di progetto per aiutare a dare il via allo sviluppo ibrido.
- Adobe PhoneGap ha il supporto continuo di Adobe e centinaia di sviluppatori open source.

Punti di debolezza:

- Adobe PhoneGap essendo un wrapper che contiene Cordova e lo espande, potrebbe tradursi in alcuni casi in prestazioni delle applicazioni scadenti.

Ionic

Ionic [26] combina Angular con la propria libreria UI per fornire un'esperienza di sviluppo di applicazioni mobili multipiattaforma [28]. Il processo di costruzione di applicazioni mobile in Ionic si basa su PhoneGap e pertanto eredita anche tutti i plug-in di Apache Cordova [27]. Punti di forza:

- Ionic è una soluzione ibrida multipiattaforma estremamente comoda perchè consente agli sviluppatori Web di continuare a utilizzare un framework front-end molto conosciuto: Angular. Incorporando Angular nel framework, Ionic facilita la transizione dallo sviluppo web a quello mobile. Inoltre supporta anche le applicazioni Web progressive eseguite in Chrome.

Punti di debolezza:

- come per gli altri framework di sviluppo di applicazioni ibride, le prestazioni delle app realizzate tramite Ionic possono non essere ottimali

React Native

React Native [24] eredita da React stesso, un tool per la creazione di applicazioni Web. React Native interpreta il codice sorgente e lo converte in elementi nativi [23]. Esso è senza dubbio la piattaforma più popolare per la realizzazione di applicazioni native mediante linguaggio JavaScript e deve la sua popolarità al suo fondatore: Facebook. Entrambe le applicazioni native di Facebook e Instagram sono realizzate con React Native. Punti di forza:

- React Native è testato in contesti estremamente ampi e ottimizzati. Facebook e Instagram hanno milioni di utenti attivi ogni giorno e questi richiedono prestazioni elevatissime. Finché tali prodotti si basano su React Native, gli sviluppatori possono aspettarsi che gli ingegneri di Facebook aggiornino regolarmente questo framework.

Punti di debolezza:

- React Native non è una soluzione WORA, richiede che gli sviluppatori personalizzino ogni implementazione dell'interfaccia per una specifica piattaforma. Ciò significa che è necessario creare sia una versione Android che una versione iOS dell'applicazione

- nonostante faccia affidamento su tecnologie web come JavaScript, l'interfaccia è un mix tra linguaggio di markup personalizzato e interfaccia utente nativa. Pertanto, gli sviluppatori non possono utilizzare le librerie CSS esistenti o i plugin jQuery.

NativeScript

NativeScript [22] è il concorrente diretto di React Native. Esso offre un'esperienza di sviluppo cross-platform simile al suo rivale supportato da Facebook e come React Native, NativeScript compila le applicazioni JavaScript trasformandole in native [7]. Promuove Angular 2 come framework applicativo, ma gli sviluppatori lo possono disattivare e utilizzare JavaScript standard con le API NativeScript. Consente inoltre agli sviluppatori di avere un'idea del prodotto finale fornendo applicazioni di dimostrazione dell'interfaccia per Android e iOS. Per entrambe le piattaforme, le applicazioni vengono generate da un singolo codebase, dimostrando che NativeScript è una soluzione WORA in grado di raggiungere alte prestazioni su entrambe le piattaforme. Punti di forza:

- NativeScript offre le stesse prestazioni rispetto alle soluzioni ibride creando applicazioni native da JavaScript.

- è una soluzione WORA

- NativeScript si integra anche direttamente con Gradle e CocoaPods [8], consentendo così agli sviluppatori di incorporare librerie native Swift, Objective-C e Java nei loro progetti NativeScript.

Punti di debolezza:

- rispetto a React Native, non si conosce la robustezza delle applicazioni realizzate con NativeScript, con milioni di utenti giornalieri come Instagram e Facebook

- come React Native, NativeScript usa un markup personalizzato per progettare le sue interfacce, nessun CSS3 o HTML5.

1.3 Alternative a JavaScript

Come spiegato, la maggior parte delle applicazioni non native sono ibride, realizzate mediante l'utilizzo di linguaggi platform-independent come HTML5 e JavaScript. In letteratura però JavaScript non è nato per svolgere tale compito e presenta problemi strutturali, come ad esempio il fatto che non è un linguaggio object-oriented. A fronte di ciò sono nati nuovi linguaggi con lo scopo di estendere le potenzialità di JavaScript e più adatti alla realizzazione di pagine web, ma anche di applicazioni mobile platform-independent. Due linguaggi degni di nota sono Dart e TypeScript. Il primo è il linguaggio che è utilizzato in Flutter, mentre il secondo è il diretto discendente di JavaScript.

1.4 Panoramica su TypeScript

TypeScript è un super-set di JavaScript, ovvero ne estende la sintassi, permettendo a qualunque programma realizzato in JavaScript di funzionare anche con TypeScript senza il bisogno di effettuare alcuna modifica [49]. Tale linguaggio è nato dalla necessità di rendere più robusto e type safe JavaScript ed infatti introduce molti nuovi meccanismi importanti come:

- tipizzazione per i tibi base, come Boolean, Number e String

- classi ed interfacce, rendendo il linguaggio di fatto object-oriented e non più object-based, permettendo quindi di utilizzare i meccanismi tipici dei linguaggi orientati agli oggetti come l'ereditarietà

- mixin tra classi, che permette di costruire classi a partire da componenti riutilizzabili, combinando classi parziali più semplici [30].

Capitolo 2

Il linguaggio Dart

Dart è un linguaggio di programmazione di proprietà di Google, nato per lo sviluppo di applicazioni Web nel 2011. L'intento era quello di creare un linguaggio che potesse sostituire JavaScript, in quanto quest'ultimo, per caratteristiche strutturali, non sarebbe potuto essere migliorato ulteriormente. Dart ebbe un discreto successo, ma non riuscì ad avere sul mercato gli effetti sortiti. Google ha quindi deciso di impiegarlo come linguaggio per lo sviluppo di applicazioni mobile platform-independent all'interno del framework Flutter. L'applicazione citata nel caso di studio dunque ha richiesto l'utilizzo di Dart come linguaggio di programmazione.

2.1 Panoramica sul linguaggio

Dart è un linguaggio estremamente versatile in quanto multi paradigma. Nato come linguaggio object-oriented abbraccia anche la programmazione funzionale, implementa meccanismi di riflessione e come Java fa uso di garbage collector [42]. Dart è fortemente tipizzato e type safe, ossia utilizza una combinazione di controllo di tipo statico e controlli runtime ed è possibile perchè Dart permette sia la compilazione AOT(compilazione anticipata) che JIT(compilazione just-in-time) [18]. Questi meccanismi garantiscono che il valore di una variabile corrisponda sempre al tipo statico della stessa, fornendo solidità al codice. Presenta inoltre meccanismi di inferenza.

2.1.1 Vantaggi del typing

Una caratteristica che constraddistingue Dart rispetto a JavaScript è il fatto di essere un sistema fortemente tipizzato. Un sistema type safe presenta numerosi vantaggi:

- rilevazione dei bug al momento della compilazione, poichè un sistema solido obbliga il codice a non presentare ambiguità circa i suoi tipi, di conseguenza i bug relativi al tipo che potrebbero essere difficili da trovare, vengono rivelati al momento della compilazione
- codice più leggibile, infatti il codice è più facile da leggere in quanto ogni variabile ha il suo tipo specificato
- codice più gestibile, in quanto modificando una porzione di codice, il sistema avvisa il programmatore nel caso ci siano altre porzioni che, in seguito a tali modificazioni, incorrono in errore

2.1.2 L'event loop di Dart

Dart possiede un modello a singolo thread basato su architettura event loop [37]. Un event loop è una architettura asincrona che permette di inserire gli eventi generati in una coda e gestirli uno alla volta estraendoli dalla coda ed eseguendo il rispettivo handler.

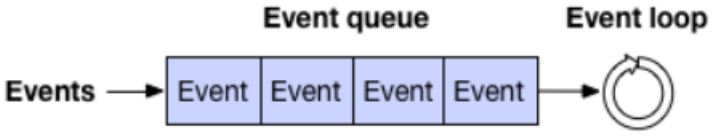


Figura 2.1: Architettura event loop

Ogni qualvolta si verifica una interruzione, che può essere scatenata dall'interazione con l'utente, da un timer, dalla risposta a una chiamata di rete, l'event loop aggiunge alla coda l'evento generato e successivamente lo estrae dalla coda per eseguirlo. Questo meccanismo va avanti finchè ci sono eventi in coda, altrimenti attende la notifica di nuovi eventi. Anche JavaScript si basa su una architettura

analoga [20]. Una caratteristica di questa architettura è quella di non poter eseguire compiti bloccanti all'interno degli handler degli eventi che sono processati, per cui si rendeva necessario in JavaScript l'utilizzo di numerose callbacks per gestire gli eventi, andando incontro al problema chiamato "nested callbacks", ovvero molte callbacks innestate che rendono il codice meno leggibile. Dart e anche le ultime versioni di JavaScript implementano le parole chiave **async** e **await** che permettono di ovviare a questo problema. Inoltre Dart è un modello a singolo thread, ma è possibile creare sezioni di codice concorrente mediante l'uso degli Isolates, ovvero oggetti che hanno una propria memoria e che comunicano tra loro mediante scambi di messaggi. Quando viene avviata un'applicazione Dart, viene creato il suo Main Isoltate che esegue la funzione main(). In seguito il thread del Main Isolate comincia a gestire gli eventi in coda, uno alla volta.

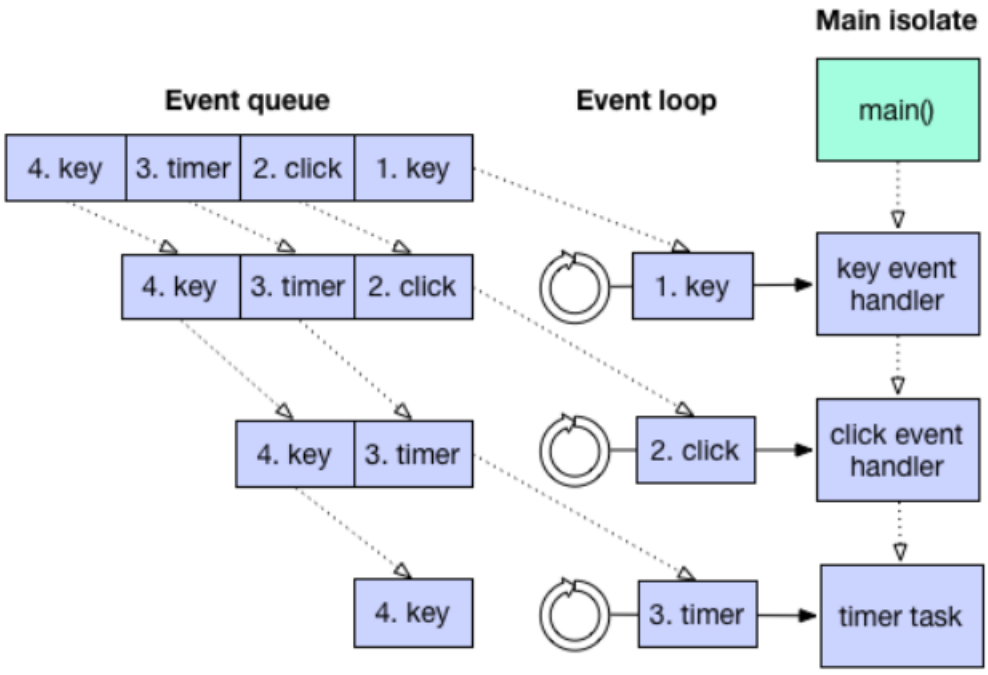


Figura 2.2: Modello a singolo thread di Dart

Oltre alla coda di eventi dell'event loop, Dart possiede anche quella che viene chiamata "coda di microtask", ovvero task molto brevi necessari perché a volte il codice di gestione degli eventi deve completare un'attività in un secondo momento,

ma prima di restituire il controllo all'event loop. Ad esempio, quando un oggetto osservabile cambia, effettua diverse modifiche in modo asincrono. La coda di microtask consente all'oggetto osservabile di segnalare queste modifiche prima che il modello a oggetti possa mostrare uno stato incoerente. Mentre l'event loop sta eseguendo un'attività dalla coda di microtask, la coda degli eventi è bloccata.

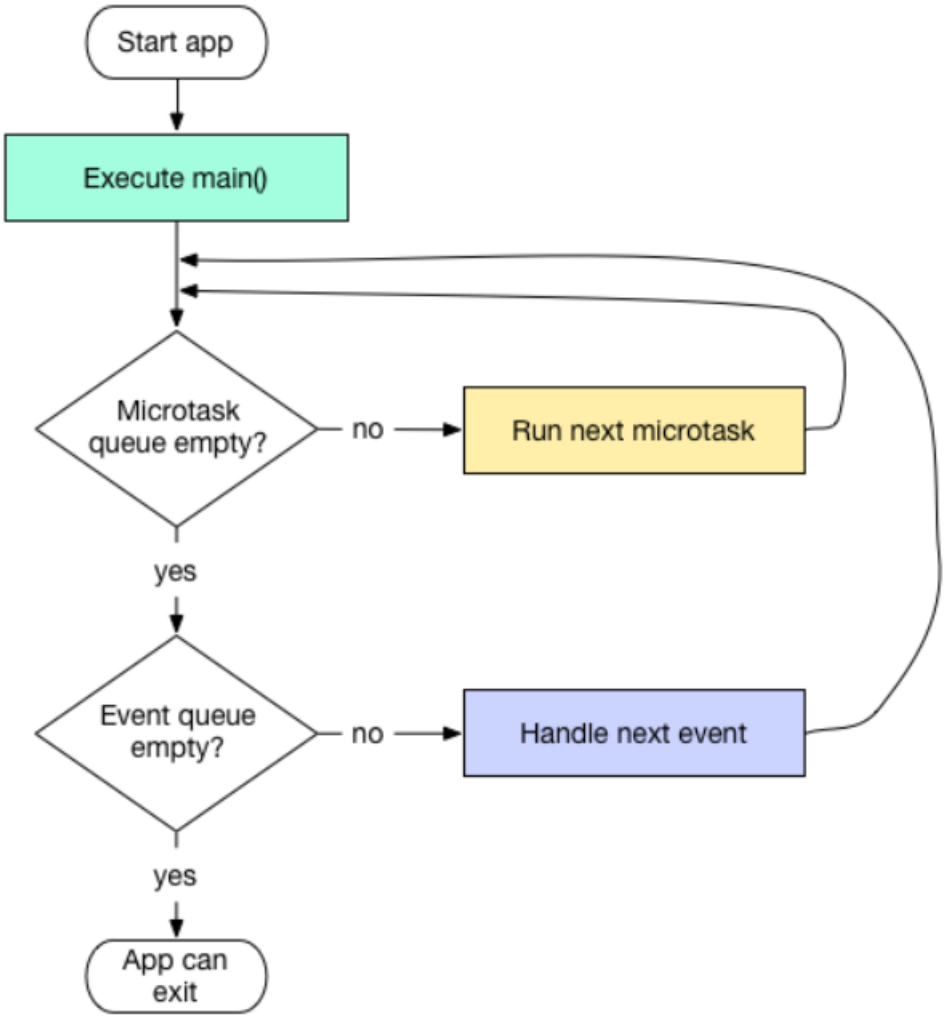


Figura 2.3: Funzionamento dell'event loop in una applicazione Dart

2.1.3 Gestore di pacchetti Pub

Pub è uno strumento per la gestione dei pacchetti per il linguaggio Dart. Si tratta di un vero e proprio raccoglitore di plugin open source che possono essere poi utilizzati

nei progetti [16]. Vengono messi a disposizione pacchetti sia per Dart Web che per la parte relativa a Flutter. Qualsiasi applicazione Dart fa largo uso di Pub per l'inclusione di pacchetti all'interno dell'applicazione stessa.

2.2 Principali meccanismi

2.2.1 dynamic

Dart è un linguaggio fortemente tipizzato ed è dotato di inferenza. Quando però esplicitamente non è utilizzato alcun tipo viene implicitamente utilizzato il tipo dynamic. In Dart ogni cosa è un oggetto che deriva dalla classe Object, un dynamic invece rappresenta un tipo più complesso. L'uso di dynamic può stare a significare che è necessario rappresentare un tipo di dato non rientrante tra quelli consentiti o comunque al di fuori della portata del sistema di tipi statici, o che si desidera esplicitamente il dinamismo a runtime per quel dato. Concretamente significa che il sistema smette di applicare il controllo sul dato dynamic. Se si dichiara un dato di tipo Object, quando si prova a chiamare su quel dato un metodo non definito per il tipo Object, il sistema avvisa che si sta commettendo un errore. Sul tipo dynamic invece è possibile chiamare qualsiasi metodo, anche se questo creerà un errore a runtime, poichè il sistema smette di applicarvi il meccanismo di controllo statico.

2.2.2 Future

Come spiegato nel capitolo 2.1.2, Dart possiede un modello a singolo thread basato su event loop. Del codice bloccante eseguito sul thread comporterebbe il blocco dello stesso. É possibile quindi fare uso della programmazione asincrona per eseguire operazioni in modo asincrono. Il tipo Future<T> è un'operazione asincrona che produce un risultato di tipo T. Se il Future non produce risultato, ma è semplicemente necessario effettuare un'operazione asincrona, si utilizza Future<void>. Quando viene invocata una funzione che restituisce un Future accadono due cose:

1. la funzione accoda le operazioni da eseguire

2. al termine delle operazioni il Future viene completato con il risultato o con un errore

Come in JavaScript, anche Dart fa uso di callbacks per poter utilizzare il risultato di un Future.

