Università degli studi
di Modena e Reggio Emilia

Dipartimento di Scienze Fisiche, Informatiche
e Matematiche

Progettazione, Sviluppo e Analisi di un'Applicazione Android nelle Tecnologie Cross-Platform Xamarin e Flutter

Marco Maretti

Corso di Laurea in Informatica

Relatore:
Riccardo Martoglia

Anno Accademico 2020/2021

Ringraziamenti

...

Parole chiave

Cross-Platform
Framework
Xamarin
Flutter
MyTriageApp

Indice

Elenco delle figure

Elenco estratti di codice

Introduzione

L'evoluzione della tecnologia ha portato nelle case e nelle aziende una vasta diversificazione di dispositivi. In questa varietà di dispositivi, lo smartphone rappresenta un fenomeno sociale di massa che stravolge completamente il vecchio pensiero di telefono, creando migliaia di funzionalità attraverso le app.

Proprio per questo diventa importante sviluppare applicazioni mobile in modo rapido ed efficiente, cercando di soddisfare la domanda data da un mercato ormai dominato dai due sistemi operativi mobile, iOS e Android. Molte organizzazioni si rivolgono ad aziende che utilizzano strumenti di sviluppo multipiattaforma, perché consentono di sviluppare app per entrambi i sistemi operativi citati prima, riducendo tempi e costi di sviluppo.

Lo scopo di questo elaborato è studiare due degli strumenti di sviluppo multipiattaforma (anche detti framework *cross-platform*) più utilizzati: Flutter e Xamarin. Per farlo, verrà progettata e sviluppata la stessa applicazione con entrambe le piattaforme, per confrontare le loro diverse caratteristiche e scoprire quali vantaggi o svantaggi comportano.

L'applicazione in questione si chiama *MyTriageApp*, consiste in un semplice software per il tracciamento di persone, senza troppe funzionalità complicate, dato che il vero scopo di questa app è quello di fungere da "cavia" per il testing dei framework.

La tesi è organizzata in sei capitoli che racchiudono le diverse fasi di progettazione, sviluppo e confronto delle due applicazioni.

Nel primo capitolo vengono introdotti il contesto e gli obiettivi dell'applicazione. Successivamente, nel secondo capitolo vengono presentate dettagliatamente le tecnologie utilizzate per lo sviluppo.

Nel terzo capitolo si passa alla fase di progettazione dell'applicazione, specificando le attività e le funzionalità che dovrà avere.

Il quarto e il quinto capitolo racchiudono l'implementazione dell'applicazione

nei rispettivi strumenti multi-piattaforma, mostrando la struttura dei progetti e commentando relativi estratti di codice.

Infine, all'interno del sesto capitolo vengono analizzati e confrontati, tramite diversi criteri, le caratteristiche di Flutter e Xamarin, utilizzando anche le informazioni apprese durante lo sviluppo delle due applicazioni.

Parte I

Il Caso di Studio e le Tecnologie Utilizzate

Capitolo 1

Analisi degli Obiettivi

1.1 Contesto

L'emergenza sanitaria ha condizionato le nostre vite e la routine di molte, se non tutte, persone e aziende, rendendo necessario nelle attività pubbliche il tracciamento delle persone per poter risalire ad una possibile catena di contagi.

Un ambito che è stato sensibilmente riorganizzato è quello sportivo. In quasi tutti gli sport sono stati introdotti dei protocolli sanitari rigidi che intervengono sui comportamenti delle società e dei proprietari degli impianti sportivi. Per fare un esempio concreto, ogni società che affitta un impianto sportivo è tenuta a segnare nome, cognome, orario di entrata-uscita e temperatura di ogni tesserato/spettatore che partecipa all'attività, per poi inoltrare questo tabulato al proprietario dell'impianto sportivo. È ovviamente un'operazione molto ripetitiva e lenta da fare in cartaceo.

Oltre che allo sport, quelle citate sopra sono problematiche che si possono applicare in modo generale ad altri ambiti, in quanto il tracciamento delle persone è un operazione resa ormai obbligatoria anche in ristoranti, negozi, piscine ecc.

Il progetto **MyTriageApp** punta a facilitare ed automatizzare situazioni come queste tramite una semplice ma efficace applicazione per smartphone.

1.2 Scopo dell'Applicazione

Obiettivo dell'applicativo è offrire poche ma semplici funzionalità, creando un servizio generale affidabile.

Come prima funzionalità, il software fornisce supporto all'utente/responsabile all'ingresso dell'impianto, informatizzando totalmente la fase che ho denominato di *Triage*. La fase di Triage è la procedura con la quale l'utente, inizialmente, registra nell'applicazione le informazioni generali di ogni persona entrata nell'edificio (nome, cognome, temperatura e orario di entrata) e successivamente, in modo interattivo, inserisce l'orario di uscita.

L'applicazione poi, offre la possibilità di salvare il tabulato che si è venuto a creare durante la fase di Triage in due modi: *esportandolo* in una tabella all'interno di un file `.pdf`, oppure salvandolo permanentemente all'interno della sezione *Gruppi* dell'utente.

Grazie a questi Gruppi l'utente può accedere velocemente ai dati di persone già note all'impianto e velocizzare ulteriormente il loro Triage.

L'ultima funzionalità principale che l'applicazione mette a disposizione dell'utente è la *Cronologia Esportazioni*: come si può ben intendere, il software tiene traccia di tutte le esportazioni effettuate dall'utente, in modo da poter accedere in qualsiasi momento a vecchi tabulati esportati e rigenerare il corrispondente file `.pdf`.

Concludendo, per rendere affidabili tutte queste funzionalità, l'applicativo deve interagire con un database online, per poter salvare in tempo reale le informazioni e renderle accessibili da più dispositivi.

1.3 Framework di Sviluppo

Quando si progetta un'applicazione, la scelta della piattaforma di sviluppo è uno step fondamentale.

Negli ultimi anni, in conseguenza all'ampliamento del mercato delle app, la programmazione mobile ha avuto una grande crescita. Sono stati ideati, sviluppati e messi a disposizione dei programmatori sempre più strumenti, consentendo agli sviluppatori una più ampia ma anche più ardua scelta.

Da qui nasce l'idea del confronto: sviluppare la stessa applicazione, con le stesse funzionalità su due framework diversi e successivamente analizzare le differenze sia a livello di risultato, quindi tutto ciò che concerne l'applicazione finale (ad esempio *design, performance, user experience*, ecc.), sia dal punto di vista della programmazione, quindi le varie scelte che ogni piattaforma di sviluppo

ti permette o ti obbliga a fare. In questo modo sarò in grado di identificare vantaggi e svantaggi dei framework utilizzati e capire quali strumenti sono più o meno adatti a progetti di questo tipo.

Capitolo 2

Tecnologie Utilizzate

2.1 Introduzione

In un mercato così ampio come quello delle applicazioni mobile, è importante offrire il proprio prodotto a tutti gli utenti, senza tagliare fuori possibili acquirenti. Questo vuol dire sviluppare il proprio software per diversi sistemi operativi che eseguono su diversi tipi di hardware. Nel caso della programmazione mobile, i due sistemi operativi che dominano il mercato degli ultimi decenni sono *iOS*, il SO dei dispositivi mobile marcati Apple, e *Android*, sviluppato da Google e basato su kernel linux.

Questi sistemi operativi sono estremamente diversi e richiedono tecnologie e conoscenze differenti per costruire un'applicazione.

2.1.1 Sviluppo Nativo

Con il termine *applicazione nativa* si intende un'applicazione mobile costruita esclusivamente per un singolo sistema operativo utilizzando strumenti specifici ad esso.

É possibile sviluppare un'applicazione nativa Android utilizzando *Java, Kotlin* e *C++* come linguaggi di programmazione. Inoltre Google mette a disposizione degli sviluppatori strumenti avanzati come l'Android Software Development Kit, o *SDK*.

Analogamente, è possibile progettare un'applicazione nativa iOS utilizzando linguaggi come l'*Objective-C* e *Swift*. Anche Apple offre diversi strumenti e developement kit per la programmazione nativa, ma a differenza di Android,

il sistema operativo non è open source, dunque è ristretto solo ai dispositivi hardware della Apple, rendendo necessario l'utilizzo di un *Macintosh* per lo sviluppo nativo.

Questo tipo di sviluppo porta enormi vantaggi: grazie alla sua enorme verticalità sfrutta al meglio le risorse hardware incrementando le performance, accede anticipatamente a tutte le nuove feature del sistema operativo e offre sempre un design consistente, anche su diversi dispositivi.

Costruire un software nativo però non è un processo semplice e, nonostante tutti questi vantaggi, spesso le aziende non hanno sufficienti risorse temporali ed economiche per sviluppare la stessa applicazione nativamente sia su Android che iOS.

2.1.2 Sviluppo Cross-Platform

La soluzione a questo problema sono i framework *cross-platform*: sono strumenti che si pongono ad un livello d'astrazione superiore a quello dei framework nativi, permettendo lo sviluppo di un'applicazione attraverso un unico *codebase*, che può essere poi compilato e distribuito separatamente su sistemi operativi diversi.