2.2.3 Async e Await

Le parole chiave async e await sono di supporto alla programmazione asincrona. Async viene utilizzata nella signature delle funzioni che restituiscono un Future o che ne fanno uso al loro interno, mentre await permette letteralmente di "aspettare" che il Future possa restituire un risultato. Await non è bloccante, semplicemente permette all'event loop di proseguire con gli eventi in coda e quando l'azione long-running è terminata allora l'event loop riprenderà l'esecuzione del Future che ha prodotto il risultato, permettendo così di non fare uso delle callbacks.

2.2.4 Stream

Gli Stream in Dart rappresentano, assieme ai Future, un'importante astrazione per la programmazione asincrona. Essi sono sequenze di eventi asincroni, che restituiscono un evento quando questo è terminato. Il contenuto di uno Stream può essere elaborato mediante la parola chiave await o la callback listen(), la quale consente a un Listener di ascoltare lo Stream [13]. Prima di utilizzare questo metodo, lo Stream è un oggetto inerme che contiene al suo interno degli eventi. Nel momento in cui si utilizza il metodo listen() viene restituito un oggetto di tipo StreamSubscription che è lo stream attivo che produce eventi ed è possibile iniziare ad estrarre gli eventi contenuti al suo interno.

Gli Stream si suddividono in due categorie:

- single subscription stream, in cui gli eventi vengono restituiti tutti e in ordine. É la tipologia utilizzata quando ad esempio si legge un file o si ricevono dati in seguito ad una chiamata Web. Tale stream però può essere ascoltato da un solo Listener un'unica volta, poichè è possibile che un ascolto successivo comporti la perdita degli eventi iniziali.

- broadcast stream, tipologia destinata ai singoli messaggi che possono essere gestiti uno alla volta, come ad esempio gli eventi generati dal mouse. Più Listener possono mettersi in ascolto sul medesimo Stream di questa categoria e ci possono essere ascolti successivi senza questo comporti la perdita di eventi

Quando viene attivato un evento, i Listener collegati riceveranno istantaneamente l'evento. Se un Listener viene aggiunto a uno Stream Broadcast mentre viene attivato un evento, quel Listener non riceverà l'evento attualmente in corso di esecuzione mentre se un Listener viene cancellato, interrompe immediatamente la ricezione degli eventi.

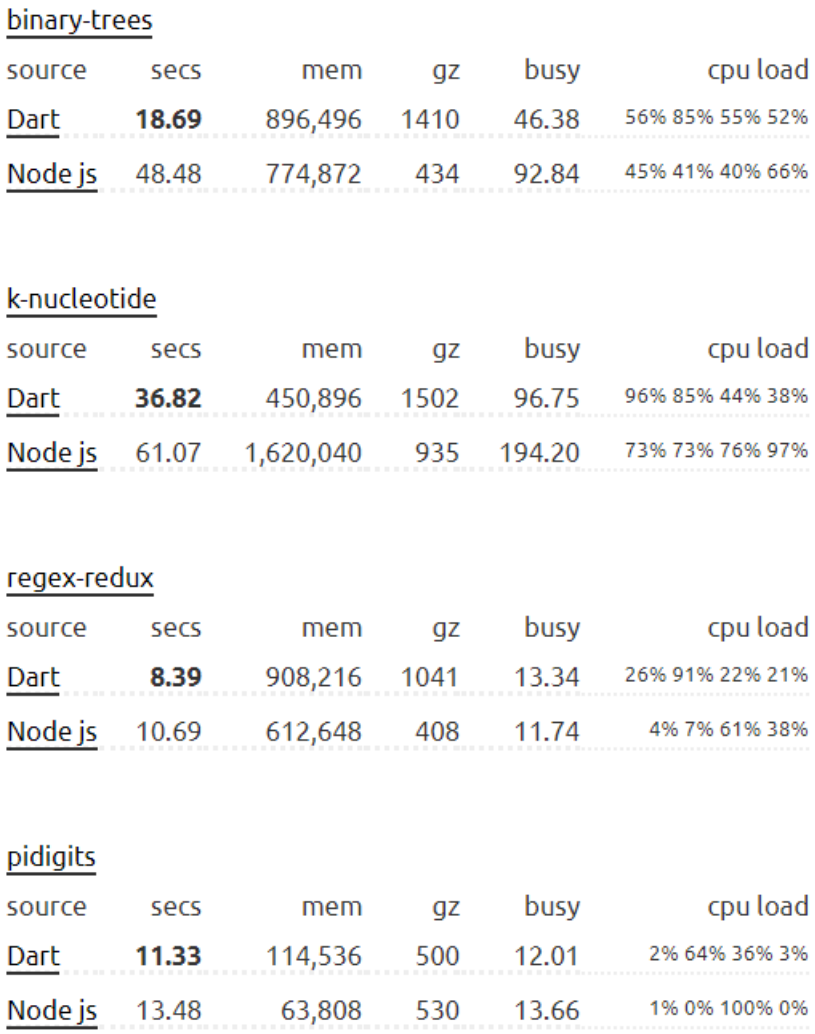
2.3 La scelta di Dart

2.3.1 Introduzione a JavaScript

JavaScript è un linguaggio object-based [38] basato su architettura event loop, utilizzato nello sviluppo delle applicazioni Web, specialmente lato client. Non è un linguaggio compilato ma interpretato ed è debolmente tipizzato ossia non ha tipi specifici di dati come possono essere gli interi o le stringhe.

2.3.2 Vantaggi di Dart

Dart è un lingauggio object-oriented, supporta quindi istanze di veri e propri oggetti e per questo motivo è preferito dalla scuola di pensiero più legata all'ingegneria del software. É un linguaggio compilato, caratteristica che permette di trovare la maggior parte degli errori di programmazione in fase di compilazione ed essendo type-safe è più sicuro dal punto di vista della tipizzazione rispetto a JavaScript, rendendo più semplice il debugging delle applicazioni. Inoltre JavaScript è un linguaggio interpretato, caratteristica strutturale che ne riduce le prestazioni. Dart ha infatti dimostrato di essere molto più veloce quando confrontato con JavaScript [11].



binary-trees

source	secs	mem	gz	busy	cpu load
Dart	**18.69**	896,496	1410	46.38	56% 85% 55% 52%
Node js	48.48	774,872	434	92.84	45% 41% 40% 66%

k-nucleotide

source	secs	mem	gz	busy	cpu load
Dart	**36.82**	450,896	1502	96.75	96% 85% 44% 38%
Node js	61.07	1,620,040	935	194.20	73% 73% 76% 97%

regex-redux

source	secs	mem	gz	busy	cpu load
Dart	**8.39**	908,216	1041	13.34	26% 91% 22% 21%
Node js	10.69	612,648	408	11.74	4% 7% 61% 38%

pidigits

source	secs	mem	gz	busy	cpu load
Dart	**11.33**	114,536	500	12.01	2% 64% 36% 3%
Node js	13.48	63,808	530	13.66	1% 0% 100% 0%

Figura 2.4: Confronto con Node.js nell'ambito della velocità di esecuzione di alcuni programmi di problem solving.

2.4 Confronto tra Dart e TypeScript

Utile a comprendere meglio le caratteristiche di Dart può essere il confronto con un linguaggio a lui più simile come TypeScript, che è per altro un suo competitor diretto tra i linguaggi per lo sviluppo di applicazioni Web.

2.4.1 Introduzione a TypeScript

TypeScript è un linguaggio di programmazione open source, nato dall'esigenza di avere un linguaggio diffuso come JavaScript che però sia tipizzato e object oriented.

É sviluppato e gestito da Microsoft e poiché TypeScript è un superset di JavaScript, tutti i programmi JavaScript esistenti sono anche programmi TypeScript validi. É un linguaggio fortemente tipizzato, orientato agli oggetti e compilato, progettato per lo sviluppo di applicazioni di grandi dimensioni che poi compila come applicazioni JavaScript.

2.4.2 Differenze fondamentali

I due linguaggi si differenziano per le loro caratteristiche, in particolare:

1. essendo TypeScript derivato dal JavaScript, è semplice per gli sviluppatori JavaScript la migrazione verso TypeScript, mentre Dart risulta un linguaggio piuttosto diverso da apprendere

2. al contrario di Dart che è un linguaggio di programmazione vero e proprio, TypeScript è un superset per JavaScript

3. TypeScript ha una tipizzazione statica facoltativa basata sulle annotazioni, ovveri il tipo della variabile viene dichiarato come :Tipo di seguito [29] mentre Dart ha una tipizzazione tradizionale

4. TypeScript supporta ogni applicazione scritta in JavaScript ed essendo questo il linguaggio più utilizzato in ambito web non ha problemi di compatibilità con la maggioranza delle applicazioni web esistenti

2.4.3 Conclusioni

Sia Dart che TypeScript sono indubbiamente due linguaggi molto robusti per via delle loro caratteristiche. Se si è orientati alla realizzazione di un'applicazione web può essere una scelta vincente orientarsi verso TypeScript per via del suo supporto a JavaScript e la solidità che deriva da quest'ultimo in ambito web. Dart è un linguaggio che sta maturando con Flutter e per questo più indicato per lo sviluppo di applicazioni mobile mediante il medesimo framework.

2.5 Conclusioni

In questo capitolo è stato mostrato come Dart sia un linguaggio nato per sostituire JavaScript in principio per la creazione di pagine Web, ma oggi anche per la scrittura di applicazioni platform-independent. Esso si ispira a JavaScript per quanto riguarda la sua architettura event loop e lo estende introducendo un modello object-oriented e abbracciando la tipizzazione statica, oltre ad essere migliore dal punto di vista delle prestazioni. É a fronte di queste considerazioni che Google ha deciso di integrare Dart all'interno del framework Flutter piuttosto che utilizzare JavaScript o TypeScript.

Capitolo 3

Il framework Flutter

Flutter è una tecnologia di Google, un framework che permette di codificare un'applicazione in un unico linguaggio e renderla funzionante sia su dispositivi Android che su dispositivi iOS. L'obiettivo di Google era quello di creare un framework leggero, intuitivo ed efficiente che semplificasse la scrittura delle app mobile e che fosse facilmente integrabile in ambienti di lavoro come Android Studio e Visual Studio Code. É quindi stato utilizzato Flutter come framework per la realizzazione del caso di studio e Dart, descritto precedentemente, è il linguaggio di programmazione utilizzato dal framework stesso.

3.1 Concetti fondamentali

Flutter è basato principalmente sulle Widgets [6] ossia elementi visuali che possono modificarsi dinamicamente in base a logiche implementative. Inoltre utilizzando Dart come linguaggio, il framework possiede tutti i suoi meccanismi asincroni per i compiti long-running, come ad esempio una chiamata di rete. Per rendere possibile la scrittura di applicazioni mobile cross-platform, Flutter mette a disposizione delle astrazioni che si comportano in modo analogo o quasi a quelle native di Android e iOS. Di seguito nel capitolo saranno spiegate le analogie e le differenze con le applicazioni native e come è possibile ripensare e reingegnerizzare le applicazioni scritte per Android o iOS in termini di astrazioni di Flutter.

3.1.1 Struttura di Flutter

Flutter ha una struttura a strati che consente di poter fare più cose scrivendo meno codice, oltre che semplificando notevolmente ciò che un programmatore deve scrivere [19]. Si divide in tre strati:

- strato framework: è il primo strato, quello più superficiale, fa uso del linguaggio Dart, il quale mette a disposizione librerie ed astrazioni per lo sviluppo delle applicazioni mobile

- strato engine: utilizza il C++ per rendere molto efficienti e veloci le applicazioni realizzate con Flutter rispetto a linguaggi interpretati

- strato embedder: è lo strato più profondo ed interviene sul device a livello nativo

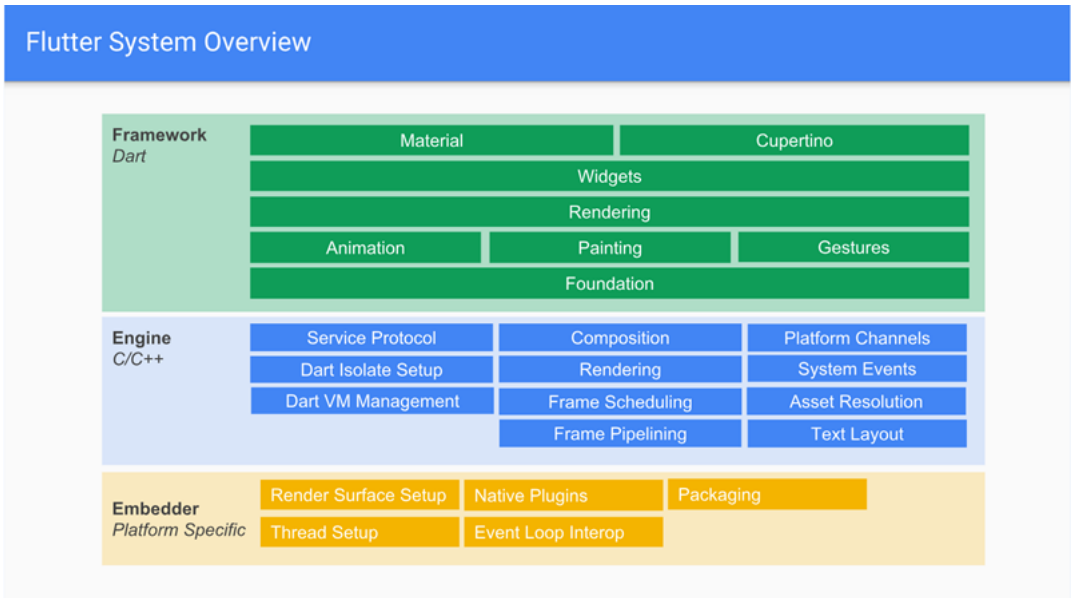


Figura 3.1: Struttura a strati di Flutter

3.1.2 Motore grafico Skia

Flutter basa i suoi meccanismi di rendering su Skia, una libreria grafica 2D open-source scritta in C++. Essa fornisce API comuni che funzionano su una grande varietà di piattaforme hardware e software. Per citarne alcune, è utilizzata come motore grafico per Google Chrome e Chrome OS, Android, Mozilla Firefox e Firefox OS e molti altri prodotti [34]. Skia ha diversi back-end, incluso uno per la rasterizzazione del software basato su CPU, uno per l'output PDF e uno per OpenGL, altra libreria grafica scritta in C++, accelerato da GPU [48].

3.1.3 Widget

In Flutter ogni elemento visuale è una Widget. Le Widget rappresentano gli oggetti che compaiono a schermo, con cui l'utente può o meno interagire. Tali astrazioni vanno intese come una descrizione dell'interfaccia utente, che vengono poi istanziate come componenti reali della stessa. Esse sono immutabili, ossia una volta istanziate non possono cambiare. Esistono due tipologie di Widget:

- Stateless, ovvero non mantengono informazioni sul loro stato;
- Stateful, le quali mantengono informazioni sul loro stato.

Le Stateful Widget, mantenendo informazioni sullo stato, nonostante siano immutabili possono essere modificate. Quando viene modificato il loro stato Flutter crea un nuovo albero di istanze delle Widget sostituendo quelle che hanno subito modifiche. Questa tipologia viene utilizzata quando sono necessarie modifiche a runtime della Widget in seguito ad un evento, come l'interazione con l'utente o la ricezione dei dati come conseguenza di una chiamata HTTP. Le Stateless Widget invece, non mantenendo informazioni sul loro stato, sono a tutti gli effetti immodificabili e vengono quindi utilizzate per elementi visuali che non devono mai cambiare, come può essere un'immagine o un'icona.

Il layout grafico viene impostato direttamente nelle Widget, semplificando notevolmente la scrittura di codice, ma originando così un problema che riguarda la riusabilità e la modularità del codice stesso: è molto complesso separare la logica implementativa da aspetti puramente grafici.

Generalmente le Widget non mettono a disposizione metodi per modificare l'albero delle Widget, ma è possibile farlo passando le Widget come parametri a seconda di flag booleani. Ad esempio se si vuole mostrare una Widget quando il flag è true e niente quando il flag è false, si può utilizzare un operatore ternario che ritorna la Widget che si vuole o un Container vuoto mediante l'utilizzo del flag. Un Container vuoto è una Widget che semplicemente non ha nessuna visibilità a schermo, e quindi è come non mostrare nulla. Non è possibile ritornare null al posto di una Widget in quanto le Widget, per implementazione, non possono essere null. Questo comportamento causerebbe un errore a runtime.

```
body: this.flag ? Center() : Container(),
```

Figura 3.2: Metodo corretto per modificare l'albero delle Widget mediante l'uso di un Container vuoto.

Metodo setState()

Nelle Stateful Widget il metodo setState() comanda a Flutter di ridisegnare l'albero delle Widget con le modifiche scritte all'interno del metodo stesso. Di conseguenza se si vuole modificare una Stateful Widget, la logica implementativa deve essere all'interno della stessa, nel metodo setState().

Per ovviare a questo problema è necessario fare uso della programmazione reattiva, spiegata nel capitolo 4.

Metodo build()

Il metodo build() è presente sia nelle Stateless che nelle Stateful Widget ed è il cuore centrale delle Widget. Al suo interno è definita la Widget stessa, il suo aspetto grafico e la sua logica. Tale funzione restituisce l'albero delle Widget. Nel caso delle Stateful Widget ogni volta che viene invocato il metodo setState() e avvengono

cambiamenti nel suo stato, viene richiamato anche il metodo build() per ridisegnare la Widget apportandole le modifiche specificate.

Per personalizzare una Widget è possibile crearne una nuova come combinazione di altre preesistenti a cui si fornisce un comportamento differente.

3.1.4 Flutter predilige la composizione

Le Widget possono essere composte a partire da Widget più semplici che si combinano per produrre gli effetti desiderati. Flutter infatti è appositamente progettato per prediligere la composizione rispetto all'ereditarietà. Ad esempio, un Container, una Widget comunemente utilizzata, è composto da diverse Widget responsabili del layout, della posizione e della dimensione dello stesso. Invece di creare una sottoclasse di Container per produrre un effetto personalizzato, è possibile comporre le Widget.

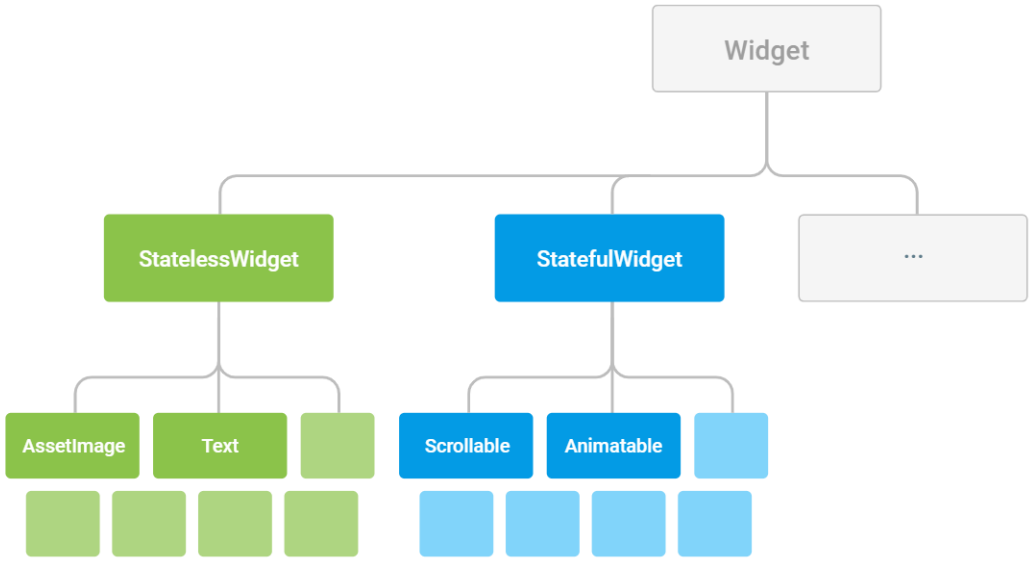


Figura 3.3: Gerarchia delle Widget e Widget composte.

3.1.5 Navigator e Route

In Flutter per navigare tra le schermate della stessa applicazione si utilizzano i Navigator e le Route. Una Route è un'astrazione per una schermata di un'app mentre un Navigator è una Widget che gestisce le Route [17]. La classe Navigator mette a disposizione diversi metodi, i due più importanti sono push() e pop(), per gestire il cambiamento di schermata. Con il metodo push() è possibile aggiungere una nuova schermata al percorso da navigare, mentre con pop() è possibile risalire a ritroso alle schermate precedenti. Navigator funziona quindi come una pila. Ci sono due modi possibili per muoversi tra le schermate in Flutter: specificare una Map che contiene i nomi delle Route oppure utilizzare i Navigator per muoversi direttamente al percorso specificato. La classe Navigator permette il passaggio da una schermata all'altra con il metodo push() e immagazzina il risultato passato alla schermata precedente con il metodo pop(). Il metodo pop() ritorna un Future, ossia sfrutta la programmazione asincrona. Si utilizza quindi la parola chiave await per attendere il risultato. Nella prima schermata è presente un codice simile al seguente:

```
String text = await Navigator.of(context).pushNamed('/location');
```

Figura 3.4: Navigator push

E nella seconda schermata, ottenuta in seguito alla chiamata a Navigator, si ha un codice simile a:

```
Navigator.of(context).pop({"text"});
```

Figura 3.5: Navigator pop

Questo codice indirizza a una nuova schermata mediante il metodo pushNamed() e quando nella seconda schermata viene chiamato il metodo pop() restituisce alla prima la stringa "text" che viene assegnata al campo String text.

3.1.6 Interfaccia utente asincrona

Come spiegato nel capitolo 2.2.2, Dart possiede un modello di esecuzione a singolo Thread, un event loop e supporta la programmazione asincrona. Il codice Dart viene eseguito nell'interfaccia utente principale ed è gestito dall'event loop. Il modello a singolo Thread non implica che tutto il codice debba essere eseguito come un'unica operazione, in quanto genererebbe un blocco dell'interfaccia utente. Flutter consente di utilizzare le funzionalità asincrone che il linguaggio mette a disposizione su di esso, come async e await. Un tipico compito asincrono è effettuare una chiamata di rete e in Flutter è molto semplice grazie all'integrazione del plugin http. Esso astrae gran parte del codice utile per il networking, rendendo così efficiente effettuare chiamate di rete. Per mostrare i progressi di un long-running task si utilizza una Widget chiamata ProgressIndicator che mostra i progressi effettuati controllando quando viene eseguito il rendering tramite un flag booleano. Flutter aggiorna il suo stato prima dell'avvio del task mostrando il ProgressIndicator e lo nasconde dopo che è il medesimo task è terminato.

3.1.7 Ciclo di vita della applicazioni

In Flutter è possibile ascoltare gli eventi del ciclo di vita mediante una Widget observer WidgetsBinding che ascolta la callback didChangeAppLifecycleState() richiamata quando si verifica una evento in seguito ad un cambiamento. Gli eventi osservabili nel ciclo di vita sono:

- inactive: l'applicazione si trova in uno stato inattivo e non riceve input dall'utente. Questo evento funziona solo su iOS in quanto non esiste un equivalente evento mappabile su Android;
- paused: l'applicazione non è momentaneamente visibile all'utente, non risponde ai suoi input ed è eseguita in background. É l'equivalente di onPause() in Android;
- resumed: l'applicazione visibile e risponde all'input dell'utente. Corrisponde a onPostResume() di Android;

- suspending: l'applicazione è momentaneamente sospesa. Equivale a onStop() in Android, mentre in iOS non c'è un evento equivalente.

Flutter mette a disposizione pochi metodi per il controllo del ciclo di vita di una Activity. Questo perchè ci sono poche ragioni per osservare il ciclo di vita lato Flutter e quindi tali metodi vengono nascosti al programmatore, vengono eseguiti internamente dal framework. Nel caso si renda strettamente necessario osservare il ciclo di vita per acquisire o rilasciare risorse, conviene farlo sul lato nativo [12].

3.1.8 Gestione degli eventi a schermo

Flutter rende possibile la gestione degli eventi touch in due modi differenti:

- parametri delle Widgets, ossia alcune Widgets mettono a disposizione tra i loro parametri metodi come onPressed() o onTap() che permetto l'handling degli eventi rilevati dallo schermo;

- uso di GestureDetector che è una vera e propria Widget che può incapsulare altre Widgets e che possiede il metodo onTap(). In questo modo le Widgets interne reagiscono all'evento scatenato dall'onTap() del GestureDetector;

Mediante l'utilizzo di GestureDetector è possibile gestire molti eventi. Le principali categorie sono Tap, Double Tap, Long Pressed e Drag orizzontale e verticale.

Gli eventi onTap identificano il tocco dell'utente sullo schermo e sono:

- onTapDown: un Tap avvenuto sullo schermo in una particolare posizione;

- onTapUp: un Tap rapido che si è concluso;

- onTap: un generico Tap;

- onTapCancel: un precedente tocco che ha attivato l'onTapDown non causa nessun Tap;

L'evento onDoubleTap identifica un doppio tocco in rapida successione sulla stessa zona dello schermo da parte dell'utente.

L'evento onLongPressed identifica un tocco prolungato dell'utente.

Gli eventi VerticalDrag e HorizontalDrag indentificano un trascinamento dello schermo da parte dell'utente.

3.1.9 Libreria Material

Material è un design sviluppato da Google i cui layout sono ottenuti mediante l'uso di griglie, animazioni, transizioni ed effetti di profondità come le ombre. La libreria Material contiene questi layout ed è il cuore centrale di Flutter. Le Widget native del framework infatti implementano in gran parte la libreria Material, la quale conferisce loro effetti visivi molto realistici. L'idea che sta alla base di questo design è quello di avere "carta e inchiostro digitali" che possono modificarsi in modo differente dai corrispettivi reali [43]. É così possibile ottenere Widget con effetti e animazioni personalizzati modificando tali layout. Siccome il design Material è utilizzato da Google all'interno dei suoi prodotti è arcinoto, facilmente riconoscibile e questo conferisce alle applicazioni realizzate in Flutter un aspetto familiare. Per le applicazioni Android native, il Material design è compatibile solo dalla versione di Android 5.0.

3.1.10 Libreria Cupertino

Cupertino è una libreria che rende la grafica delle applicazioni realizzate con Flutter identica a applicazioni iOS native. Essa infatti contiene al suo interno l'implementazione di Widget vere e proprie, corrispettive a quelle delle libreria Material, ma con l'estetica dei sistemi iOS.

3.1.11 pubspec.yaml

Il file pubspec.yaml si trova in tutti i progetti Flutter e contiene al suo interno tutte le dipendenze dell'applicazione e i path delle icone, delle immagini e i font. Se si vuole importare nel progetto un plugin di Pub è necessario aggiungerlo alla lista delle dipendenze, ad esempio se si vuole importare il plugin che implementa delle animazioni per lo scorrimento delle Widget si utilizza il seguente codice:

```
dependencies:
  page_transition: ^1.0.9
```

Figura 3.6: Inserimento di una dipendenza

All'interno del file pubspec.yaml tutto ciò che è indentato al di sotto della parola chiave "dependecies" viene incluso nel progetto senza la necessità di effettuare il download in locale.

3.2 Widget principali

In Flutter esistono diverse tipologie di Widget, ognuna è specializzata in un determinato compito ed utile in un preciso contesto. É possibile ottenere Widget più complesse combinando le Widget che il framework mette a disposizione. Di seguito è riportata una panoramica sulle principali Widgets di Flutter, le quali sono state utilizzate anche nel caso di studio per la realizzazione dell'estetica dell'applicazione.

3.2.1 Scaffold

Uno Scaffold è una Widget che implementa il layout base del Material Design. Esso può essere inteso come "la schermata di fondo", lo sfondo dello schermo che si espande per occupare tutto lo spazio disponibile. Quando è necessario utilizzare la tastiera, ad esempio per scrivere all'interno di campi TextField, lo Scaffold si ridimensiona automaticamente per lasciare posto alla stessa. Tale Widget accetta nel costruttore vari parametri, i più importanti sono:

- appBar, che accetta una PreferredSizeWidget e disegna la barra in alto nella schermata
- body, che accetta qualsiasi Widget e disegna il corpo centrale dello Scaffold
- floatingActionButton, che permette di disegnare bottoni in posizioni dello schermo indicate

- bottomNavigationBar, che accetta una Widget e si occupa di posizionarla in basso nello schermo

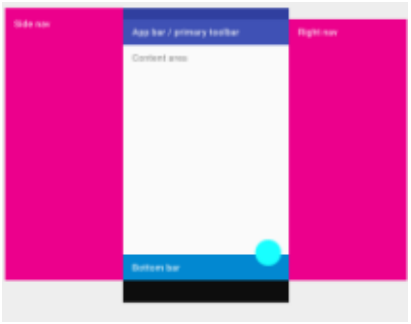


Figura 3.7: Widget Scaffold

3.2.2 AppBar

Una AppBar rappresenta una Widget che viene collocata nella parte superiore dello schermo. Essa generalmente permette di esprimere operazioni mediante l'utilizzo di icone. Si compone di tre campi principali:

- leading: accetta una Widget che viene posizionata prima del title
- title: accetta una Widget che viene posizionata nella parte centrale della AppBar
- actions: accetta una lista di Widget che saranno posizionate nella zona successiva al title

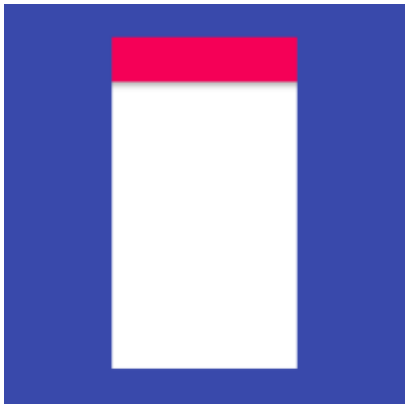


Figura 3.8: Widget AppBar

3.2.3 Container

Un Container è una Widget di convenienza, utilizzata per comporre Widget più complesse mediante il parametro child. Esso permette di esprimere parametri di tutti i tipi, che siano di posizionamento, di spaziature come margin e padding, di dimensioni come width e height, o di colore. Siccome il Container come dice il nome stesso è una classe di contenimento, cercherà di assumere le dimensioni e il layout più adatti in funzione di quelli delle Widget figlie. Si hanno quindi diverse combinazioni a seconda degli oggetti e dei parametri che contiene:

- nessun child e nessun parametro: il Container assume le dimensioni più piccole possibili
- nessun child e nessun vincolo di allinamento ma width e height definiti: assume le dimensioni più piccole possibili tenendo conto delle dimensioni indicate
- nessun vincolo e nessun child, ma la Widget genitore fornisce vincoli limitati: il Container si espande per adattarsi a tali vincoli
- vincoli di allineamento e il genitore fornisce vincoli limitati: il Container cerca di ridimensionare la Widget figlia
- nessun vincolo e un child che ha una dimensione specificata: si adatta alla dimensione della Widget figlia.

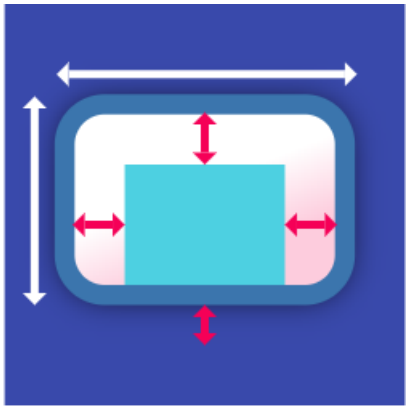


Figura 3.9: Widget Container

3.2.4 Row

Una Row è una Widget che mostra i suoi figli in un array orizzontale. Non è una Widget in grado di scorrere lateralmente, di conseguenza si verificherà un errore nel caso gli si assegnino più Wiget figlie di quelle che può contenere. Nel caso in cui una Widget figlia debba espandersi per tutto lo spazio orizzonatale disponibile è possibile inserirla all'interno della Widget Expanded.

Figura 3.10: Widget Row

3.2.5 Column

Una Column è una Widget che mostra i suoi figli in un array verticale. Esattamente come per le Row, anche le Column non scorrono e di conseguenza non è possibile aggiungere un numero troppo grande di Widget figlie. Anche qui, nel caso in cui una Widget figlia debba espandersi per tutto lo spazio verticale disponibile è possibile inserirla all'interno della Widget Expanded.

Figura 3.11: Widget Column

3.2.6 ListView

Una ListWiew è una Widget che permette lo scorrimento lineare di un elenco di Widget figlie nella direzione indicata. É la Widget di scorrimento più utilizzata poichè permette la costruzioni di elenchi complessi in modo molto semplice. É possibile passare nel costruttore una lista di Widget oppure è possibile indicare un ListView-Builder che costruisce automaticamente l'elenco. Una ListView è implementata in modo da cancellare dall'albero delle Widget tutte le Widget figlie che non sono visibili poichè al di fuori dello schermo e le costruisce in tempo reale in seguito allo scorrimento. Così facendo però si perdono anche gli stati delle Widget non visibili. Questo comportamento può essere un'arma a doppio taglio: sicuramente aumenta l'efficienza dell'elenco, ma se le Widget figlie necessitano di mantenere uno stato interno che cambia non è l'astrazione più indicata.

Figura 3.12: Widget ListView

3.2.7 GridView

Una GridView è una Widget che permette lo scorrimento delle Widget figlie e le rappresenta come matrice bidimensionale. Generalemente è utilizzata con due metodi principali:

- GridView.count che dispone le Widget figlie trasversalmente rispetto all'asse indicata in numero massimo passato a count
- GridView.extent che dà alle Widget figlie dimensione massima rispetto all'asse scelta

Figura 3.13: Widget GridView

3.3 Differenze e analogie con Android

In Flutter molte astrazioni native di Android trovano una loro astrazione corrispettiva oppure possono anche non esistere. Anche molti dei concetti fondanti di una applicazione Android non esistono, o meglio, possono essere tralasciate dal programmatore in una applicazione Flutter, come ad esempio la gestione dei thread. Non vi è tuttavia un'integrazione di Android completa nel framework, per questo motivo può essere necessario integrare codice nativo nel file AndroidManifest.xml di Android per autorizzare l'applicazione ad accedere a determinati sensori o servizi per esempio.