In questo modo è necessario solo un ciclo di sviluppo per la creazione di un app che esegue su più piattaforme, risparmiando tempo e denaro. Aggiungere un livello di astrazione superiore però comporta inevitabili svantaggi, quali calo delle performance e accesso limitato alle funzionalità del dispositivo.

Per scegliere tra sviluppo nativo e cross-platform bisogna considerare diversi aspetti (vedi tabella 2.1): se si vuole costruire un'applicazione performante, facilmente aggiornabile e con accesso totale alle risorse del dispositivo, lo sviluppo nativo è la strada migliore. Al contrario, se si vuole progettare un'applicazione eseguibile su più sistemi operativi, in relativamente poco tempo e senza funzionalità estremamente complicate, le piattaforme cross-platform fanno al caso vostro.

Parametro	**Applicazione Nativa**	**Applicazione Cross-Platform**
Costi di Sviluppo	Elevati	Relativamente bassi
Utilizzo del Codice	Funziona su una sola piattaforma	Codice unico che può essere utilizzato su più piattaforme
Accesso al dispositivo	L'SDK della piattaforma garantisce accesso completo alle API del dispositivo	Accesso completo alle API non garantito
Interfaccia Utente (UI)	Consistente con i componenti UI del dispositivo	Consistenza limitata con i componenti UI del dispositivo
Performance	Senza limiti dettati dal framework	Può essere elevata, ma rallentamenti e problemi di compatibilità hardware non sono rari

Tabella 2.1: Confronto tra applicazioni Native e Cross-Platform [11].

2.2 Xamarin

Xamarin [22] è una piattaforma open source di proprietà Microsoft per la compilazione di applicazioni Android ed iOS. Utilizza interamente il linguaggio di programmazione C# ed estende il framework *.NET* con strumenti e librerie dedicate alla progettazione di app mobile. La logica di business, l'interfaccia grafica e la comunicazione con le API del sistema operativo vengono scritte interamente in C#, con l'aiuto del linguaggio di markup estendibile *XAML* per la codifica della parte dinamica dell'interfaccia utente.

Xamarin offre diversi framework sia per la programmazione diretta a una singola piattaforma sia per quella multi-piattaforma, che sono:

- **Xamarin.Android**, per sviluppare in C# applicazioni native Android.
- **Xamarin.iOS**, per sviluppare in C# applicazioni native iOS.
- **Xamarin.Forms**, per scrivere interfacce utente iOS e Android attraverso un unico codebase.

2.2.1 Architettura

Vediamo brevemente come sono costruiti i programmi in Xamarin: tutti i progetti, che siano Xamarin.Android, Xamarin.iOS o Xamarin.Forms sono costruiti un livello sopra **MONO**, una versione open-source di .NET che esegue su sistemi operativi non Windows, quindi inclusi quelli Linux e Unix-like.

MONO è un progetto nato agli albori della programmazione mobile, ma non è mai stato molto popolare. Inizialmente, era composto da due parti, *Mono Android* e *Mono Touch*, poi successivamente rinominate come Xamarin.Android e Xamarin.iOS.

Le applicazioni Xamarin.Android [24] vengono compilate da C# in un linguaggio intermedio (**IL**) che, a sua volta, quando l'applicazione esegue viene compilato in modalità *Just-in-Time* (**JIT**) nel linguaggio assembly nativo. Le applicazioni eseguono all'interno dell'ambiente di esecuzione di MONO, che procede pari passo con la macchina virtuale **Android Runtime** (**ART)**: questi due ambienti, per comunicare tra loro, sfruttano dei *wrapper* (**Managed Callable Wrapper** ed **Android Callable Wrapper**) che collegano i namespace di .NET a quelli di *Android.** e *Java.** (vedi figura 2.1).

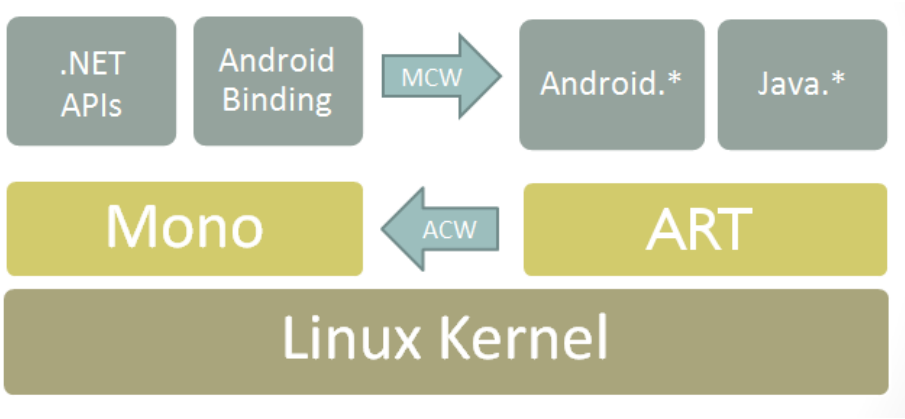


Figura 2.1: Architettura applicazione Xamarin.Android [24]

Le applicazioni Xamarin.iOS [19] funzionano diversamente: il codice C# viene totalemente compilato in modalità **Ahead-of-Time** (**AOT**) nel linguaggio assembly ARM nativo. Questo codice esegue a pari passo con l'ambiente di esecuzione **Objective-C**, a cui si appoggiano le API native di iOS. I due ecosistemi, per comunicare tra loro, utilizzano dei **binding**, ovvero dei collegamenti che permettono alle API iOS di essere usate in Xamarin (vedi figura 2.2).

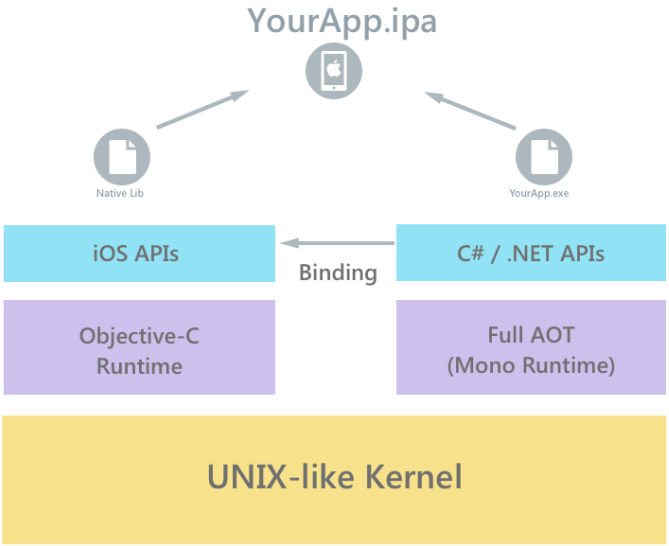


Figura 2.2: Architettura applicazione Xamarin.iOS [19]

2.2.2 Xamarin.Forms

Sia Xamarin.Android che Xamarin.iOS sono strumenti per creare applicazioni native, ma cose offre Xamarin per il cross-platform?

Xamarin.Forms [23] è un **User Interface Framework** che permette di condividere layout, design e logica di business su più piattaforme.

Per riuscirci, espone delle API per creare elementi UI tra le piattaforme. Queste API possono essere implementate sia in XAML che in C#, utilizzando tecniche di **databinding** come il pattern *Model-View-ViewModel*.

Model-View-ViewModel [18] (**MVVM**) è un pattern architetturale che forza la separazione dei tre livelli software del codebase condiviso: l'interfaccia utente XAML, detta *View*, i dati sottostanti, chiamati *Model*, ed infine un livello intermedio, chiamato *ViewModel* (vedi dettagli implementativi nella sezione 4.4.2).

Come si può ben notare in figura 2.3, a tempo di esecuzione, Xamarin.Forms renderizza gli elementi UI cross-platform in elementi nativi di Xamarin.Android e Xamarin.iOS: in questo modo si ha design, feeling e performance native con il vantaggio di un unico codebase.