3.3.1 Le View

In Android tutto ciò che compare a schermo è una View: pulsanti, barre degli strumenti, input. Una View una volta disegnata non viene più ridisegnata finché non viene richiamato su di essa il metodo invalidate(). Per dare un determinato layout a una View è necessario scrivere un file .xml che contiene tutte le caratteristiche del layout stesso: posizionamento, colore, dimensioni. Android mette a disposizione i metodi addChild() e removeChild() per poter aggiungere o rimuovere rispettivamente una View figlia e un metodo animate() che permette di animarle in modo personalizzato. Per creare View personalizzate è necessario estendere una View preesistente e fare l'override dei metodi implementando il comportamento desiderato. La corrispettiva astrazione delle View in Flutter sono le Widget, anche se con le dovute differenze, come spiegato nel capitolo 3.1.3.

3.3.2 Gli Intents

In Android gli Intents vengono dichiarati nel file AndroidManifest.xml e vengono utilizzati principalmente per due motivi:

1. navigare tra le Activities

2. comunicare con i componenti

Flutter non ha il concetto di Intents, ma se necessario è possibile integrarli nativamente mediante l'uso di plugin. In Flutter si utilizzano i Navigator e le Route per navigare tra le schermate, anche se rappresentano un concetto differente rispetto alle Activities.

startActivityForResult()

In Android il metodo startActivityForResult avvia un'Activity che restituisce un risultato. Il corrispettivo concetto in Flutter è rappresentato dai metodi push e pop della classe Navigator che permettono di cambiare schermata e ritornare un valore.

3.3.3 UI asincrona

L'event loop di Flutter è equivalente a quello di Android, ovvero è collegato al thread principale. In Android, quando si desidera eseguire codice molto pesante o che richiede tempo, come può essere una chiamata di rete, in genere si sposta il lavoro su un thread in background, in modo da non bloccare il thread principale, ad esempio vengono utilizzati gli AsyncTask o i Service. Quando si estende la classe AsyncTask, è generalmente richiesto l'override di tre metodi: onPreExecute(), doInBackground() e onPostExecute(). Non c'è un equivalente in Flutter, dato che la parola chiave await permette di eseguire funzioni che richedono molto tempo, e l'event loop di Dart si occupa già di eseguire i compiti in modo asincrono.

runOnUIThread()

A differenza di Android, che richiede di mantenere sempre libero il thread principale, in Flutter, basta utilizzare le funzionalità asincrone fornite dal linguaggio Dart, come async e await, per eseguire il lavoro asincrono. Se si vuole forzare la creazione di un thread alternativo in Flutter è necessario fare uso degli Isolates.

3.3.4 Le Activities e i Fragment

In Android, una Activity rappresenta una singola azione focalizzata che l'utente può compiere, mentre un Fragment rappresenta un comportamento o una parte

dell'interfaccia utente. I Fragments sono un modo per rende più modulare il codice, e permettono di comporre interfacce utente sofisticate per schermi più grandi aiutando a rendere scalare l'interfaccia utente dell'applicazione. In Flutter entrambi questi concetti ricadono sotto l'ombrello delle Widget.

3.3.5 Gradle

In Android, è possibile aggiungere dipendenze all'applicazione inserendo lo script necessario all'interno del file Gradle, nella cartella build. In Flutter si utilizza il gestore di pacchetti Pub e le dipendenze vengono inserite nel file pubspec.yaml.

3.4 Differenze e analogie con iOS

Esattamente come per Android, Flutter mette a disposizione per iOS astrazioni simili a quelle native, ma non è sempre detto che esprimano lo stesso concetto. Anche qui per svolgere compiti sofisticati, come l'accesso ai sensori, è necessario integrare nativamente.

3.4.1 UIView

In iOS la maggior parte degli oggetti visuali di una applicazione sono istanze della classe UIView. Tali oggetti possono fungere da contenitori per altri oggetti UIView, andando così a formare il layout dell'applicazione. Le UIView sono entità mutabili che vengono create un'unica volta all'istanziazione e non vengono più ridisegnate finchè non viene invalidata chiamando il metodo setNeedsDisplay(), inoltre se si vuole aggiornare una UIView è possibile farlo direttamente. In iOS, tutti gli elementi visuali possiedono un campo opacity o alpha mentre in Flutter è necessario inserire wrappare la Widget di interesse all'interno di un'altra specifica che possiede il parametro opacity. Per modificare la gerarchia delle UIView è possibile utilizzare i metodi addSubView() e removeFromSuperView() che rispettivamente permettono di aggiungere una UIView figlia e rimuovere una UIView figlia. Il corrispettivo concetto delle UIView in Flutter sono le Widget.

3.4.2 ViewController

In iOS, un ViewController rappresenta una porzione dell'interfaccia utente, comunemente utilizzata per uno schermo o una sezione. Questi sono connessi assieme per costruire un'interfaccia utente complessa e aiutare a rendere scalare la stessa. É possibile eseguire l'override dei metodi per la classe ViewController per acquisire i metodi del ciclo di vita della UIView o registrare le callback del ciclo di vita in una AppDelegate. In Flutter non esistono tali concetti, come spiegato nel capitolo 3.1.7.

3.4.3 Navigare tra le pagine

In iOS, per muoversi tra i controller di visualizzazione, è necessario utilizzare uno UINavigationController che gestisce la pila di controller di visualizzazione da mostrare a schermo. In Flutter si utilizzano i Navigator e le Route che funzionano in modo simile ai UINavigationController.

3.4.4 Thread e programmazione asincrona

L'event loop di Flutter è equivalente al loop principale iOS, ovvero quello connesso al thread principale. In Flutter si utilizzano i costrutti asincroni del linguaggio per ottenere il medesimo risultato. In iOS per mostrare i progressi di una long-running activity che viene eseguita in un thread in background si usa un UIProgressView, mentre in Flutter il medesimo concetto è realizzato da una Widget chiamata ProgressIndicator.

3.4.5 CocoaPods

Il concetto analogo al file pubspec.yaml di Flutter in iOS si chiama Podfile. Qui è possibile aggiungere dipendenze e pacchetti.

3.5 FireBase in Flutter

FireBase è una piattaforma creata da Google che offre alla community di sviluppatori molteplici funzionalità quali hosting, database, analisi di dati, messaggistica cloud

e molte altre ancora, che semplificano notevolemente molti dei compiti necessari allo sviluppo di applicazioni Web o mobile [5]. Nel caso di studio Firebase è stato utilizzato per la realizzazione delle notifiche push dell'applicazione.

3.5.1 Funzionalità principali

Le funzionalità più utilizzate di FireBase dai programmatori Flutter sono indubbiamente lo storing di dati in database e il cloud messaging.

Database FireBase

In FireBase è possibile creare database non relazionali anche di grandi dimensioni che sono facilmente integrabili in Flutter mediante il plugin firebase_database. Questo rende estremamente semplice l'interazione con il database, mettendo a disposizione metodi per creare una connessione allo stesso e per manipolare i dati al suo interno.

Cloud messaging FireBase

Il Cloud messaging di FireBase permette di creare notifiche push personalizzate che vengono notificate ai vari dispositivi su cui è installata l'applicazione. Ovviamente è necessaria una configurazione adeguata nell'app e una integrazione a livello nativo, specialmente per quanto riguarda i permessi. Mediante il plugin firebase_core è possibile configurare il device per la connessione con FireBase e da FireBase stesso è possibile inviare notifiche push a tutti i device configurati oppure inviarle solamente ad utenti target.

3.6 Contronto tra Flutter e Xamarin

Il mondo delle applicazioni mobile sta evolvendo velocemente e Flutter nel mercato sta decollando [15]. I suoi vantaggi sono molteplici, come spiegato in questo capitolo. Per capirne meglio i punti di forza e di debolezza può essere utile confrontarlo con un'altra tecnologia esistente già collaudata ed estremamente utilizzata: Xamarin.

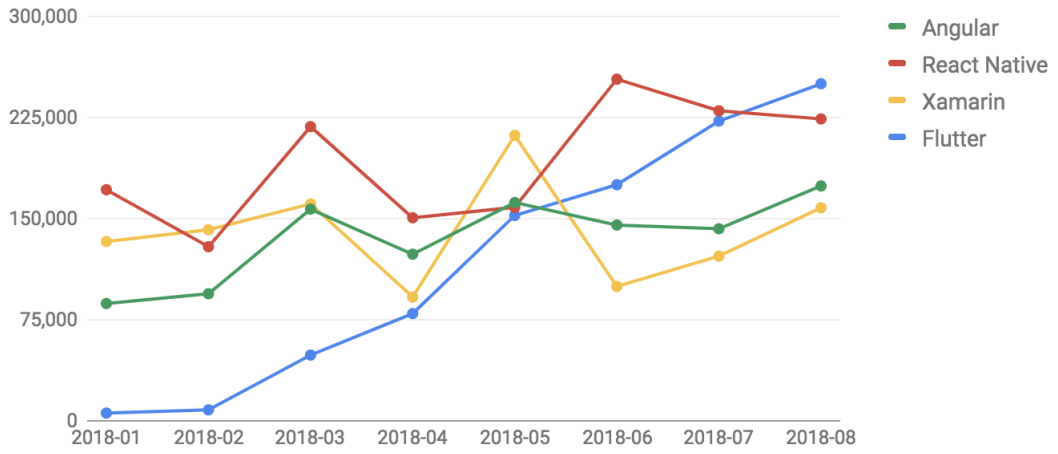


Figura 3.14: Crescita della diffusione di Flutter calcolata come numero di domande inerenti all'argomento presenti su StackOverflow.

3.6.1 Introduzione a Xamarin

Xamarin è un progetto nato nel 2011 che si basa su Mono, un'implementazione cross-pltform di Xamarin Android e Xamarin iOS ed è di fatto stato il primo framework esistente per lo sviluppo di applicazioni platform-independent [44]. Nel 2016 è stato acquistato da Microsoft diventando parte dell'enorme community di .NET. É utilizzato da molte grandi aziende e permette di creare app con design pressochè identico a quelle native.

3.6.2 Linguaggio di programmazione

Xamarin utilizza il C#, che è un linguaggio molto popolare in quanto utilizzato da molti anni da Microsoft ed è stato molto utilizzato in passato anche per lo sviluppo web.

Dart invece è poco utilizzato per lo sviluppo web e non gode di molta rinomanza, tuttavia vanta una documentazione molto corposa sul sito ufficiale.

3.6.3 Architettura

Xamarin utilizza l'ambiente di esecuzione Mono per entrambe le piattaforme Android e iOS. Mono viene eseguito insieme al runtime Objective-C e al kernel Unix

su piattaforma iOS, mentre nel caso di Android, viene eseguito insieme a Android Runtime su Linux o su altri kernel [32].

Flutter invece utilizza Skia che funziona egregiamente come spiegato nel capitolo 3.1.2 ed ha inoltre supporto per Kotlin e per il runtime di Swift.

3.6.4 Curva di apprendimento

Siccome Xamarin utilizza il C# che è un linguaggio molto diffuso e molto noto, gli sviluppatori possono trovare familiare lo sviluppo di applicazioni con Xamarin. Tuttavia esso richiede due configurazioni differenti per Android e iOS che rende più ostico l'apprendimento, specialmente se non si ha familiarità con Visual Studio.

L'apprendimento di Flutter è reltivamente semplice grazie alla notevole documentazione presente sul suo sito, inoltre per gli sviluppatori che hanno esperienza con i liguaggi di programmazione object-oriented è facile migrare verso Dart.

3.6.5 Interfaccia utente

Essenziale nello sviluppo di un'applicazione cross-platform è il supporto nativo, che consente di rendere l'applicazione pressochè identica alla corrispettiva nativa. Xamarin supporta piattaforme multiple, come iOS, Android, Forms, macOS, watchOS, tvOS, e altre, che a loro volta dispongono di molti componenti e moduli per quanto riguarda l'interfaccia utente.

Flutter è fornito in bundle con componenti di rendering dell'interfaccia utente, accesso all'API del dispositivo, navigazione, test, gestione stateful e moltissime librerie grafiche come Material e Cupertino. Questo ricco set di componenti elimina la necessità di utilizzare librerie di terze parti.

3.6.6 Community di supporto

Essendo Xamarin il più anziano tra i framework per lo sviluppo di app platform-independent vanta una vastissima diffusione nel mondo, con tanto di canale ufficiale Twitter e forum della community.

Flutter è in circolazione da tempi relativamente recenti, Flutter 1.0 è stato infatti rilasciato nel 2018, ma ha dalla sua parte il fatto di essere supportato da una community come quella di Google e quindi può avere ottime potenzialità di crescita.

3.6.7 Conclusione

In conclusione è evidente che a Flutter non manchi nulla per poter competere con gli altri framework già esistenti, è solamente penalizzato dal fatto di essere arrivato sul mercato più tardi. Per quanto però riguarda le prestazioni, l'efficienza, il supporto nativo ed anche la semplicità d'utilizzo, Flutter può ad oggi essere considerato un ottimo prodotto ed è proprio per questo che se il trend rimane tale, il mercato delle applicazioni multipiattaforma sarà in futuro sempre più orientato verso Flutter, come dimostra la figura 3.14.

Capitolo 4

Flutter e la programmazione reattiva

Flutter integra al suo interno paradigmi di programmazione reattiva come ad esempio gli Stream e fa uso di plugin che permettono di implementare pattern standard a supporto della programmazione reattiva stessa, il più importante è BLoC. Nel caso di studio la programmazione reattiva è stata implementata per poter realizzare determinate logiche implementative al di fuori di una particolare Widget di interesse in modo da renderle riutilizzabili in diverse situazioni astraendo il contesto specifico.

4.1 Programmazione reattiva

La programmazione reattiva è un paradigma basato sulla registrazione di cambiamenti che avvengono in seguito ad eventi e che si propagano nel tempo. Tale modello permette al programmatore di descrivere un applicazione in termini di cosa deve fare, lasciando al linguaggio il compito di gestire quando farlo. Mediante questo paradigma i cambiamenti che avvengono sono automaticamente propagati ai Listener ovvero gli oggetti che sono in ascolto. Un esempio chiarificatore è il seguente:

$$a = 1;$$
$$b = 2;$$
$$c = a + b;$$

Nella normale programmazione imperativa il valore di c è 3 e rimarrà 3 finchè non gli verrà assegnato un nuovo valore, anche se i valori di a e b cambiano. Nella

programmazione reattiva invece il valore di c viene sempre mantenuto aggiornato in base ai valori di a e b, ossia c dipende da a e b nel senso che un cambiamento che avviene in queste due variabili viene propagato a c [47]. Si introduce quindi il concetto di dipendenza, che altro non significa che ci sono Listener che ascoltano il cambiamento delle variabili da cui dipendono.

4.2 Programmazione reattiva in flutter

La programmazione reattiva consente di gestire flussi asincroni di eventi, come ad esempio l'interazione con l'utente o una chiamata HTTP. In Flutter gli Stream rappresentano la principale astrazione per la programmazione reattiva. Uno Stream, come spiegato nella sezione 2.2.4 è proprio una sequenza asincrona di eventi che generalemente non ha una fine. Anche le liste tuttavia possono essere considerate come stream finiti e deterministici. Uno stream può generare tre differenti tipologie di dati:

1. creazione di un nuovo valore
2. notifica di errore
3. notifica che lo stream è completo e quindi concluso

Flutter incoraggia molto l'utilizzo di programmazione reattiva per lo sviluppo delle applicazioni, anche se non sempre è possibile utilizzarla. Essa permette di non fare uso del metodo setState() con due principali conseguenze:

- migliora molto la riusabilità del codice, in quanto rende possibile la separazione della logica applicativa dalla parte puramente estetica
- migliora l'efficienza dell'applicazione poichè evita di ridisegnare una Widget e l'intero albero delle Widget ad ogni cambiamento

Google stessa incoraggia la community di Flutter ad utilizzare la programmazione reattiva mediante l'aggiornamento costante di plugin sul gestore di pacchetti Pub che ne semplificano l'implementazione. Il pattern più utilizzato prende

il nome di BLoC ed è diventato un vero e proprio pattern di riferimento per la programmazione reattiva in Flutter.

4.3 Pattern BLoC

BLoC (Business Logic Component) è un pattern di programmazione reattiva introdotto da Google la cui idea è quello di avere delle componenti logiche di business, i Bloc appunto, che contengono la logica dell'applicazione e che possono essere facilmente utilizzati in Widget differenti, ma anche in applicazioni Dart diverse, grazie alla loro portabilità.

4.3.1 Funzionamento di BLoC

BLoC basa il suo funzionamento sul concetto di stato di una Widget. Il compito del pattern non è altro che mappare gli eventi in stati. Per farlo utilizza componenti Bloc che contengono al loro interno degli Stream [3]. Un Bloc può essere considerato come una pila, la cui entrata si chiama Sink e l'uscita si chiama Stream. In entrata accetta eventi generati dall'applicazione, come ad esempio un tap dell'utente e in uscita ritorna uno stato che può essere utilizzato per propagare il cambiamento nella Widget che contiene il Bloc.

4.3.2 Bloc Provider

Per poter inserire il componente Bloc all'interno di una Widget è necessario fare uso di un Bloc Provider ossia di un oggetto che estende InehritedWidget. Ogni figlio di InehritedWidget ha accesso ai componenti forniti dalla classe InheritedWidget stessa. In questo modo è possibile inserire il Bloc Provider all'interno di una Widget, poichè può essere passato come parametro dove è accettata una InheritedWidget.

4.3.3 StreamBuilder

Per poter ascoltare le modifiche che avvengono allo Stream e ricostruire l'albero delle Widget basandoci sullo stesso, esiste una classe chiamata StreamBuilder

che permette di accettare un dato iniziale, lo stream da ascoltare e la Widget da disegnare.

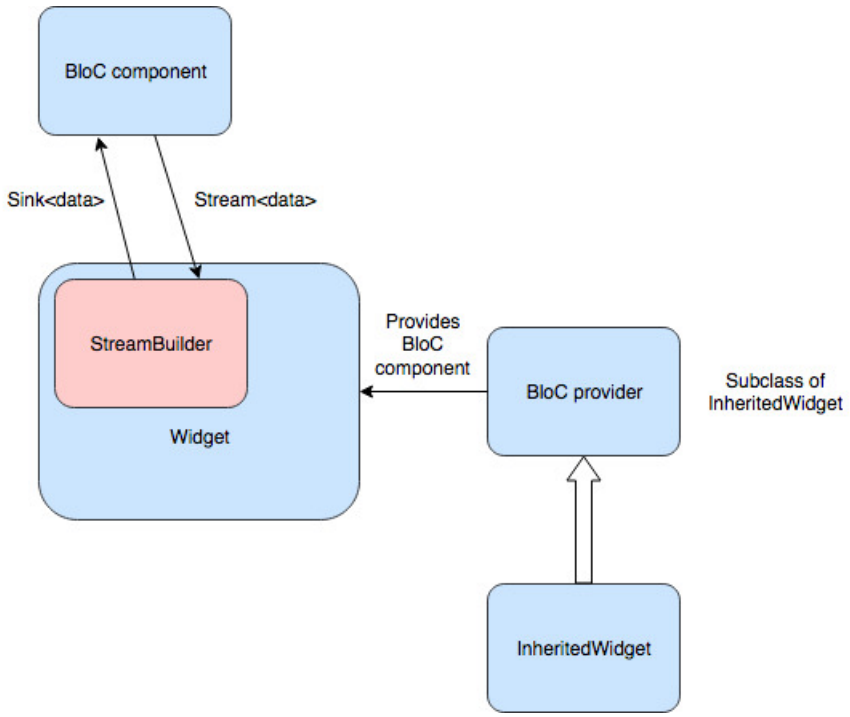


Figura 4.1: Implementazione del pattern BLoC

4.3.4 Integrazione in Flutter

L'integrazione più utilizzata del pattern BLoC in Flutter, così come è stata realizzata nel caso di studio, prevede l'inclusione di due plugin di Pub: bloc e flutter_bloc. Il primo implementa un Bloc generico che può essere esteso. É necessario fare l'override dei metodi initialState, che ritorna lo stato iniziale, e mapEventToState(event e) che prende in ingresso un evento e restituisce uno stato. Inoltre mette a disposizione vari metodi per la gestione del Bloc, il più importante è dispatch(event e). Quando viene richiamato il metodo dispatch e gli viene passato in input un evento, automaticamente viene richiamato mapEventToState e si ottiene quindi lo stato da passare alla Widget. Invece flutter_bloc implementa strutture dati che semplificano l'utilizzo del Bloc, come BlocProvider e BlocBuilder, che permettono di includere facilmente un Bloc nella Widget di interesse. Tale integrazione, come specificato,

non è obbligatoria, o l'unica possibile, ma è attualmente la più utilizzata poichè i plugin in questione sono i più collaudati, come mostra il loro ranking sul sito ufficiale di Pub.

4.4 Pro e contro della programmazione reattiva

Punti di forza:

- permette la separazione della parte estetica dell'applicazione dalla sua logica. Questo consente di ridurre notevolemente le righe di codice scritte in quanto la logica può essere esterna alla Widget che la utilizza e quindi può essere riutilizzato lo stesso codice in più Widget diverse senza doverlo riscrivere, come è stato realizzato nel caso di studio
- garantisce maggiore efficienza in quanto rende inutile l'utilizzo del metodo setState() per ridisegnare l'albero delle Widget in seguito ad un evento, perchè il cambiamento è propagato mediante Stream.

Punti di debolezza:

- non sempre è possibile utilizzare la programmazione reattiva, spesso complica molto la leggibilità di una applicazione specialmente se la Widget da implementare è complessa a tal punto da dover contenere molti Bloc.
- comporta una proliferazione di classi poichè per ogni tipologia di compito differente è necessario implementare una nuova classe Bloc ed eventualmente anche nuovi tipi event e state e quindi nuovi oggetti.

Capitolo 5

Caso di studio: Active CRM

In questo capitolo verrà presentata l'applicazione Active CRM realizzata nel contesto di questa tesi mediante framework platform-independent Flutter.

5.1 Active CRM

Active CRM è un progetto nato dalla necessità dell'azienda di realizzare la versione mobile del modulo CRM di un loro prodotto. In economia aziendale CRM significa Customer Relationship Management ed è legato al concetto di fidelizzazione dei clienti [41]. Questa applicazione organizza la parte commerciale di un'azienda, gestisce il rapporto con i clienti mediante la pianificazione di attività e segue l'utente nelle opportunità di vendita.

5.2 Requisiti

Active CRM è in grado di leggere i dati da un database mediante chiamate a Web API e mostrarli in modo basilare all'utente e permette di gestire un nuovo inserimento o una modifica di campi esistenti. L'azienda ha deciso di intraprendere la realizzazione di questo progetto facendo uso di Flutter per vari motivi, innanzitutto perchè era una tecnologia nuova, da poco lanciata sul mercato che secondo i trend e gli insider si sarebbe espansa molto in futuro e di fatti così sta avvenendo. In secondo luogo perchè era di interesse dell'azienda trovare validi strumenti che potessero

rimpiazzare la scrittura di applicazioni native. Era requisito fondamentale infatti diminuire il time-to-market dell'applicazione e soprattuto il numero di programmatori che lavorassero alla realizzazione della stessa.

L'app si compone di quattro schermate principali:

- Clienti
- Contatti
- Azioni
- Target

5.2.1 Clienti

I clienti mostrati in questa schermata sono tutti i clienti di un determinato responsabile commerciale per una data filiale selezionata nelle impostazioni. I clienti possiedono vari stati e si identificano per diverse tipologie a seconda della loro produzione e per differenti inquadramenti che rappresentano il settore commerciale. Un cliente può anche decidere di registrarsi come persona fisica e deve in questo caso inserire le sue generalità. É presente l'icona dei filtri per poter filtrare i clienti in base al loro stato.

5.2.2 Contatti

La schermata dei contatti visualizza tutti i contatti di tutti i clienti di un responsabile commerciale per una data filiale selezionata nelle impostazioni. Nella appbar è presente l'icona dei filtri per filtrare i contatti in base alla loro importanza: se sono contatti principali o meno.

5.2.3 Azioni

Le azioni sono attività che possono essere organizzate dall'azienda assieme ad un cliente. Nell'applicazione vengono mostrate tutte le azioni del responsabile commerciale che è acceduto mediante login. Le azioni servono per pianificare le attività con

i clienti, calcolando in maniera precisa un tempo di inizio e fine a seconda del tipo di azione. Qui possono essere applicati filtri relativi allo stato dell'azione.

5.2.4 Target

I target corrispondono a obiettivi di vendita di un prodotto a un cliente. Si stima una probabilità di riuscita e una priorità per ogni target. Nel percorso dalla selezione dell'obiettivo di vendita, alla vendita effettiva, un target può assumere diversi stati:

- obiettivo: si è dichiarata l'intenzione di vendere un prodotto ad un cliente, il quale però non è ancora stato contattato
- non interessato: il cliente non è al momento interessato per vari motivi al prodotto che gli viene offerto.
- in corso: il cliente è stato contattato ed informato, ma non si hanno ancora elementi per creare un'opportunità
- opportunità: è stata creata un'azione a fronte di un target
- persa: la commessa non è stata acquisita
- acquisita: la commessa è stata acquisita

I filtri realtivi ai target permettono di visualizzare target in base al loro stato.

5.2.5 Casi d'uso

Di seguito mostro tre casi d'uso principali:

1. modifica o inserimento di una entità
2. completamento di una azione
3. modifica della filiale dalle impostazioni

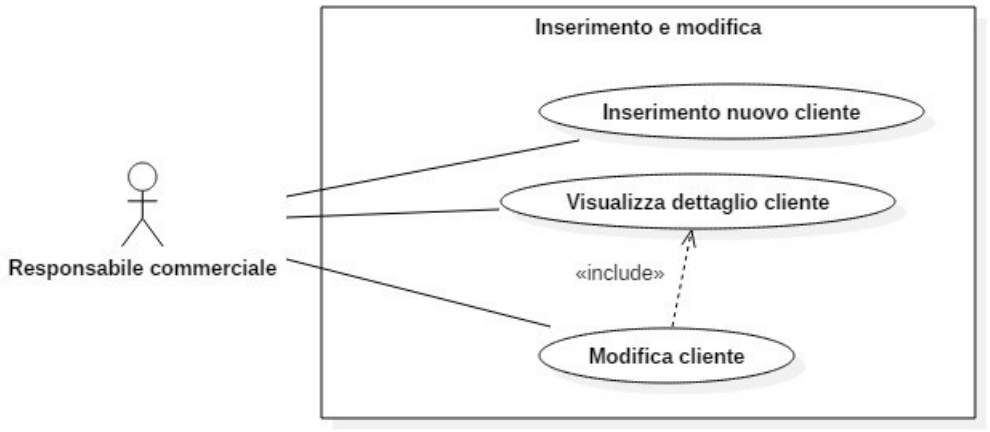


Figura 5.1: Caso d'uso per l'inserimento e la modifica di un cliente

Nella figura 5.1 viene mostrato il caso d'uso relativo all'inserimento o la modifica delle varie entità, nella figura in particolare si prende in esame il cliente, ma il processo è analogo per contatti, azioni e target. Il responsabile commerciale che fa uso dell'applicazione può inserire un nuovo cliente oppure può modificarne uno già esistente solo se è acceduto precedentemente al suo dettaglio.

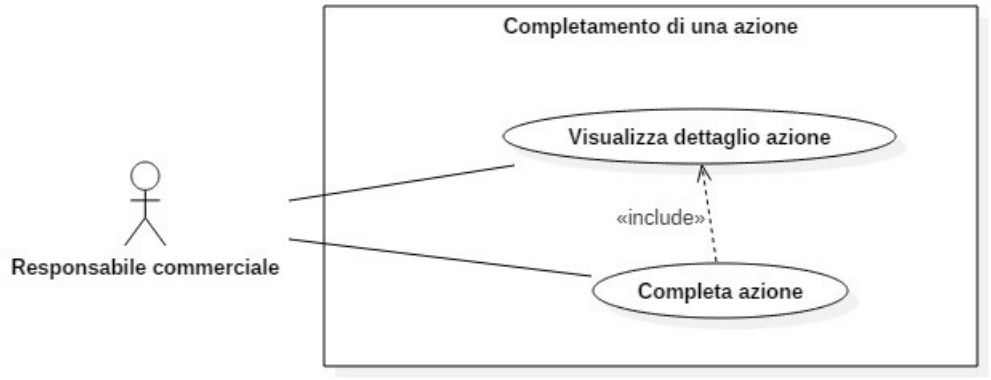


Figura 5.2: Caso d'uso per il completamento di una azione

La figura 5.2 mostra il caso d'uso relativo al completamento di una azione. É possibile procedere a tale scopo solamente se si è prima visualizzato il dettaglio della azione. Il completamento di una azione non è altro che un modo automatico e semplificato per rendere lo stato della stessa "completata" senza doverlo modificare

dalla form di modifica. Un'azione può essere completata scegliendo una data di completamento, che può essere quella programmata col cliente o un'altra scelta a piacere e una volta che risulta completata non è più modificabile.

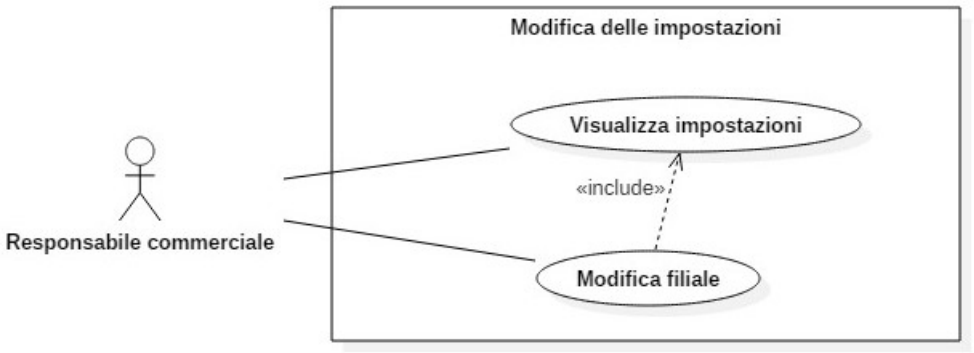


Figura 5.3: Caso d'uso per la modifica delle impostazioni

Nella figura 5.3 viene mostrato il caso d'uso relativo alla modifica delle filiali nelle impostazioni. Per farlo ovviamente è necessario essere acceduti alle impostazioni. La modifica della filiale corrente avviene mediante la selezione della stessa da una checklist in un popup tra tutte le filiali che possiede l'azienda del responsabile commerciale che ha effettuato il login. Una volta che viene modificata la filiale anche i clienti e i contatti che erano stati finora visibili cambieranno poichè verranno aggiornati con i clienti e i contatti relativi alla filiale scelta. Azioni e target invece rimangono gli stessi perchè dipendono dal responsabile commerciale e non dalla filiale corrente.

5.3 Progettazione dell'applicazione

La progettazione dell'applicazione è stata lunga e prudente, specialemente agli inizi, quando la documentazione relativa a determinati meccanismi era ancora scarsa per Flutter. Il pattern MVC ad esempio si è deciso di non implementarlo nell'applicazione poichè le pubblicazioni riguardo alla sua integrazione in Flutter non erano ancora sufficienti. Si è deciso invece di provare a implementare il pattern BLoC per tutti i campi delle form di inserimento e modifica. Con la programmazione reattiva infatti

è stato possibile implementare la logica di tali campi al di fuori della Widget di interesse, creando una classe generale e riutilizzabile che aggiorna automaticamente i suoi parametri mediante la propagazione del cambiamento. Di seguito sono riportati concetti principali dell'applicazione e come sono stati progettati nel contesto di Active CRM.

5.3.1 Widget dell'applicazione

Ogni schermata visualizzata nel progetto corrisponde a una Stateful Widget, questo perchè ogni schermata deve reagire ad eventi, che siano di input o in seguito a risposte a chiamate di rete. Le quattro schermate principali implementano un FutureBuilder, ovvero una Widget che permette di fare una chiamata di rete e attende la risposta. Quando tale risposta arriva, visualizza il risultato a schermo. Per navigare tra le schermate si è fatto uno dei Navigator e di un plugin che permette di animare in modo semplice lo scorrimento delle schermate.

5.3.2 Implementazione delle web API

La chiamata di Web API è fondamentale per l'applicazione in quanto è grazie a questo meccanismo che vengono prelevati dal backend, o inseriti nello stesso. Si fa uso di chiamate HTTP prevalentemente di tipo get, questo perchè l'header delle chiamate GET può contenere tipi di dato anche di tipo Map<String, int>, come ad esempio gli id delle varie entità, mentre il body di una chiamata POST può contenere solo dati di tipo Map<String, String>. Si è deciso di creare un package con un file che contenga le implementazioni delle Web API in modo da richiamare il solo metodo necessario per effettuare la chiamata di interesse. Tali chiamate ritornano come output dei file JSON contenenti una risposta, ad esempio una chiamata mediante Web API al database per visualizzare la lista dei clienti, ritorna un oggetto JSON contenente un array con tutti i clienti. Ho quindi dovuto decodificare le risposte e il contenuto istanziarlo come classi dell'applicazione.

5.3.3 Classi dell'applicazione

Mediante chiamate alle web API, vengono popolate liste globali che rappresentano ad esempio la lista dei clienti, delle azioni, dei contatti e dei target, ma anche di altre entità dell'applicazione necessarie per l'inserimento e la modifica come le filiali, i responsabili commerciali, le priorità del cliente. Tutte queste entità erano inizialmente pensate come liste generiche, senza avere una vera e propria classe che le rappresentasse come oggetti. Successivamente si è deciso di trasformale in liste specifiche ed è stato quindi necessario introdurre nuove classi, una per ognuna di queste entità. La decisione è stata scaturita dal fatto che oltre ad avere codice più leggibile e meglio organizzato, creando oggetti persistenti nell'applicazione è possibile in futuro salvare determinate configurazioni magari in locale nello smartphone, per possibili estensioni dell'applicazione.

5.3.4 Autorizzazioni

Le autorizzazioni sono valori applicati ai campi di azioni e target che determinano i permessi che i campi stessi possiedono. Una volta entrati nel dettaglio di queste entità vengono richiamate due ulteriori web API per le autorizzazioni e si riceve come risposta un JSON in cui sono scritti i valori dei permessi relativi ai campi in modifica a cui sono applicati. Le autorizzazioni possono assumere tre differenti valori:

- NO: significa che il campo in questione deve essere nascosto nella form di modifica
- VIS: significa che il campo in questione deve essere visualizzato nella form di modifica, ma non deve essere modificabile
- MOD: è possibile modificare il campo nella form di modifica

Le autorizzazioni sono state gestite con una enumerazione che contiene questi tre valori che sono predisposti da backend a seconda delle esigenze del responsabile commerciale.