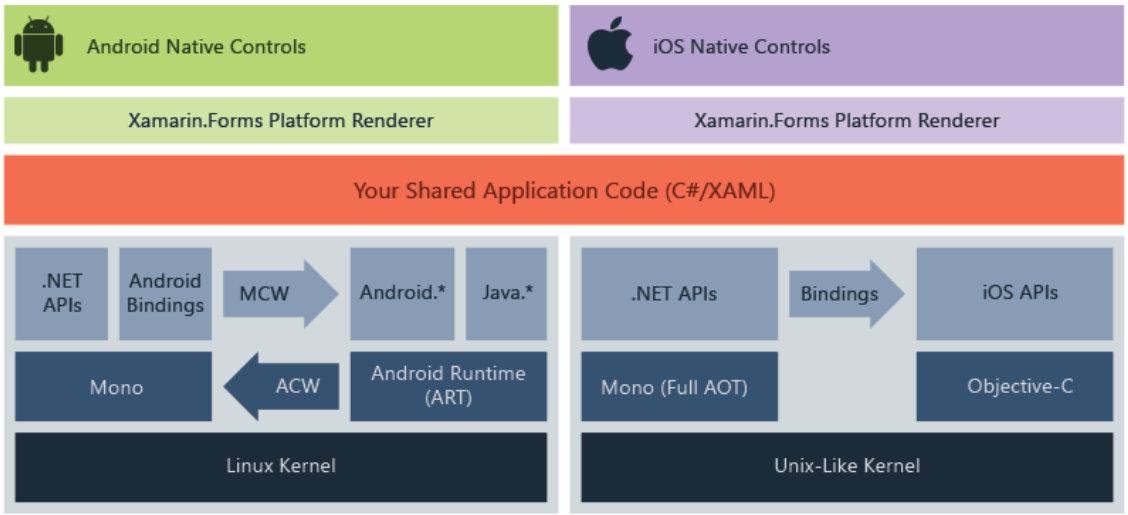


Figura 2.3: Architettura applicazione Xamarin.Forms [23]

2.2.3 Struttura di un Progetto

I progetti delle applicazioni Xamarin.Forms sono generalmente costituiti da un primo progetto di libreria .NET standard e da un progetto per ogni piattaforma su cui si vuole sviluppare.

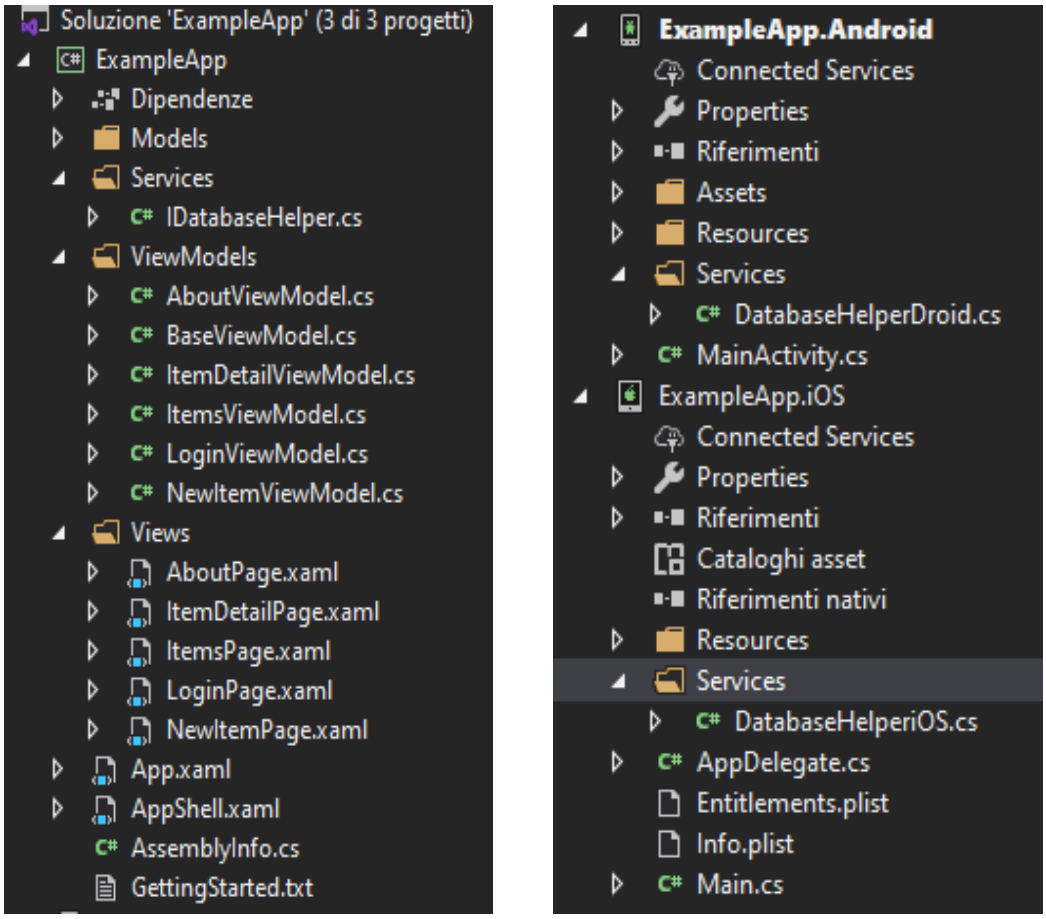
Il progetto di libreria .NET contiene il codice condiviso, ovvero le *View* in XAML e la logica di business in C#, mentre i progetti delle piattaforme contengono codice specifico che non può essere condiviso.

In figura 2.4 viene rappresentata la struttura di un progetto Xamarin.Forms: possiamo notare che, all'interno del progetto .NET condiviso, oltre alle cartelle per l'architettura MVVM, è presente anche una cartella per i **servizi**.

La cartella dei servizi nel progetto .NET condiviso ha un compito speciale: essa non contiene la vera implementazione dei servizi, ma contiene delle classi **interfaccia** (`IDatabaseHelper.cs`) che definiscono solamente l'intestazione dei metodi. Successivamente, all'interno della cartella servizi dei progetti specifici, le classi erediteranno l'interfaccia del progetto condiviso e implementeranno i metodi in modo diverso (`DatabaseHelperDroid.cs` e `DatabaseHelperiOS.cs`). In questo modo, all'interno del progetto .NET condiviso sarà possibile usufruire dei servizi creando un'istanza dell'interfaccia (grazie alla libreria `Xamarin.Forms.DependencyService`) in questo modo:

```
1 public LoginViewModel(){
2     ...
3     databaseHelper = DependencyService.Get<IDatabaseHelper>();
4     ...
5 }
```

Sarà poi compito di Xamarin, a tempo di esecuzione, decidere (a seconda del sistema operativo su cui sta eseguendo l'applicazione) quale implementazione dell'interfaccia usare.



(a) Progetto libreria .NET (b) Progetti specifici Android e iOS

Figura 2.4: Esempio struttura di un progetto Xamarin.Forms

2.3 Flutter

Flutter [16] è un **User Interface Framework** gratuito e open-source, sviluppato e rilasciato da Google nel maggio 2017. Permette di creare applicazioni compilate nativamente per più piattaforme (iOS e Android) con un solo codebase.

Per sviluppare in Flutter si utilizza il linguaggio di programmazione *Dart* (sintassi simile a *JavaScript*), anch'esso di casa Google. *Dart* è un linguaggio orientato agli oggetti, che si focalizza principalmente sullo sviluppo front-end di applicazioni mobile e web application.

Flutter è composto da un **SDK**, ovvero l'insieme di strumenti per sviluppare e compilare il codice in assembly nativo, e da una libreria UI basata su **Widget**, ovvero elementi di interfaccia utente personalizzabili. É importante ricordare che Flutter è un framework, dunque non basta installare Flutter per poter creare ed eseguire un applicazione: per lo sviluppo di app Android, Flutter si affida ad **Android Studio** [4] per risolvere le dipendenze alla piattaforma; mentre per lo sviluppo di app iOS, vi è comunque bisogno di un Mac con **XCode** [3] installato.

L'obiettivo di Flutter è permettere agli sviluppatori di creare applicazioni ad alte prestazioni con look nativo su diverse piattaforme, cercando di condividere più codice possibile.

2.3.1 Architettura

Flutter è stato progettato come un sistema a livelli estendibile [17]. Come si può vedere in figura 2.5, è formato da un insieme di librerie, raggruppate in livelli e tra loro indipendenti, ma ciascuna dipendente dal livello sottostante.

Per il sistema operativo, le applicazioni Flutter sono tutte configurate e impacchettate allo stesso modo, come qualsiasi altra applicazione nativa. Questo è possibile grazie al livello più basso chiamato **Platform-Specific Embedder**: esso si occupa di comunicare con il sistema operativo su cui deve eseguire l'applicazione. L'Embedder, a seconda della piattaforma, è scritto in diversi linguaggi: *Java* e *C++* per Android, *Objective-C* per iOS e macOS, *C++* per Windows e Linux.