5.3.5 Notifiche push

All'interno dell'applicazione sono state inserite le notifiche push, servizio realizzato mediante la funzionalità di cloud messaging di Firebase. É stato necessario creare un progetto dalla console di Firebase e integrarlo nativamente in Android. Alla creazione del progetto, Firebase gli assegna una API server univoca che servirà per l'invio del messaggio. All'avvio dell'applicazione viene fornito un token Firebase all'applicazione stessa che è univoco anch'esso e rimane tale anche a seguito di aggiornamenti. Lato backend l'API send() di SqlServer permette di specificare un token, una API server e il contenuto del messaggio per inviare notifiche push ad utenti target. Ad esempio se viene completata un'azione da parte di un cliente, il responsabile riceverà la notifica push che tale azione è stata completata. Alcune criticità sono state gestite, ad esempio il fatto che le notifica push non apparivano nella barra di Android se l'applicazione era in foreground. Questo perchè tali notifiche servono appositamente per riportare l'utente nell'applicazione se questa è chiusa o in background. Per ovviare al problema è stato fatto uso della funzione nativa onMessage() che viene richiamata alla ricezione di un messaggio di notifica e se l'applicazione è in foreground fa comparire una dialog di notifica.

5.4 Architettura dell'applicazione

Di seguito verrà spiegata la struttura dell'applicazione per quanto riguarda l'interazione fra le Widget del caso di studio. Mediante l'uso dei Navigator è possibile passare da una Widget all'altra cambiando schermata e costruendo una Route ovvero il cammino che è stato eseguito e che può essere ripercorso a ritroso per tornare alle Widget precedenti. In questo modo è possibile interconnettere le schermate di una applicazione.

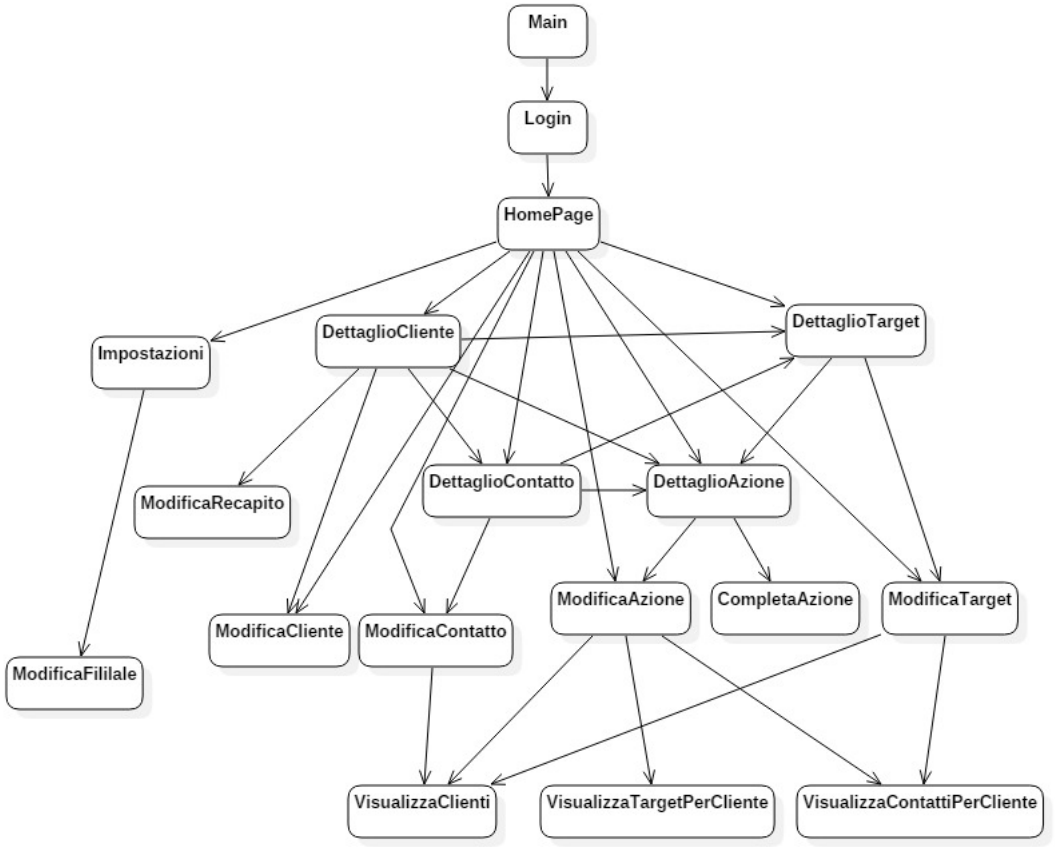


Figura 5.4: Architettura dell'applicazione

All'apertura l'applicazione mostra una schermata di login in cui è possibile inserire le proprie credenziali di responsabile commerciale. Le azioni e i target sono gli stessi per un dato responsabile commerciale, mentre i clienti e i contatti cambiano a seconda della filiale selezionata nelle impostazioni.

5.4.1 HomePage

La HomePage è la Widget principale dell'applicazione. É realizzata mediante una PageView ovvero una Widget che permette di scorrere lateralemente tra le sue Widget figlie. Qui sono inserite le liste di Cliente, Contatto, Azione e Target. Alla creazione della Widget vengono richiamate le web API necessarie per il corretto funzionamento dell'applicazione, le quali si occupano di popolare liste di entità fondamentali. Ad esempio i responsabili commerciali, le filiali, la priorità del cliente, la priorità del target, il tipo di cliente, lo stato del cliente, lo stato del target e molte altre ancora.

Da qui è possibile accedere a tutti i dettagli e alle form di inserimento mediante il tap su appositi bottoni di inserimento. Al tap su un entità della lista si apre il dettaglio, ovvero viene eseguita una chiamata alla relativa web API e viene popolato un oggetto che rappresenta il dettaglio dell'entità. Sono quindi visualizzabili parametri aggiuntivi nascosti nella homepage. Da qui è possibile modificare l'entità, infatti mediante il tap sugli appositi pulsanti si apre la schermata di modifica, in cui la form è già popolata con i parametri presenti e sarà possibile salvare solo dopo aver effettivamente modificato almeno un campo della form. É inoltre possibile accedere alla Widget Impostazioni mediante il drawer laterale. Essa permette di modificare la filiale in uso mediante la classe ModificaFiliale. Essa si occupa di aprire un popup in cui è possibile selezionare una filiale tra tutte quelle presenti nella lista popolata dalla web API. Nel momento in cui si cambia filiale vengono richiamate anche le API per la lista dei clienti e dei contatti, che saranno modificate, mentre la lista delle azioni e dei target non dipendono dalla filiale ma solo dal responsabile commerciale.

5.4.2 Form di modifica

Le form di modifica dell'applicazione sono ModificaCliente, ModificaContatto, ModificaTarget, ModificaAzione, CompletaAzione e ModificaRecapito. Nel momento in cui si sta accedendo a tali form in modifica significa che precedentemente si è richiamato il dettaglio della rispettiva entità e di conseguenza sarà popolato un oggetto di dettaglio per l'entità stessa che contiene informazioni aggiuntive. Le form

in modifica sono quindi già popolate con i parametri ricevuti dalla web API del dettaglio ed è possibile salvare, ovvero richiamare la web API di inserimento, solo dopo l'effettiva modifica. Modificando un campo infatti compare in alto a destra l'icona per salvare che altrimenti è nascosta. Questo comportamento è stato realizzato utilizzando dei controller per i campi ai quali listener è stata attaccata una funzione che controlla il cambiamento dei campi stessi. Tali form sono le medesime nel caso di un nuovo inserimento, cambia semplicemente il fatto che tutti i campi sono vuoti e non popolati e l'icona per il salvataggio è sempre presente.

5.4.3 Dettagli

Le Widget di dettaglio nell'applicazione sono DettaglioCliente, DettaglioContatto, DettaglioAzione e DettaglioTarget. In queste Widget vengono mostrate le informazioni dettagliate di un'entità che sono state ricevute mediante una chiamata alla web API corrispondente. Un'entità può avere associata una lista di altre entità, come ad esempio un cliente ha associato una lista di contatti, una lista di azioni, una lista di target e una lista di recapiti, che sono qui visualizzati, ed è possibile modificarli o inserirne di nuovi. Da qui è possibile accedere alle form di modifica.

5.4.4 Classi di visualizzazione

VisualizzaClienti, VisualizzaContatti e VisualizzaTarget sono Widget a cui si accede dalle form di modifica che mostrano le liste delle rispettive entità secondo una logica precisa. VisualizzaClienti ad esempio mostra tutti i clienti per la filiale selezionata nelle impostazioni. VisualizzaContatti invece visualizza tutti i contatti per il cliente precedentemente selezionato nella form.

5.5 Implementazione prototipale

Di seguito sono esposti i prinicipali diagrammi UML che descrivono come è strutturata l'applicazione e vengono mostrate le principali schermate che la compongono.

5.5.1 Classi principali

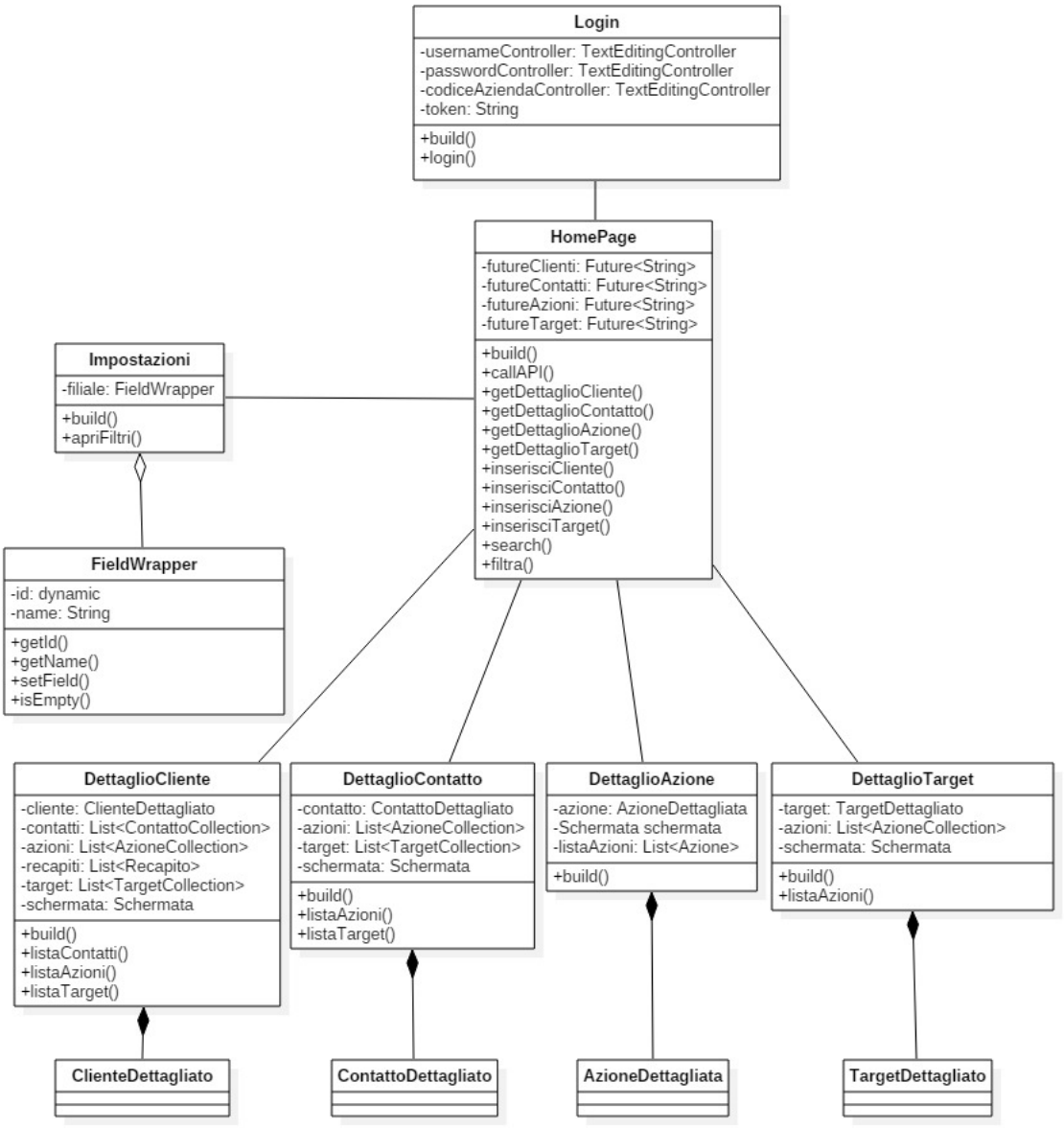


Figura 5.5: Struttura principale di HomePage

La figura 5.11 mostra il collegamento tra le principali classi del progetto. Dalla homepage è possibile raggiungere i dettagli delle entità e ciascuna contiene al suo interno un oggetto dettagliato dell'entità, che recupera mediante una chiamata alla web API corrispondente e che vive solo all'interno della classe stessa. Per realizzare il comportamento reattivo è stato utilizzata una struttura chiamata FieldWrapper che non fa altro che incapsulare al suo interno un id e un valore da visualizzare a schermo. In tal modo il pattern BLoC mantiene sempre aggiornato l'id anche modificando tale oggetto all'esterno della Widget di interesse.

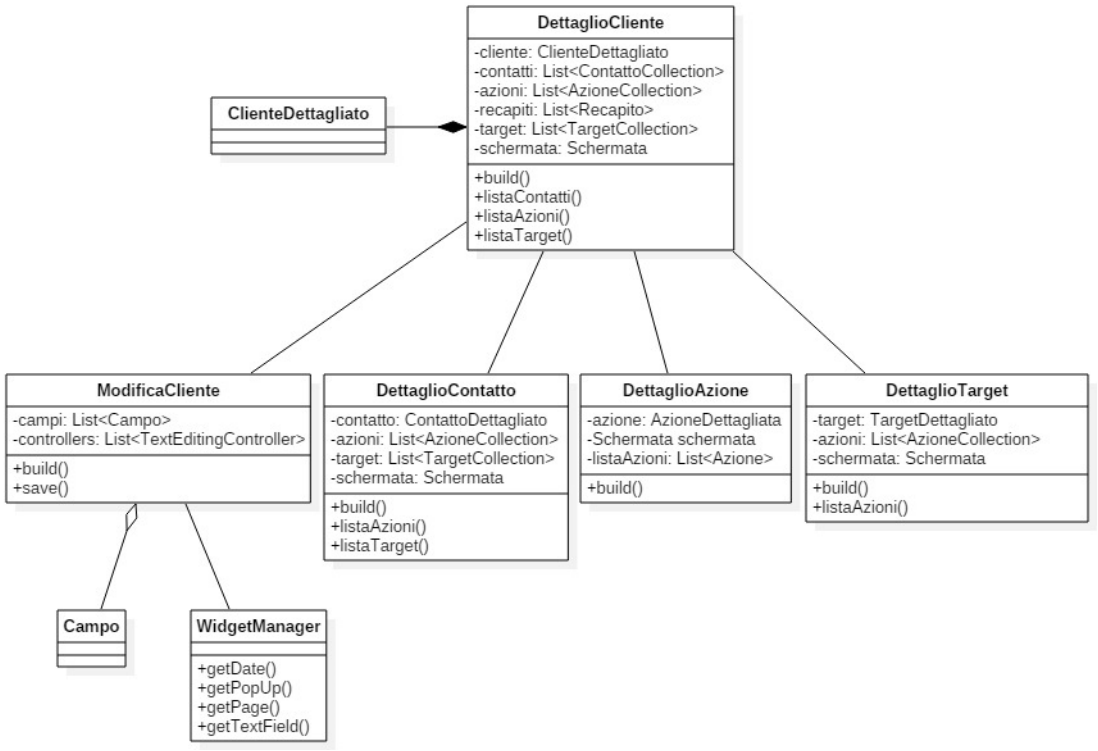


Figura 5.6: Struttura di DettaglioCliente

Nella figura 5.6 viene mostrato il focus su DettaglioCliente, in particolare tale classe contiene al suo interno un oggetto di tipo ClienteDettagliato che esiste di fatto solo all'interno di quella particolare Widget e che rappresenta un cliente con tutte le sue specifiche. Da qui è possibile raggiungere i dettagli delle altre entità o aprire la form di inserimento e modifica. La classe WidgetManager incorpora la logica applicativa ed è stata realizzata mediante programmazione concorrente.

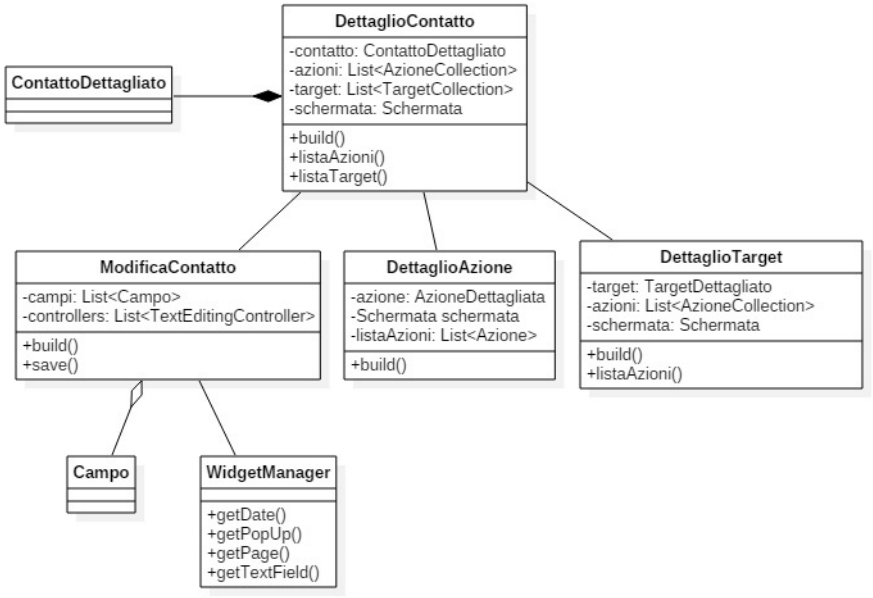


Figura 5.7: Struttura di DettaglioContatto

Nella figura 5.7 il focus si sposta sulla classe DettaglioContatto e mostra come sono gestiti i contatti dettagliati. Anche qui si avrà il rispettivo oggetto dettagliato ed è da qui possibile spostarsi su DettaglioAzione e DettaglioTarget, rispettivamente per le azioni e i target che riguardano quel particolare contatto. Altrimenti è possibile accedere alla form di inserimento e modifica.