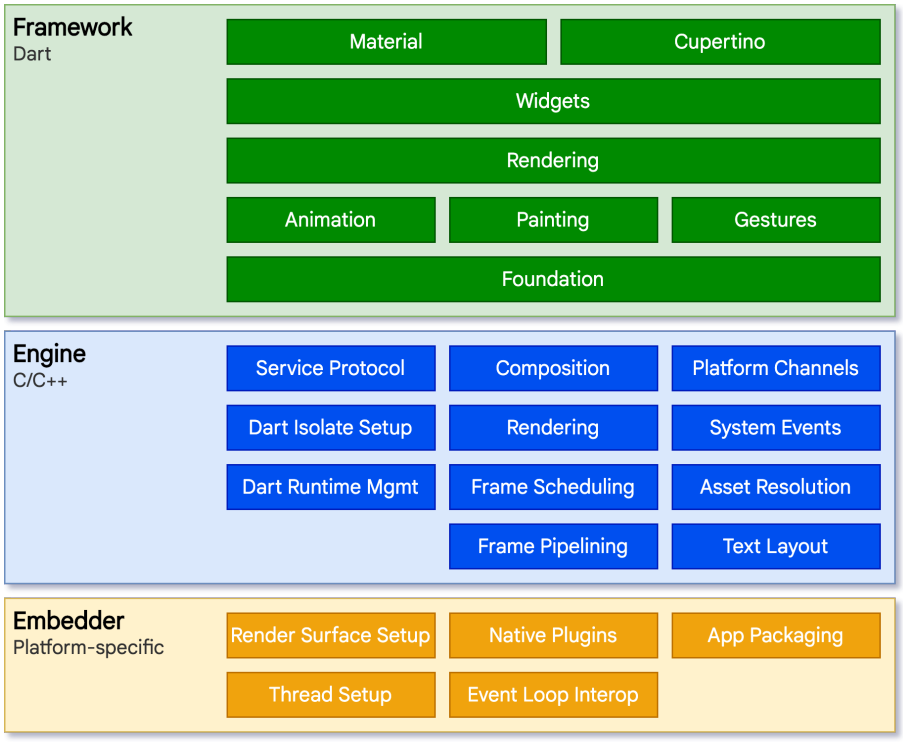


Figura 2.5: Architettura a livelli di Flutter [17]

Al cuore di Flutter troviamo il **Flutter Engine**, scritto per la maggior parte in *C++*, che è responsabile della renderizzazione di nuovi componenti ogni volta che un nuovo frame deve essere disegnato. Le primitive di basso livello dell'engine vengono esposte al framework Flutter attraverso la libreria `dart:ui`, che racchiude il codice *C++* sottostante in classi *Dart*.

Arriviamo dunque al livello più alto del sistema, quello con cui tipicamente gli sviluppatori interagiscono, il **Framework**. É formato da un ricco insieme di librerie, layout e sottolivelli di astrazione. Partendo dal basso e andando verso l'alto, i principali componenti sono:

- Classi base di fondamenta (**foundational classes**) e servizi per la costruzione di blocchi come *animation, painting e gesture*.

- Livello di **rendering**, fornisce un astrazione per lavorare con i layout. Grazie a questo livello, è possibile creare un **albero** di oggetti renderizzabili.

- Livello dei **widget**. Ogni oggetto del livello di rendering ha una corrispondente classe nel livello di widget che aggiunge la possibilità di combinare e personalizzare classi.
- Librerie **Material** e **Cupertino**, utilizzano composizioni di primitive del livello widget per implementare il design Material [6] (per Android) e Cupertino [15] (per iOS).

Di per sé, il framework Flutter è relativamente piccolo; la maggior parte delle feature ad alto livello infatti vengono implementate attraverso pacchetti e plugin esterni.

2.3.2 Widget

Come precedentemente accennato, Flutter utilizza i widget come unità di composizione. Ogni widget è un elemento costruttivo dell'interfaccia utente dell'app.

I widget formano una gerarchia basata sulla composizione. Ogni elemento è annidato all'interno dell'elemento padre, da cui può ricevere il contesto. La struttura procede in questo modo fino all'elemento radice: il contenitore dell'app Flutter (tipicamente un widget di tipo `MaterialApp` o `CupertinoApp`).

Le applicazioni aggiornano l'interfaccia utente in risposta ad eventi (ad esempio un'interazione dell'utente) istruendo il framework a rimpiazzare un widget della gerarchia con uno nuovo. Per fare ciò, il framework compara il vecchio e il nuovo widget e poi aggiorna efficientemente l'interfaccia.

A questo proposito, Flutter introduce due grandi classi per la creazione di widget: *stateful* e *stateless* widget.

Molti widget non hanno bisogno di mantenere uno stato **mutabile**, ovvero non hanno proprietà che devono cambiare nel corso del tempo (ad esempio un icona o una etichetta). Questi widget sono sottoclassi della classe `StatelessWidget`.

Al contrario, se le caratteristiche uniche del widget devono cambiare in corrispondenza di interazioni dell'utente o di altri fattori, il widget viene chiamato *stateful*.

Ad esempio, se avessimo un widget che rappresenta un contatore, ed ogni volta che l'utente preme un bottone il contatore deve aumentare, il valore del contatore sarebbe parte dello *stato* del widget. Quando il valore del contatore cambia, parte del widget deve essere ricostruita per aggiornare la visualizzazione nella UI.

Questi tipi di widget sono sottoclassi della classe `StatefulWidget`, e salvano il proprio stato mutabile in una classe separata che eredita da `State` (vedi sezione 5.4 per i dettagli implementativi).

2.3.3 Struttura di un Progetto

La documentazione di Flutter non obbliga a seguire uno specifico pattern per la strutturazione del progetto, ma è comunque buona norma organizzare le cartelle in modo tale da rendere il codice facilmente leggibile e interpretabile.

In figura 2.6 è rappresentata la struttura di un progetto Flutter appena creato. Entrando nel dettaglio troviamo:

- Le cartelle `android/..` ed `ios/..`, contenenti tutti i file e le cartelle necessarie all'esecuzione dell'applicazione su piattaforme Android ed iOS. Solitamente non è necessario modificare nulla al loro interno.
- La cartella `lib/..`, probabilmente la più importante di tutto il progetto. Contiene la maggior parte del codice *Dart* dell'applicazione e solitamente viene organizzato in questo modo:
  - Cartella `models/..`, contiene i modelli del database che l'app utilizza;
  - Cartella `services/..`, contiene le classi che implementano la logica dei servizi che interagiscono con parti esterne all'applicazione;
  - Cartella `screens/..`, contiene le classi che verranno utilizzate per costruire l'interfaccia utente.
  - Cartella `widgets/..`, contiene le classi di widget personalizzati che vengono usati ripetutamente all'interno dell'applicazione.
  - Il file `main.dart`, solitamente l'entrypoint d'esecuzione del progetto.
- La cartella `test/..`, contiene le classi per il testing dell'applicazione.
- Il file `pubspec.lock`, generato da Flutter e non modificabile, contiene tutte le configurazioni e le dipendenze del progetto.
- Il file `pubspec.yaml`, utilizzato per aggiungere nuove configurazioni e dipendenze a pacchetti esterni.

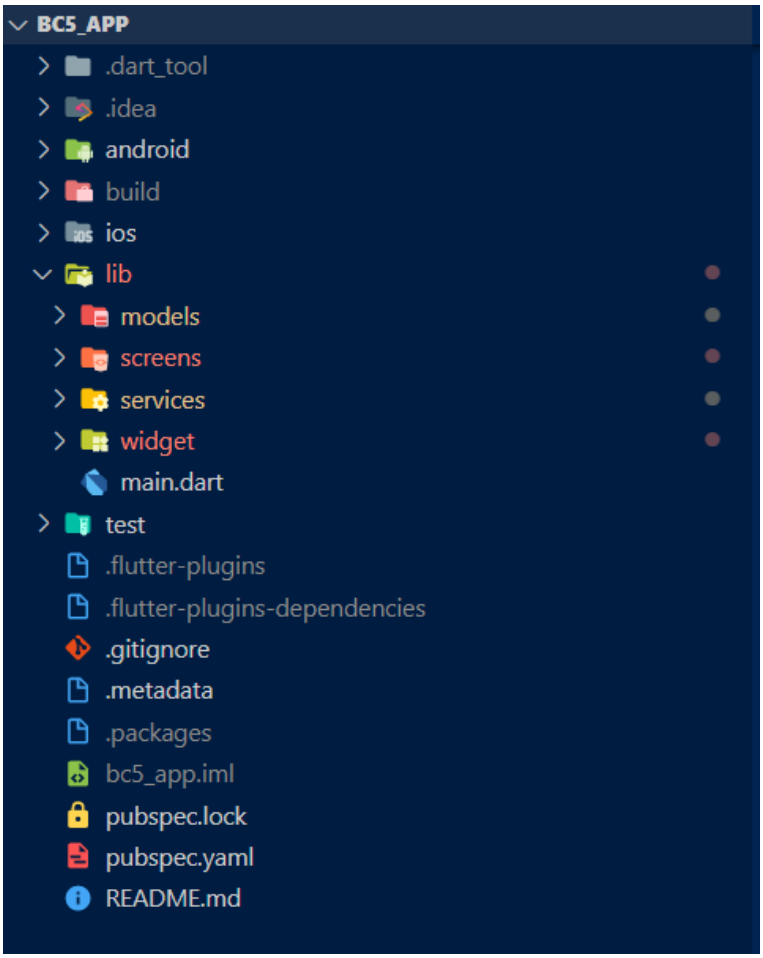


Figura 2.6: Esempio struttura di un progetto Flutter

2.4 Firebase

Firebase [5] è una piattaforma **Backend-as-a-Service** (*BaaS*) creata da Google nel 2011. Fornisce una vasta gamma di strumenti e servizi per migliorare lo sviluppo (lato server) di applicazioni mobile e web.

Una piattaforma *BaaS* automatizza il lato backend di sviluppo di un applicazione, sollevando gli sviluppatori dai problemi di gestione e manutenzione diretta dei server. Questo tipo di modello backend si adatta perfettamente alle esigenze del progetto *MyTriageApp*.