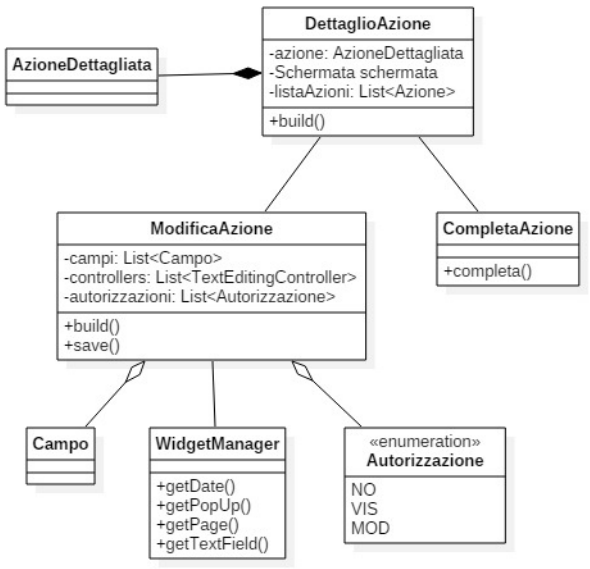


Figura 5.8: Struttura di DettaglioAzione

La figura 5.8 mostra la porzione di programma relative alle azioni. Qui non è possibile accedere ai dettagli delle altre entità perchè sostanzialmente a livello logico una azione non "ha un target" o non "ha un contatto", come invece può essere viceversa, cioè a un target può essere associata una azione o un contatto. Tuttavia l'azione può essere completata, ovvero è possibile modificarne lo stato rendendola di fatto completata e in quel caso non più modificabile. É possibile accedere alla form di inserimento e modifica che qui è legata con una aggregazione alle autorizzazioni, questo perchè tali autorizzazioni esistono anche al di fuori delle azioni e influiscono sul comportamento dei campi della form.

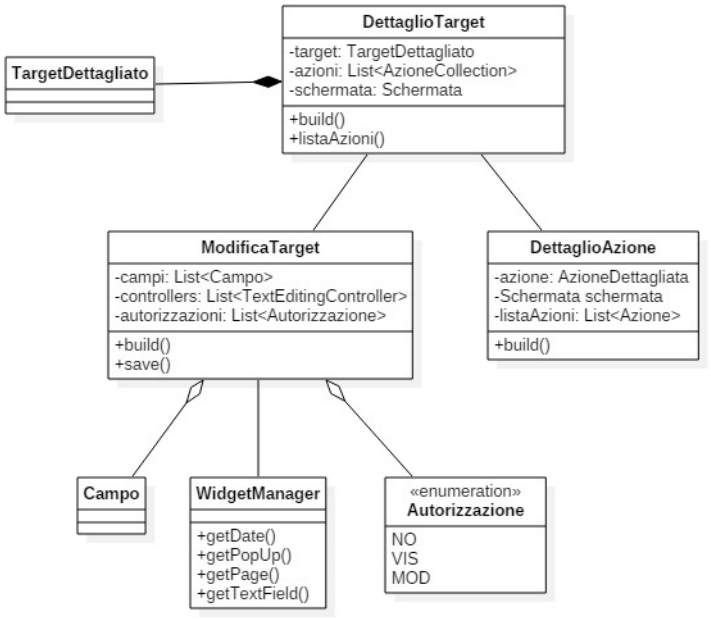


Figura 5.9: Struttura di DettaglioTarget

La figura 5.9 pone l'attenzione sui target e come essi sono stati gestiti nel caso di studio. Da qui è possibile muoversi ai dettagli delle azioni associate a un determinato target. É possibile accedere alla form di inserimento e modifica che anche qui è legata con una aggregazione alle autorizzazioni, questo perchè tali autorizzazioni esistono anche al di fuori dei target e influiscono sul comportamento dei campi della form.

5.5.2 View dell'applicazione

La view dell'applicazione è stata realizzata mediante Widget native del framework che implementano il design Material. L'estetica è volutamente minimale per poter essere personalizzata in seguito a richieste da parte dei clienti.

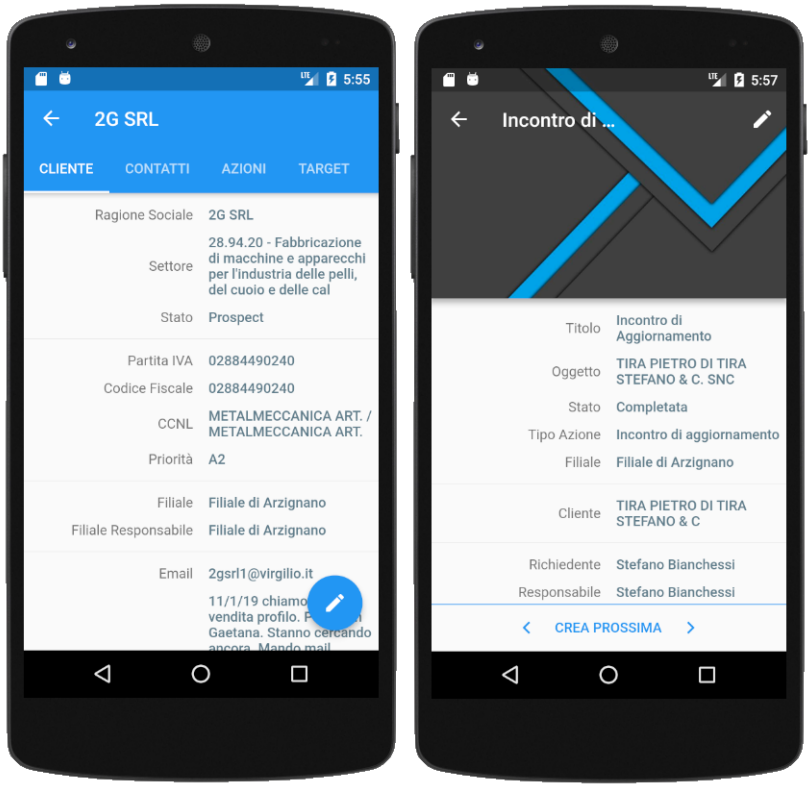


Figura 5.10: Dettagli di due tipologie differenti

Le due schermate della figura 5.10 rappresentano i dettagli di due differenti tipologie. La prima immagine mostra il dettaglio di un cliente, che è analogo a quello di un contatto o di un target, mentre la seconda immagine mostra il dettaglio di una azione. Nei dettagli sono mostrate le informazioni approfondite per l'elemento che si sta visualizzando.

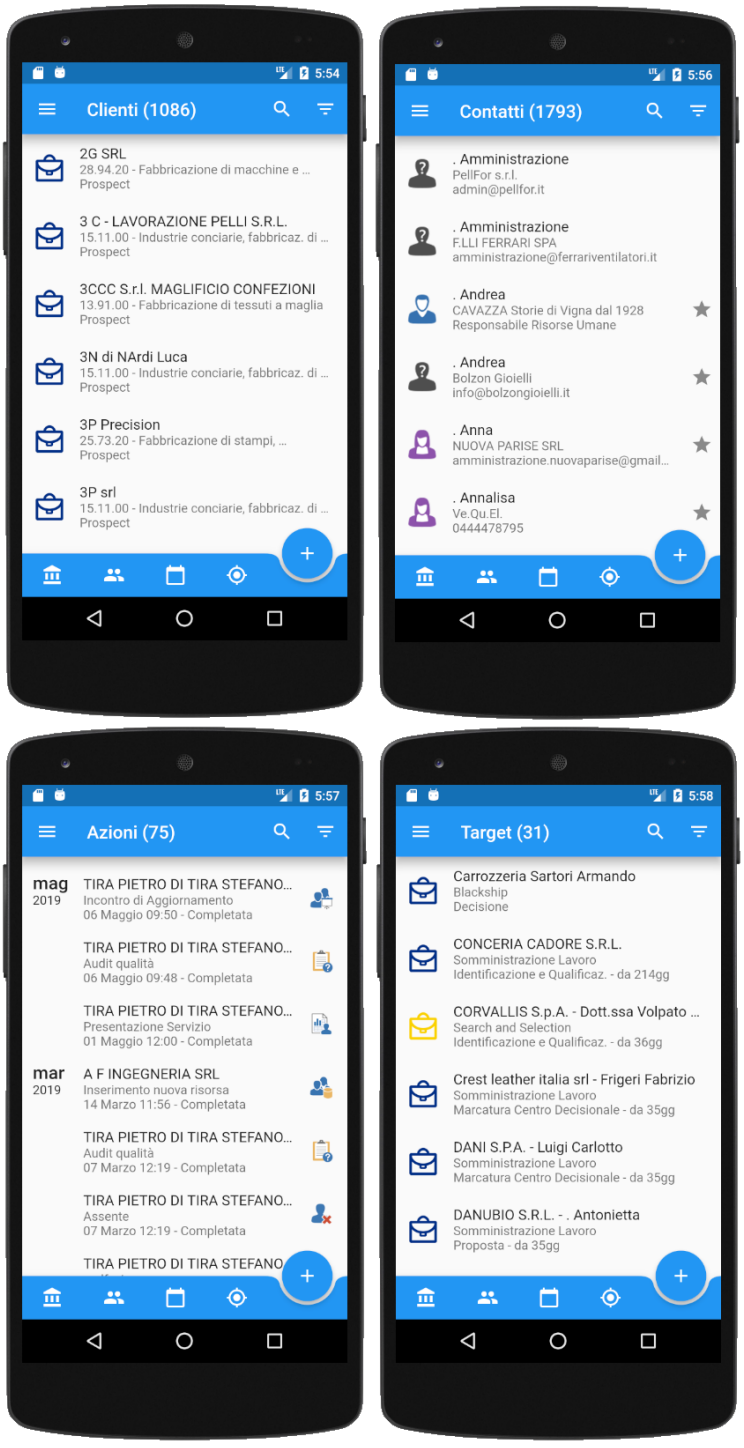


Figura 5.11: PageView della homepage

L'immagine 5.11 mostra la homepage che si compone delle quattro schermate principali dell'applicazione: clienti, contatti, azioni e target, contenenti le rispettive liste di elementi.

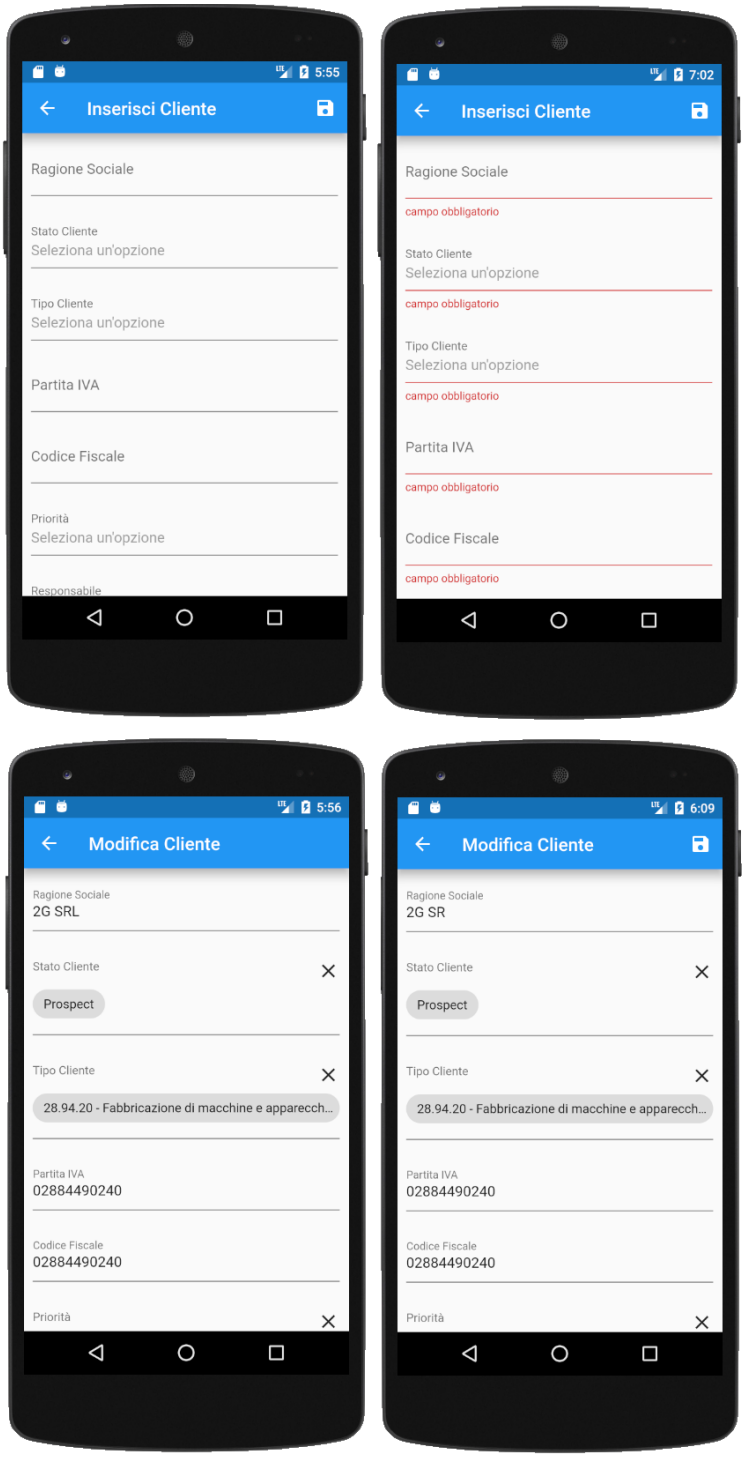


Figura 5.12: Form di inserimento e modifica del cliente

La figura 5.12 mostra le form di inserimento e modifica del cliente. Per le form di modifica i campi sono già compilati con i dati esistenti e l'icona che permette di salvare i cambiamenti compare solo a seguito di una effettiva modifica.

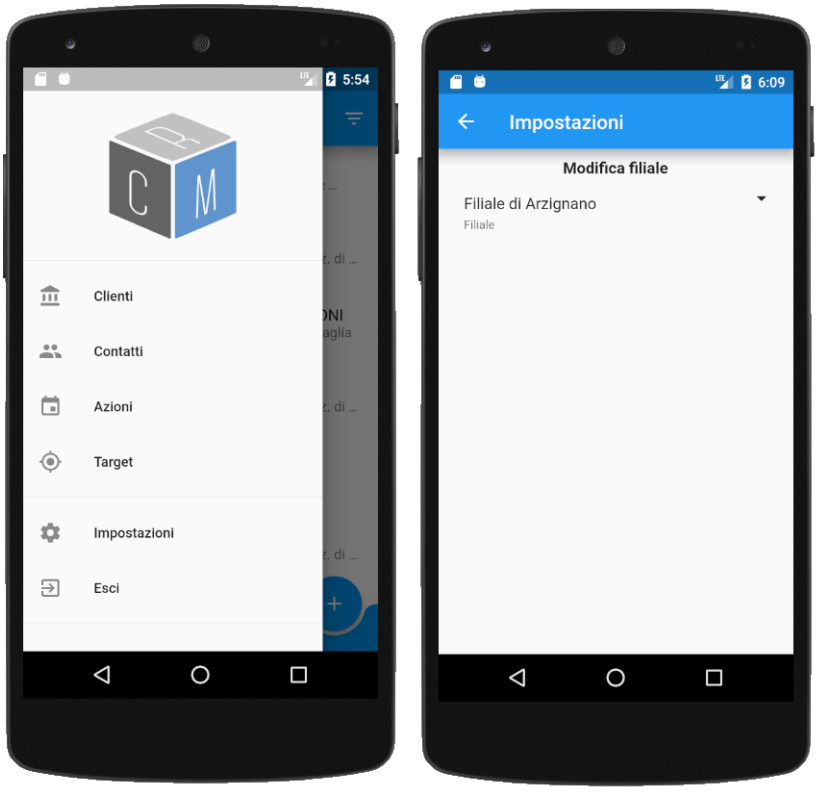


Figura 5.13: Drawer laterale

L'immagine 5.13 mostra il drawer laterale che compare al tap sul menù ad hamburger nella homepage ed è da qui possibile accedere alle impostazioni per modificare la filiale.

5.6 Valutazione dell'applicazione

L'applicazione sviluppata rispetta in toto i requisiti descritti per la parte Android. Per verificare il corretto funzionamento sono stati eseguiti test di funzionamento mediante l'inserimento e la modifica delle entità. Una volta che l'inserimento è andato a buon fine e l'entità inserita rispetta i requisiti per essere visualizzata essa viene immediatamente inserita nella lista visualizzata nella homepage, ordinata secondo un ordinamento preciso, senza che ci sia il bisogno di richiamare le web API di lettura.