Tra gli strumenti che Firebase offre possiamo trovare:

- **Firebase Authentication** (vedi approfondimento sezione 2.4.1), per l'autenticazione tramite credenziali;
- **Cloud Firestore** (vedi approfondimento sezione 2.4.2), database NoSQL flessibile e scalabile per archiviare e sincronizzare dati.
- **RealTime Database**, per il salvataggio di dati in formato JSON in tempo reale;
- Altri servizi di **storage**, **hosting** e **machine learning**;

Per poter utilizzare Firebase, per prima cosa bisogna creare un progetto. All'interno del progetto poi, è possibile collegare le proprie applicazioni iOS, Android o Web che vogliono usufruire dei servizi.

2.4.1 Autenticazione

L'autenticazione dell'utente è il primo passo importante all'interno del flusso d'utilizzo dell'applicazione. Firebase Authentication semplifica questo step, fornendo agli sviluppatori delle API semplici ed intuitive.

Entrando nel dettaglio, è possibile scegliere i metodi di accesso dell'utente (ad es. autenticazione con email e password, oppure tramite provider), impostare i metodi di modifica e verifica delle mail di nuovi utenti (ad es. tramite mail di verifica o SMS) e di reimpostazione password.

Ad ogni nuovo utente registrato Firebase assegna un *UID*, ovvero una stringa alfanumerica di 32 caratteri che identifica univocamente l'utente. Dalla console di Firebase, come mostrato in figura 2.7, è possibile visualizzare e gestire ogni utente registrato.

Questo *UID* è molto importante perché può essere utilizzato all'interno di un database per recuperare altre informazioni collegate all'utente (ad es. i suoi Gruppi, Esportazioni ecc.).

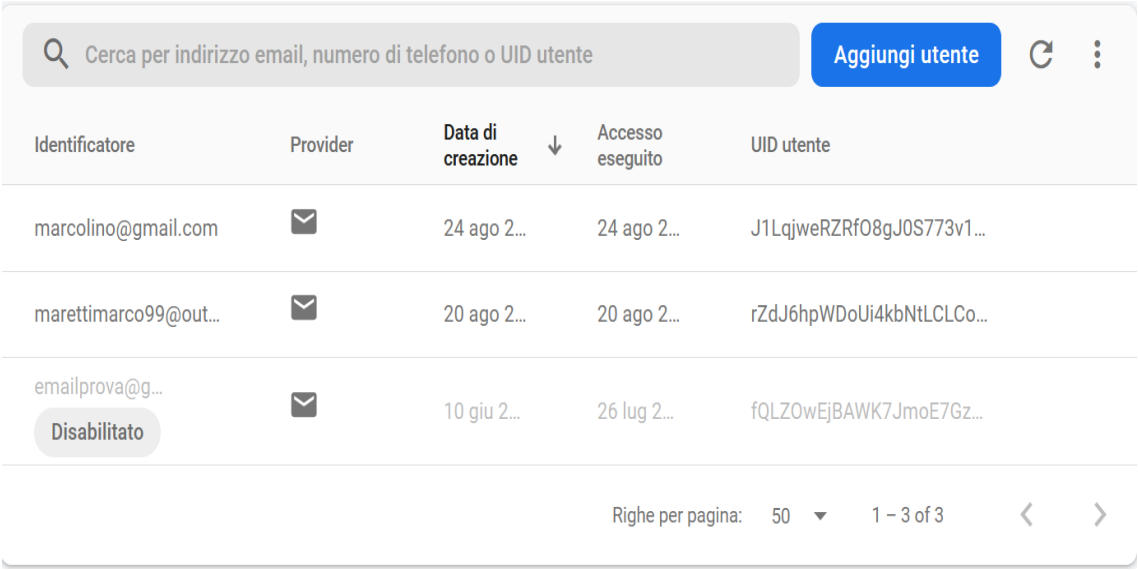


Figura 2.7: Interfaccia console di gestione utenti di Firebase

2.4.2 Cloud Firestore

Cloud Firestore è un database NoSQL, ospitato nel cloud, a cui le applicazioni iOS e Android possono accedere direttamente tramite SDK nativi.

I database NoSQL sono database **non relazionali**, ovvero le informazioni al loro interno sono immagazzinate in *file* e non in *tabelle*. Il principale vantaggio di questo modello è che tutte le informazioni utili all'applicazione sono all'interno di un unico file e non sparse in diverse tabelle del database, dunque si eliminano tutti i possibili problemi di inconsistenza nella ricostruzione dei dati.

Seguendo questo modello, in Cloud Firestore è possibile archiviare dati in **documenti** che contengono campi mappati a valori. Questi documenti sono salvati in **raccolte**, che non sono altro che contenitori utili per organizzare i dati e creare query.

(a) Documento di un utente (b) Raccolta documenti utenti

Figura 2.8: Esempio struttura di Cloud Firestore Database

Dalla struttura mostrata in figura 2.8 possiamo subito notare come i documenti assomiglino a file `.json`. In realtà esistono alcune differenze (ad es. i documenti hanno dimensioni limitate a 1 MB), ma generalmente è possibile trattarli come record JSON.

I documenti supportano diversi tipi di dati: stringhe, numeri semplici, complessi ed oggetti nidificati. Cloud Firestore è privo di schemi, quindi si ha completa libertà su quali campi inserire in ogni documento e quale tipo di dato utilizzare per essi.

É possibile creare delle sottoraccolte all'interno dei documenti, in modo da creare strutture gerarchiche più avanzate. I nomi dei documenti all'interno di raccolte e sottoraccolte devono essere univoci: si possono utilizzare chiavi personali oppure lasciare il compito di generare ID automatici direttamente al database.

Interrogare il database Cloud Firestore è molto semplice: si possono creare sia query superficiali, che puntano a recuperare i dati di intere raccolte o sottoraccolte, sia query più specifiche, che vanno a recuperare dati a livello di singoli documenti. Inoltre, per ottimizzare le performance delle query, Firestore permette di creare indici singoli e composti.

Molto importanti nel funzionamento di Cloud Firestore sono le **regole di sicurezza**: consentono di limitare e controllare l'accesso ai documenti e raccolte nel database. Le regole vengono definite attraverso una sintassi specifica (descritta nella documentazione [5]) tramite la console di Firebase. In figura 2.9 possiamo vedere un esempio di regola che vieta l'accesso in scrittura e lettura a

richieste con data successiva al primo Novembre 2021.

```
rules_version = '2';
service cloud.firestore {   match /databases/{database}/documents {
     match /{document=**} {
       allow read, write: if
            request.time <= timestamp.date(2021,11,11);
   }   } }
```

Figura 2.9: Esempio regola di sicurezza per Cloud Firestore Database

2.4.3 Interazione con Flutter e Xamarin

Il flusso di interazione con i servizi Firebase, nonostante la varietà di librerie per ogni framework di sviluppo, rimane simile tra i diversi linguaggi di programmazione.

Prima di poter utilizzare qualsiasi servizio Firebase però, è necessario **inizializzare la connessione**: in Xamarin tramite la libreria `Xamarin.Firebase.Core` [27] e in Flutter tramite il package `firebase_core` [9].

Per poter comunicare all'applicazione a quale *backend* connettersi abbiamo bisogno del file `google-services.json` che è stato generato durante il collegamento dell'applicazione all'interno del progetto Firebase (vedi introduzione sezione 2.4).

Le librerie "*core*" andranno a recuperare in questo file tutte le informazioni necessarie all'inizializzazione della connessione. Il file deve essere posizionato nel progetto Flutter all'interno della cartella `projectname/android/app/..` e nel progetto Xamarin.Forms all'interno della cartella radice dei rispettivi sottoprogetti nativi.

Per quanto riguarda l'autenticazione, Xamarin e Flutter utilizzano rispettivamente le librerie `Xamarin.Firebase.Auth` [26] e `firebase_auth` [7] (approfondimenti sull'implementazione nelle sezioni 4.3.1 e 5.3.1).

Mentre per l'interazione con il database Cloud Firestore, Xamarin utilizza la libreria `Xamarin.Firebase.Firestore` [25] e Flutter il package `cloud_firestore` [8] (approfondimenti sull'implementazione nelle sezioni 4.3.2 e 5.3.2).

Parte II

Progettazione, Implementazione e Confronto

Capitolo 3

Progettazione

3.1 Analisi dei Requisiti

L'analisi dei requisiti è la fase iniziale più importante del ciclo di vita di un software. In questa fase si specificano e analizzano i requisiti, ovvero le caratteristiche e le funzionalità che l'utente si aspetta di trovare nell'applicazione.

Lo scopo di questo processo è identificare tutti e soli i requisiti rilevanti, definendoli in modo coerente, non ambiguo e indipendente dall'architettura, linguaggio o piattaforma utilizzata. Nel caso del progetto *MyTriageApp* è importante che, soprattutto per coerenza del confronto finale tra i framework di sviluppo, tutti i requisiti siano presenti in entrambe le applicazioni sviluppate.

Generalmente possiamo distinguere due tipi di requisiti: quelli *funzionali*, che descrivono cosa deve fare il software e quali servizi offre, e quelli *non funzionali*, che rappresentano vincoli legati alle proprietà che deve avere il prodotto finale.

All'interno di questo capitolo entreremo nel dettaglio solamente dei requisiti funzionali, dato che il software prodotto dovrà avere solo un unico, ma importantissimo, requisito non funzionale: la **portabilità** dell'applicazione finale su due sistemi operativi (Android ed iOS).

3.1.1 Requisiti Funzionali

Per poter specificare i requisiti funzionali dell'applicazione dobbiamo prima capire quali sono le operazioni principali che deve supportare. A questo proposito, i servizi fondamentali che l'applicazione deve avere sono:

- **Registrazione**: nel caso in cui sia la prima volta che l'utente utilizza l'app, il software deve permettere la creazione di un nuovo account con le informazioni dell'utente.

- **Autenticazione**: per poter accedere al resto delle funzionalità l'utente deve poter autenticare la propria identità tramite e-mail e password (l'applicazione **NON** offre alcun tipo di servizio a utenti anonimi).

- **Creazione, modifica e eliminazione di una scheda Triage**: offrire un interfaccia con cui l'utente può inserire le informazioni generali di ogni persona entrata ed uscita dall'impianto, raggruppandole in una *scheda Triage*.

- **Esportazione di una scheda Triage**: possibilità di generare e salvare nella memoria interna del dispositivo un file `.pdf` relativo alla scheda Triage creata.

- **Creazione e gestione di Gruppi**: dopo aver creato una scheda Triage, poter salvare le informazioni relative alle persone entrate ed uscite in un oggetto chiamato *Gruppo*. L'applicazione deve inoltre fornire un'interfaccia per visualizzare ed eliminare i Gruppi di un utente.

- **Generazione di una scheda Triage da un Gruppo**: possibilità di creare una scheda Triage nuova con all'interno tutte le informazioni delle persone appartenenti a un Gruppo.

- **Cronologia delle Esportazioni**: l'applicazione deve disporre un elenco con le informazioni di tutte le esportazioni effettuate dall'utente e permettere di rigenerare il relativo file.

3.1.2 Casi D'uso

Il diagramma dei casi d'uso [2] è un tipo di diagramma UML (*Unified Modeling Language*) comportamentale utilizzato per la descrizione e l'analisi dei ruoli e delle funzioni offerte da un sistema.

É il primo diagramma ad essere creato durante l'analisi dei requisiti ed è particolarmente utile per avere una visione ad alto livello delle interazioni del sistema.

In figura 3.1 è rappresentato il diagramma dei casi d'uso del progetto *MyTriageApp*, utilizzando come casi d'uso i requisiti funzionali della sezione 3.1.1.

Il diagramma ha un solo attore, lo *User*, che si relaziona con sei funzionalità principali dell'applicazione, tre delle quali relative alla parte di autenticazione del software, che sono:

- *Login*, si occupa dell'autenticazione di un utente già registrato.
- *Logout*, include il caso d'uso *Login*, perché può essere effettuato solo se effettivamente l'utente prima si è autenticato.
- *Registrazione*, caso in cui un utente non sia ancora registrato nel sistema.

Le rimanenti funzionalità riguardano le caratteristiche più importanti del sistema. Tutte includono il caso d'uso del *Login* (l'applicazione non offre servizi ad utenti anonimi) e sono:

- *Creazione Scheda Triage*, caso in cui l'utente voglia creare una nuova scheda. Estende la possibilità di aggiungere una persona entrata nella scheda e anche di eliminarla. Inoltre, estende i casi in cui l'utente voglia esportare la scheda oppure creare un nuovo gruppo con le informazioni delle persone attualmente inserite nella scheda.
- *Visualizzazione Gruppi*, caso in cui l'utente voglia visualizzare i suoi gruppi. Estende la funzionalità di dettaglio di un gruppo, che può essere eliminato oppure usato per creare una nuova scheda triage. Quest'ultimo caso include il caso di *Creazione Scheda Triage*.
- *Visualizzazione Esportazioni*, caso in cui l'utente voglia visualizzare la cronologia delle esportazioni che ha effettuato. Estende la funzionalità di dettaglio di una singola esportazione, e anche il caso in cui l'utente ne voglia rigenerare il file.

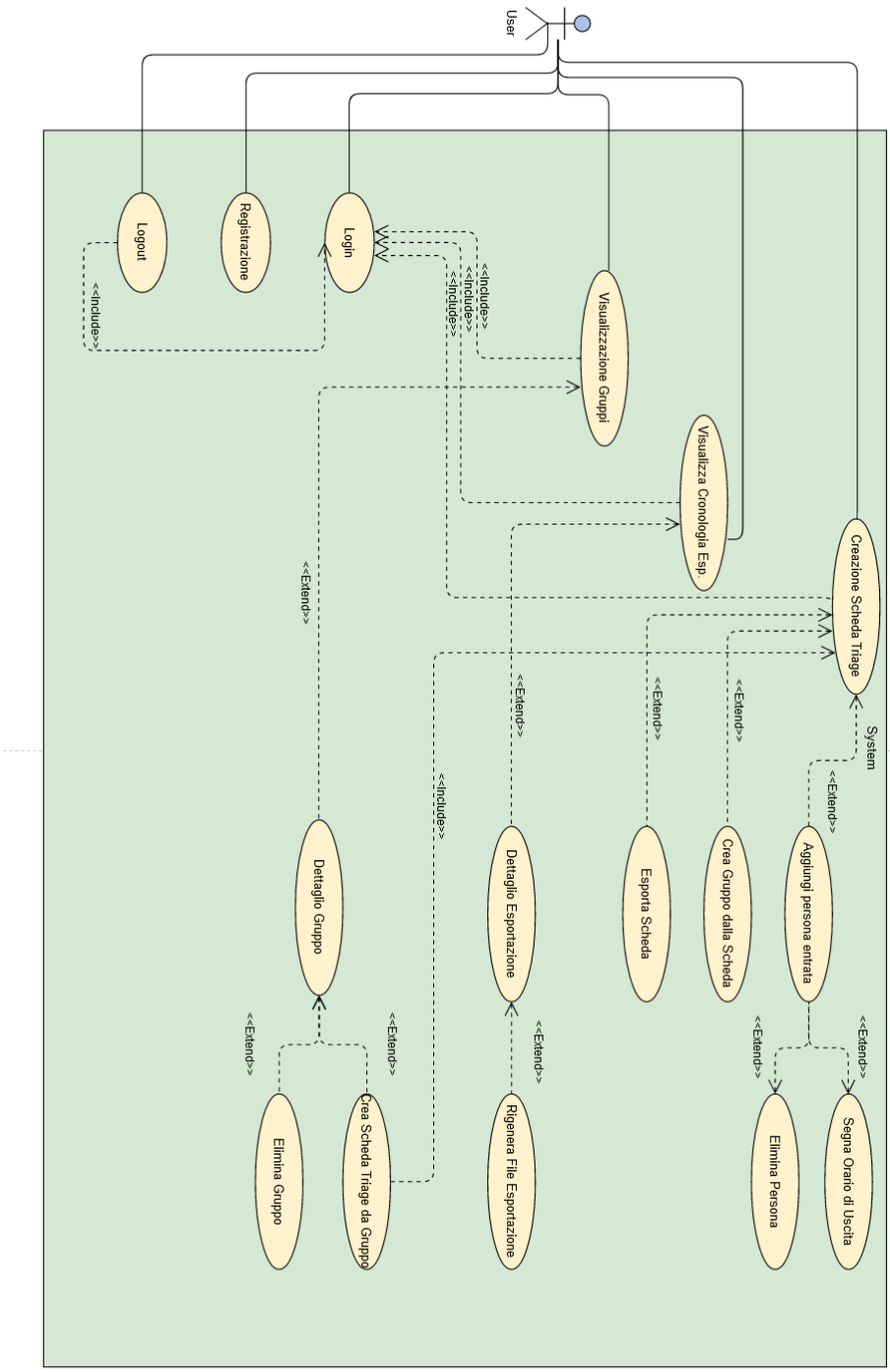


Figura 3.1: Diagramma completo dei casi d'uso

3.1.3 Diagramma delle Attività

I diagrammi delle attività [28] sono diagrammi che modellano il flusso di lavoro delle funzionalità del sistema.

Questo tipo di diagramma UML viene utilizzato per descrivere e relazionare ad un livello più dettagliato i requisiti funzionali dei casi d'uso.

Si tratta di un grafo in cui nodi identificano delle *attività/azioni* (rappresentati come rettangoli) e gli archi orientati rappresentano l'*ordine di esecuzione*. La differenza tra nodi attività e nodi azione è che quest'ultimi rappresentano un attività *atomica*, ovvero che non può essere scomposta in sotto-attività più piccole. Nonostante questa distinzione, entrambi vengono rappresentate graficamente con lo stesso simbolo.