5.6.1 Criteri di valutazione

I criteri di valutazione dell'applicazione corrispondono ai criteri base di valutazione di ogni applicazione, come la coerenza e il funzionamento rispetto ai requisiti, la riusabilità, la portabilità ovviamente garantita da Flutter e in particolar modo si è deciso di dare importanza particolare all'estendibilità. La struttura dell'applicazione è stata infatti pensata e progettata per lasciare spazio ad estensioni future, in quanto lo stato attuale di Active CRM rappresenta una versione base dell'applicazione, che potrà essere estesa in futuro aggiungendo e integrando nuove funzionalità su richiesta dei clienti. Altri criteri importanti per la valutazione dell'app sono la fluidità, l'efficienza, la dimensione dell'applicazione. Flutter ha dimostrato di essere un framework sufficientemente maturo per realizzare un'applicazione di medie dimensioni che mette a disposizione le principali funzionalità utilizzate dalla maggioranza delle app: interfacciamento col backend, gestione degli eventi, personalizzazione dell'estetica dell'applicazione, programmazione reattiva. L'applicazione sviluppata è ottimale in termini di performance, non si registrano infatti delay percepibili e anche la navigazione tra le schermate appare fluida. I tempi di realizzazione sono stati tutto sommato esigui se si pensa che è un'applicazione commerciale realizzata da un'azienda per i propri clienti, infatti ha richiesto circa tre mesi di sviluppo. Rispetto alla realizzazione nativa in Java o Kotlin e Swift sicuramente Flutter ha semplificato notevolmente l'implementazione di codice, grazie proprio alle sue strutture di alto livello che nascondo la gestione di concetti più ostici propri dei linguaggi nativi. Anche l'integrazione con plugin sempre nuovi e aggiornati ha permesso di realizzare in

modo semplice ed efficace operazioni per nulla scontate come l'interfacciamento con il backend mediante chiamate a web API e la decodifica di oggetti JSON. Il funzionamento generale è fluido e veloce, le dimensioni stesse dell'app sono relativamente esigue, a riprova che Flutter è un framework estremamente ottimizzato.

5.6.2 Cosa migliorare

L'estetica dell'applicazione è volutamente minimale poichè rappresenta una versione base, che poi l'azienda personalizzerà a seconda delle richieste dei clienti. L'applicazione è stata realizzata su emulatore Android e con integrazioni native funzionanti sul medesimo sistema, di conseguenza è necessario una integrazione corrispettiva per quanto riguarda iOS che non è ancora stata sperimentata. Inoltre il pattern BLoC è stato utilizzato nell'applicazione solamente per la gestione dei campi delle form, è possibile estenderlo in tutte le Stateful Widget eliminando la necessità di richiamare il metodo setState(). Inizialmente il pattern BLoC era ancora ostico da utilizzare a causa della mancanza di plugin che ne semplificassero l'utilizzo, di conseguenza non è stato integrato completamente, ma ne è stata piuttosto fatta una sperimentazione con esito positivo. Anche il pattern MVC è stato lasciato fuori dal progetto, poichè inizialmente era difficile reperire informazioni su come integrare al meglio tale pattern in una applicazione Flutter.

Capitolo 6

Conclusione

Questa tesi si è concentrata sul framework Flutter per lo sviluppo di applicazioni platform-independent, mediante l'utilizzo del linguaggio di programmazione Dart. Per dimostrare la validità e l'efficienza di questa tecnologia ho portato il caso di studio Active CRM che ho esposto nel capitolo precedente. Come però spiegato nel capitolo 1, nel panorama delle applicazioni mobile esistono molte tecnologie valide per la loro realizzazione ed è quindi necessario ponderare la propria scelta e avere ben chiari alcuni aspetti fondamentali che possono incidere su una determinata direzione piuttosto che un'altra. Innanzitutto bisogna valutare se è necessario che la propria app funzioni su diversi sistemi ed anche le tempistiche e i costi che l'implementazione in due linguaggi può comportare. Inoltre è necessaria lungimiranza, per capire come si muoverà il mercato in tempi futuri e quale tecnologia sarà il migliore investimento.

6.1 Considerazioni sulle app platform-independent

Sempre più spesso è possibile trovare nel PlayStore o nell'AppStore applicazioni che non sono state realizzate in modo nativo, ma mediante le tecnologie alternative spiegate nel capitolo 1.1. Se fino a pochi anni fa realizzare un'applicazione platform-independent non garantiva prestazioni e risultato paragonabili alla corrispettiva nativa, oggi si può dire che costituisce in tutto e per tutto un'alternativa valida ed efficace. Valida perchè i framework presenti permettono di creare un app che sia pressochè identica a quella nativa, sia per utilizzo di sensori e plugin, sia

per prestazioni, efficace perchè il time-to-market delle applicazioni e il loro costo è notevolmente ridotto.

6.2 Considerazioni su Dart

Come linguaggio Dart è sicuramente un linguaggio ben strutturato e davvero efficiente, probabilmente dovuto al fatto che è stata la comunità di Google a idearlo e crearlo. Una volta lanciato sul mercato sembrava un progetto fallito, a causa della scarsa diffusione in quanto JavaScript già dominava il web, con l'avvento di Flutter invece Dart si sta prendendo la sua rivincita occupando spazi sempre maggiori nel mercato. Nella mia esperienza personale con Dart ho trovato un linguaggio a oggetti ben fatto, sintatticamente simile al Java e con astrazioni che mi hanno permesso di gestire in maniera semplice e davvero efficiente tutte le situazioni problematiche con cui mi sono scontrato nella realizzazione della parte progettuale di questa tesi. Inoltre il suo doppio approccio di compilazione AOT e JIT permette l'individuazione di errori già in fase di compilazione, semplificando notevolmente il tempo di debugging, senza però privare il linguaggio delle astrazioni che si basano sulla tipizzazione a runtime. Anche rispetto a Kotlin ho trovato Dart un linguaggio più pulito. Entrambi hanno di fatto sintassi molto simile e meccanismi comuni, come la tipizzazione implicita e esplicita. Dall'esperienza che ho avuto con questi due linguaggi penso che Flutter abbia deciso di integrare al suo interno Dart piuttosto che Kotlin poichè già per lo sviluppo Web faceva uso di librerie sviluppate da Google come ad esempio Material e di conseguenza sarebbe risultato più facile realizzare applicazioni in Flutter mediante Dart, piuttosto che dover creare librerie apposite anche per Kotlin.

6.3 Considerazioni su Flutter

Nonostante la recente nascita, Flutter sta diventando un punto di riferimento per lo sviluppo di applicazioni cross-platform. Sempre più aziende fanno uso di Flutter come strumento di sviluppo, probabilmente a causa della sua capacità di creare app

pressochè native sia per Android che per iOS. Lo stesso Linkedin mostra come, secondo le sue rilevazioni, Flutter sia la skill che sta prendendo piede più velocemente tra gli sviluppatori in questo periodo e che la richiesta delle aziende a riguardo sta crescendo maggiormente rispetto a tutte le altre tecnologie mobile [36]. Sicuramente Flutter è un framework che sta crescendo molto e si sta sviluppando, anche a fronte del fatto che ha alle sue spalle una community molto forte. Sta facendo veri e propri passi da gigante questa tecnologia e me ne sono reso conto io stesso nei mesi di tirocinio. Infatti tra febbraio 2019 e aprile 2019 sono comparsi su Pub moltissimi plugin che, una volta integrati, hanno apportato notevoli migliorie all'applicazione. Questo per sottolineare come Flutter sia costantemente in evoluzione, e come vengano sempre rilasciati nuovi aggiornamenti. L'ultimo aggiornamento ha integrato una nuova funzionalità di Flutter: Flutter Desktop. É ora possibile creare applicazioni desktop con Flutter rendendolo un framework a tutto tondo, dato che ora permette la sviluppo per web, mobile e desktop. Nonostante ciò ci sono ancora migliorie che possono essere apportate a questa tecnologia, per citarne alcune:

- rendere possibile la creazione della componente iOS anche da Windows e la componente Android anche da MacOS. Ad oggi infatti per poter compilare e emulare l'applicazione per ambiente iOS è necessario avere computer Apple o comunque una macchina virtuale che supporti tali sistemi operativi. Uno sviluppo interessante potrebbe essere creare anche un emulatore che sia cross-platform.

- migliorare l'integrazione di caratteristiche native nel linguaggio, in modo da non aver bisogno di aggiungere porzioni di codice nel file Manifest di Android ad esempio per integrare determinate funzionalità come Firebase.

- migliorare la documentazione online. Durante il mio tirocinio ho infatti trovato davvero ben scritte e documentate certe parti di Flutter, mentre altre erano davvero oscure e sconosciute ai più, anche in rete e nei forum, come ad esempio l'utilizzo di Firebase.

6.4 Considerazioni personali

In questa mia esperienza con Flutter posso dire di aver appreso in maniera davvero approfondita le caratteristiche del linguaggio Dart e le varie astrazioni e strutture dati che il framework permette di utilizzare. L'ho apprezzato molto, in parte a causa del fatto che avendo studiato in maniera approfondita Java, non ho avuto difficoltà a passare a questo linguaggio e in parte perchè permette di fare alcune cose in maniera molto efficiente con poca scrittura di codice. Inoltre rispetto a JavaScript apprezzo molto il fatto che sia un linguaggio di per sè fortemente tipizzato, anche se come detto supporta anche la tipizzazione runtime. Avendo già progettato e fatto esperienza in ambiente Android posso dire che Flutter semplifica davvero la scrittura di una applicazione, specialmente poichè permette di lavorare in maniera indipendente rispetto a quei meccanismi che invece sono dei vincoli in Android, come ad esempio la creazione di task appositi per le operazioni long-running. Inoltre l'ho trovato davvero smart come tecnologia, è veloce, leggera e permette la creazione di UI davvero molto elaborate e user friendly. Probabilmente vedendo lo sviluppo che questa tecnologia sta avendo e anche come interesse personale, perseguirò gli studi riguardanti Flutter.

6.5 Sviluppi futuri

Il caso di studio citato nella tesi è solo una prima release dell'applicazione di conseguenza tutto il codice è stato pensato per rendere l'applicazione stessa il più possibile estendibile e personalizzabile a seconda delle esigenze del cliente. Nonostante non sia stato utilizzato il pattern MVC, l'implementazione del pattern BLoC e l'uso della programmazione reattiva ha permesso di scorporare la logica applicativa dalle Widget puramente estetiche, aprendo alla possibilità in futuro di modificare le Widget, ad esempio con differenti layout o colori, senza intaccarne il comportamento delle stesse, o anche viceversa, è possibile in maniera estremamente efficiente modificare la logica applicativa senza dover mettere mano alle Widget. Un esempio di probabile modifica futura sarà estendere le autorizzazioni trattate nel capitolo 5.3.4 anche a clienti e contatti ed è proprio grazie all'uso del pattern BLoC che sarà possibile

farlo in modo efficace senza doversi preoccupare del codice interno alle Widget di interesse.

Ringraziamenti

Desidero ringraziare il professor Alessandro Ricci per i consigli che ha saputo darmi e per la disponibilità concessami nella realizzazione di questa tesi. Ringrazio la Bookmark SRL per la possibilità che mi ha donato e per la bellissima esperienza passata in azienda. Ringrazio la mia famiglia per il prezioso sostegno che mi ha sempre dato. Ringrazio i compagni di corso, il gruppo scout Fo7, la palestra TeamFitness e tutti i miei amici per la vicinanza che mi hanno dimostrato in questi anni.

Bibliografia

[1] Academind. *Flutter tuotial for beginners, Flutter alternatives*. URL: https://www.youtube.com/watch?v=GLSG_Wh_YWc&t=3381s.

[2] Appfigures blog. *iOS Developers Ship 29% Fewer Apps In 2017, The First Ever Decline – And More Trends To Watch*. URL: https://blog.appfigures.com/ios-developers-ship-less-apps-for-first-time/.

[3] Flutter Community. *Flutter: BLoCs with Stream*. URL: https://medium.com/flutter-community/flutter-bloc-with-streams-6ed8d0a63bb8.

[4] Google Community. *Alibaba with Flutter*. URL: https://flutter.dev/showcase.

[5] Google Community. *Firebase*. URL: https://firebase.google.com/docs.

[6] Google Community. *Flutter documentation*. URL: https://flutter.dev/docs.

[7] Progress Software Corporation. *Getting stardet with Nativscript and Angular*. URL: https://docs.nativescript.org/angular/start/introduction.

[8] Progress Software Corporation. *Gradle Hooks*. URL: https://docs.nativescript.org/angular/core-concepts/android-runtime/advanced-topics/gradle-hooks.

[9] Apache Software Foundation. *Apache Cordova*. URL: https://cordova.apache.org/.

[10] Apache Software Foundation. *Apache Cordova overview*. URL: https://cordova.apache.org/docs/en/latest/guide/overview/index.html.

[11] Benchmarks Game. *Dart's algorithm speed.* URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/dart.html.

[12] Google. *Activities and fragments.* URL: https://flutter.dev/docs/get-started/flutter-for/android-devs#how-do-i-listen-to-android-activity-lifecycle-events.

[13] Google. *Asynchronous programming: streams.* URL: https://dart.dev/tutorials/language/streams.

[14] Google. *Google Trends.* URL: https://trends.google.nl/trends/explore?cat=31&q=Xamarin,React%5C%20Native,Cordova,Flutter,ionic.

[15] Google. *Growing Momentum.* URL: https://developers.googleblog.com/2018/09/flutter-release-preview-2-pixel-perfect.html.

[16] Google. *How to use packet.* URL: https://dart.dev/guides/packages.

[17] Google. *Navigate to a new screen and back.* URL: https://flutter.dev/docs/cookbook/navigation/navigation-basics.

[18] Google. *Platform.* URL: https://dart.dev/platforms.

[19] Google. *Technical Overview.* URL: https://flutter.dev/docs/resources/technical-overview.

[20] Marijn Haverbeke. *Eloquent JavaScript.* No Starch Press - Third edition, 2019. ISBN: 1593279507.

[21] Adobe System Inc. *Adobe Phonegap.* URL: https://phonegap.com/.

[22] Facebook Inc. *Native mobile apps with Angular, Vue.js, TypeScript, JavaScript - NativeScript.* URL: https://www.nativescript.org/.

[23] Facebook Inc. *React Native.* URL: https://facebook.github.io/react-native/.

[24] Facebook Inc. *React Native - A framework for building native apps using React.* URL: https://facebook.github.io/react-native/.

[25] Ionic. *Insights, trends, and perspectives from the worldwide Ionic Community.* URL: https://ionicframework.com/survey/2017#.

[26] Ionic. *Ionic - Cross-Platform mobile developement.* URL: https://ionicframework.com/.

[27] Ionic. *What is Apache Cordova?* URL: https://ionicframework.com/enterprise/resources/articles/what-is-apache-cordova.

[28] Ionic. *What is Ionic framework?* URL: https://ionicframework.com/docs/intro.

[29] Microsoft. *Basic Types.* URL: https://www.typescriptlang.org/docs/handbook/basic-types.html.

[30] Microsoft. *Mixins.* URL: https://www.typescriptlang.org/docs/handbook/mixins.html.

[31] Microsoft. *Visual Studio tools for Xamarin.* URL: https://visualstudio.microsoft.com/it/xamarin/.

[32] Microsoft. *Xamarin - Architecture.* URL: https://docs.microsoft.com/it-it/xamarin/android/internals/architecture.

[33] Microsoft. *Xamarin Documenation.* URL: https://visualstudio.microsoft.com/it/xamarin/.

[34] Skia. *Skia Graphic Library.* URL: https://skia.org/.

[35] StackOverflow. *StackOverflow Insights: sondaggio agli sviluppatori.* URL: https://insights.stackoverflow.com/survey/2017#technology-most-loved-dreaded-and-wanted-frameworks-libraries-and-other-technologies.

[36] Application Developer Trend. *LinkedIn Data Says Flutter Is Fastest-Growing Skill Among Software Engineers.* URL: https://adtmag.com/articles/2019/03/29/linkedin-skills.aspx.

[37] Kathy Walrath. *The Event Loop and Dart.* URL: https://dart.dev/articles/archive/event-loop.

[38] WikiBooks. *JavaScript/Object-based programming.* URL: https://en.wikibooks.org/wiki/JavaScript/Object-based_programming.

[39] Wikipedia. *Adobe Phonegap.* URL: https://it.wikipedia.org/wiki/Adobe_PhoneGap.

[40] Wikipedia. *Cross platform software*. URL: https://en.wikipedia.org/wiki/Cross-platform_software.

[41] Wikipedia. *Customer relationship management*. URL: https://it.wikipedia.org/wiki/Customer_relationship_management.

[42] Wikipedia. *Dart (language)*. URL: https://en.wikipedia.org/wiki/Dart_(programming_language).

[43] Wikipedia. *Material Design*. URL: https://it.wikipedia.org/wiki/Material_Design.

[44] Wikipedia. *Mono*. URL: https://it.wikipedia.org/wiki/Mono_(progetto).

[45] Wikipedia. *Multipiattaforma*. URL: https://it.wikipedia.org/wiki/Multipiattaforma.

[46] Wikipedia. *Progressive Web App*. URL: https://it.wikipedia.org/wiki/Progressive_Web_App.

[47] Wikipedia. *Reactive Programming*. URL: https://en.wikipedia.org/wiki/Reactive_programming.

[48] Wikipedia. *Skia graphic engine*. URL: https://en.wikipedia.org/wiki/Skia_Graphics_Engine.

[49] Wikipedia. *TypeScript*. URL: https://it.wikipedia.org/wiki/TypeScript.