Possiamo trovare inoltre dei nodi *decisionali* (rappresentati come rombi) che identificano diramazioni che devono essere seguite solo se si verificano determinate condizioni.

Infine, troviamo i nodi di *inizio* e di *fine processo* che identificano gli estremi del flusso di controllo.

In figura 3.2 possiamo vedere il diagramma completo delle attività di *MyTriageApp*. Notiamo subito come il flusso inizi con un nodo decisionale: se l'utente ha già un account, può procedere con l'attività di *Login*, altrimenti viene rimandato all'attività di registrazione e, solo dopo essersi registrato, può effettuare il login. Se le credenziali inserite per il login non sono corrette, l'utente torna all'inizio del flusso. Altrimenti, procede all'interno del flusso e può decidere, idealmente tramite tre tasti, una tra le seguenti attività:

- ***Scheda Triage***, all'interno della quale è possibile: inserire una nuova persona entrata nell'impianto, eliminarla oppure segnarla come uscita. Da questa attività è possibile inoltre esportare la scheda, scegliendo il percorso all'interno del dispositivo, oppure creare un nuovo gruppo, inserendone nome e una breve descrizione.

- ***Visualizza Gruppi***, mostra la lista dei gruppi creati dall'utente: è possibile premere su un gruppo per vederne i dettagli oppure sceglierlo per creare una nuova scheda triage e pre-inserirvi gli appartenenti al gruppo.

- ***Visualizza Esportazioni***, mostra la cronologia delle esportazioni tramite una lista: è possibile premere su un elemento della lista per vedere i dettagli dell'esportazione oppure per rigenerare il pdf relativo.

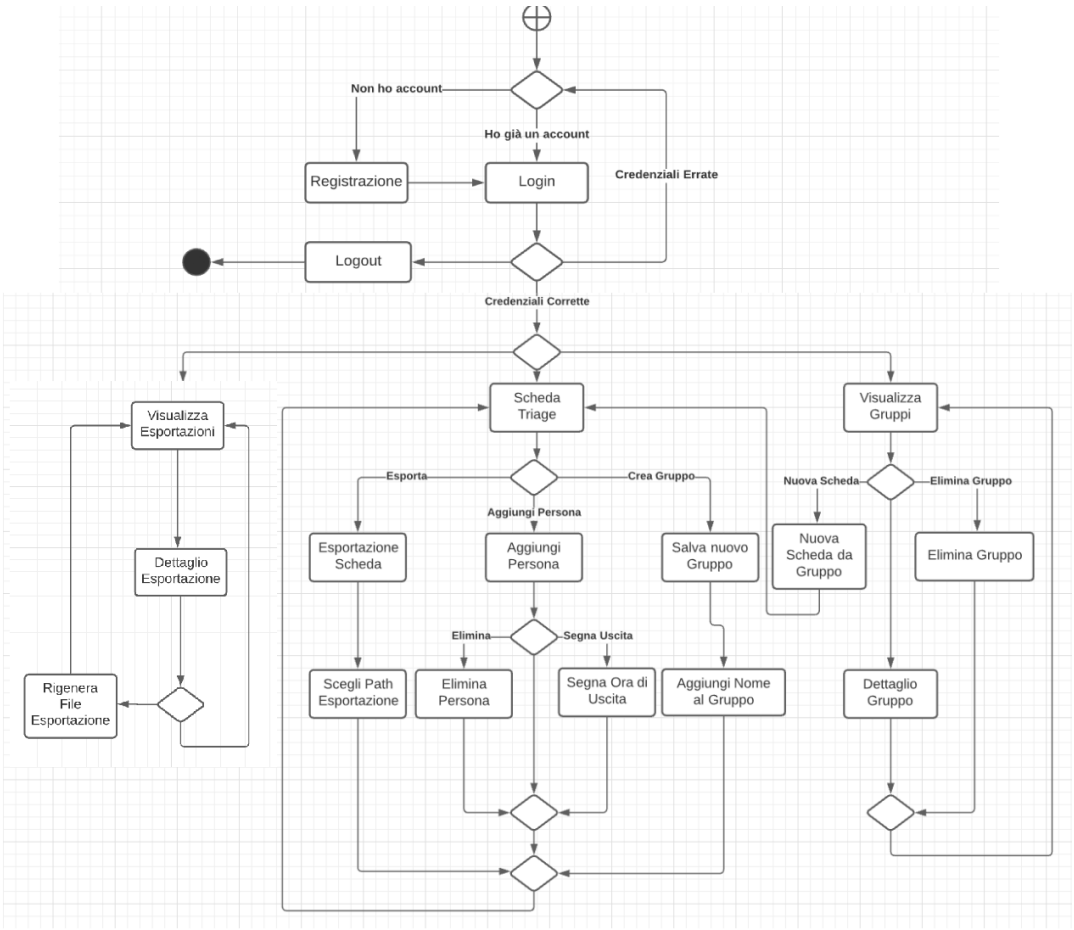


Figura 3.2: Diagramma delle Attività completo applicazione *MyTriageApp*

3.2 Progettazione del Database

Come ampiamente anticipato nella sezione 2.4.2, il progetto *MyTriageApp* utilizza il servizio *Cloud Firestore* di *Firebase* per gestire il database.

Questo servizio segue la logica dei database NoSQL, quindi nella fase di progettazione si andranno prima a definire le entità come semplici oggetti dotati di campi, e successivamente si andrà a tradurre queste entità in una struttura formata da raccolte e documenti.

3.2.1 Costruzione delle Entità

Prima di passare con la vera e propria creazione del database, dobbiamo identificare i protagonisti che ne faranno parte: le **entità**. É bene ricordare che queste entità fanno parte di una struttura con logica NoSQL: i loro campi saranno tradotti in mappe chiave-valori ("*JSON-like*") **senza alcun vincolo** di tipizzazione.

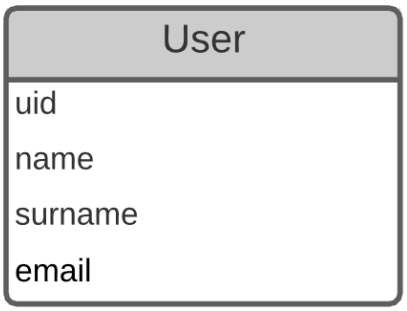


Figura 3.3: Campi entità *User*

L'entità *User* (figura 3.3) rappresenta un utente registrato all'interno dell'applicazione. Dato che l'applicazione utilizza il servizio *Firebase Authentication* per l'autenticazione degli utenti (vedi sezione 2.4.1), le credenziali di registrazione di un utente verranno già salvate nel database di Firebase Auth: salvarle ulteriormente nel database di Firestore sarebbe una ripetizione di dati inutile. A questo proposito, nell'entità *User* salviamo il riferimento all'istanza dell'utente generato da Firebase Auth (tramite campo "`uid`") e le informazioni aggiuntive che il servizio di autenticazione tralascia ("`name`","`surname`","`birth_date`").

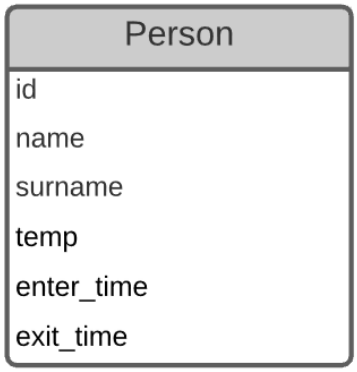


Figura 3.4: Campi entità *Persona*

L'entità *Person* (figura 3.4) identifica ogni nuova persona entrata nell'impianto che viene registrata nella scheda di triage. Oltre che al nome e il cognome, per ogni persona andremo a salvare la temperatura ("`temp`"), l'orario di entrata ("`enter_time`") e di uscita ("`exit_time`") dall'impianto.

Figura 3.5: Campi entità *Schedule*

L'entità *Schedule* (figura 3.5) rappresenta una scheda di triage creata dall'utente. Al suo interno troviamo la data di creazione ("`timestamp`") e la lista di persone entrate ("`person_list`"). L'entità *Group* (figura 3.6) rappresenta

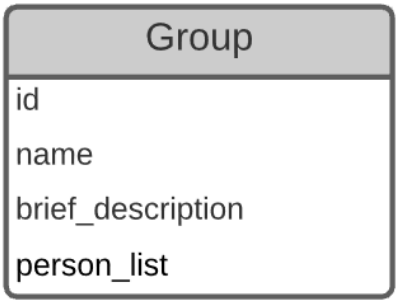


Figura 3.6: Campi entità *Group*

un gruppo di persone che l'utente salva nell'app. Contiene, oltre che alla lista di persone appartenenti al gruppo, il nome ("`name`") e una breve descrizione ("`brief_desc`").

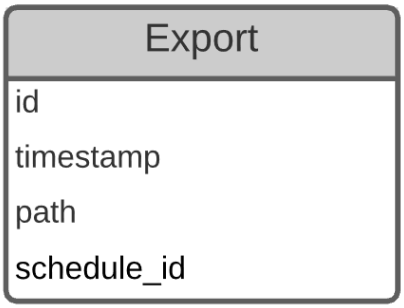


Figura 3.7: Campi entità *Export*

L'entità *Export* (figura 3.7) rappresenta una singola esportazione di una scheda di triage (o *Schedule*) a cui fa rifermento tramite il campo "`schedule`". Registra inoltre l'orario di esportazione ("`timestamp`") e il percorso del file ("`path`") relativo al dispositivo di archiviazione in cui è stato esportato.

3.2.2 Traduzione delle Entità in Raccolte e Documenti

Per la traduzione delle entità non esistono dei passi obbligatori da seguire: solitamente, ogni entità viene tradotta in una raccolta ed ogni istanza di quell'entità sarà salvata all'interno di un documento della raccolta.

Ma non è sempre così. É importante considerare diversi aspetti per evitare ripetizione dei dati e strutture errate. A volte conviene evitare di tradurre un'entità in una sottoraccolta e tradurla invece in un campo complesso del documento (vedi struttura *Exportations* figura 3.9).

La raccolta *radice* del nostro database sarà quella degli ***Users*** (figura 3.8): in questa raccolta viene salvato un documento per ogni utente registrato. Il documento viene nominato con l'*uid* dell'utente.

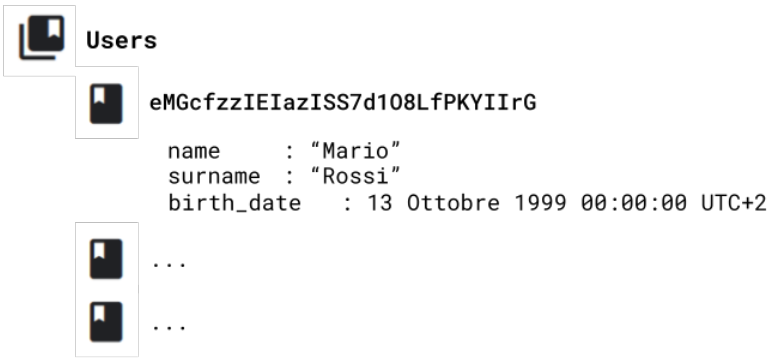


Figura 3.8: Raccolta documenti *Users*

All'interno di ogni documento sono presenti due sottoraccolte: ***Exportations*** e ***Groups***.

La raccolta *Exportations* (figura 3.9) contiene un documento per ogni esportazione effettuata dall'utente, identificato da un id univoco generato da Firestore. Ogni documento avrà un campo complesso chiamato "`schedule`" che rappresenta la scheda triage esportata. Utilizzare un campo complesso per la scheda è più conveniente che creare un'ulteriore sottoraccolta *Schedule*, perché quest'ultima conterrebbe un solo documento.

```
Exportations
    TMeX9YAwITXNiXSiu6iR
        timestamp:   8 Agosto 2021 10:42 AM
        path :       "storage/download/output_8-08-2021.pdf"
        schedule :
            timestamp:   8 Agosto 2021 10:30 AM
            person_list: [
                    {
                        name: "Giovanni",
                        surname: "Bianchi",
                        enter_time: 8 agosto 2021 10:31 AM
                        exit_time: 8 agosto 2021 10:37 AM
                    },
                    {
                        name: "Carlo",
                        surname: "Verdi",
                        enter_time: 8 agosto 2021 10:32 AM
                        exit_time: 8 agosto 2021 10:38 AM
                    },
                    ...
                    ]
    ...
    ...
```

Figura 3.9: Raccolta documenti *Exportations*

La raccolta *Groups* (figura 3.10) contiene un documento per ogni gruppo salvato dall'utente, identificato da un id univoco generato da Firestore. Ogni documento di questa raccolta ha due campi ("`name`", "`brief_description`") e un'ulteriore sottoraccolta chiamata *People*. In questa sottoraccolta si trova un documento per ogni persona che appartiene al gruppo, con i campi per il nome ed il cognome.

Figura 3.10: Raccolta documenti *Groups*

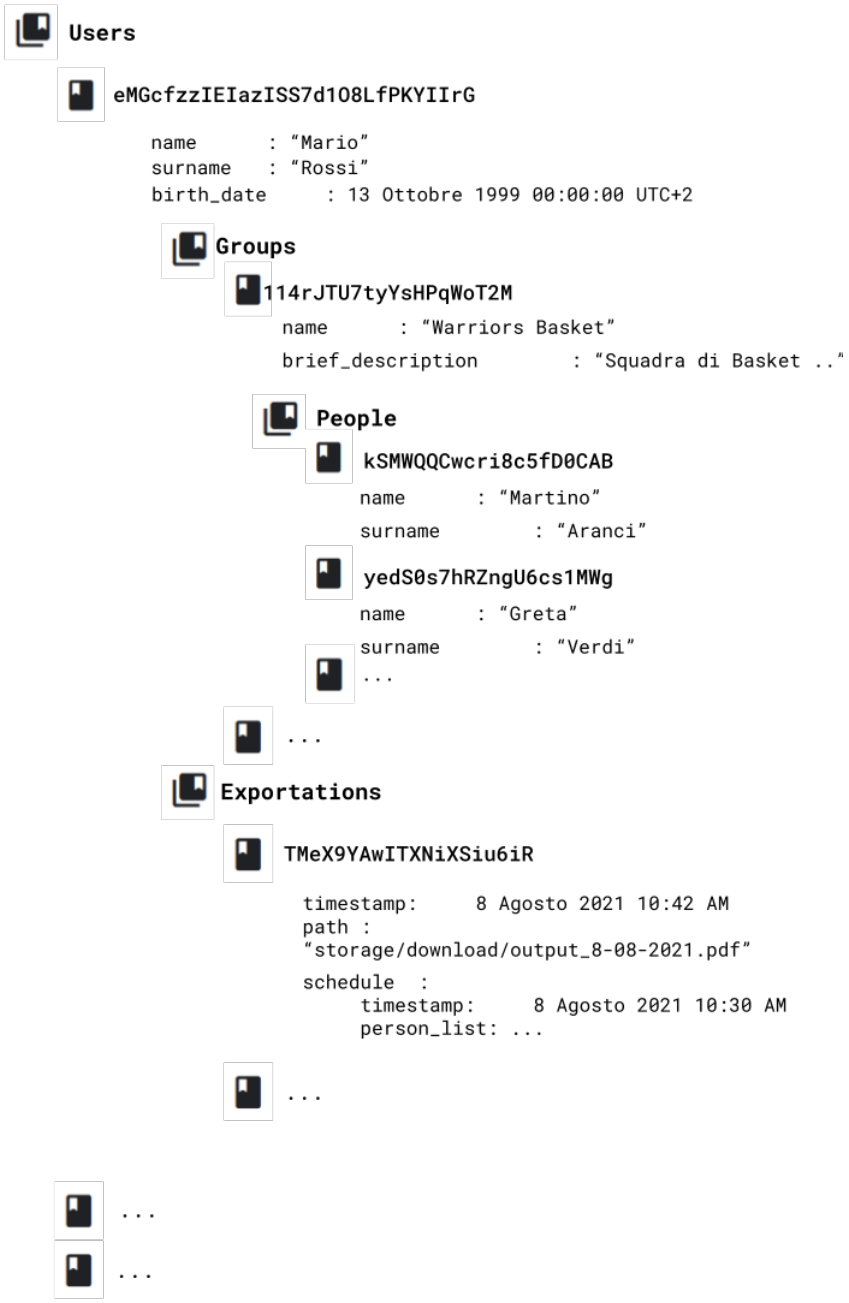


Figura 3.11: Modello struttura database completo

Capitolo 4

Implementazione Xamarin

In questo capitolo entreremo nei dettagli implementativi dell'applicazione sviluppata tramite il framework Xamarin.

Il sistema operativo target è Android, ma data la natura cross-platform della piattaforme di sviluppo non dovrebbe essere complicato compilare e distribuire l'applicazione anche per dispositivi iOS, anche se nel caso di Xamarin sono necessari alcuni accorgimenti che scopriremo più avanti.

Il progetto prende il nome di *MyTriageAppXamarin*, ne verrà spiegata la struttura e commentati i file, mostrando pezzi di codice relativo a classi e funzioni principali.

L'IDE che ho utilizzato è Visual Studio 2019 [20] (versione Community): dato che Xamarin è di casa Microsoft, Visual Studio offre supporto totale e semplifica di molto la scrittura di codice in C#, grazie alle funzionalità di *refactoring*, *debugging* ed *IntelliSense* supportata da intelligenza artificiale.

Infine, per eseguire e testare l'applicazione ho utilizzato l'emulatore di dispositivi integrato in Android Studio. Il dispositivo virtuale su cui ho eseguito le applicazione è un *Google Pixel 2* con la versione 29 delle *Google Services API* (ovvero Android 9.0).

4.1 Struttura del Progetto

Come anticipato nel capitolo sulle tecnologie utilizzate (sezione 2.2.2) utilizziamo il framework *Xamarin.Forms* e creiamo un nuovo progetto cross-platform vuoto.

4.1.1 Progetto .NET Shared

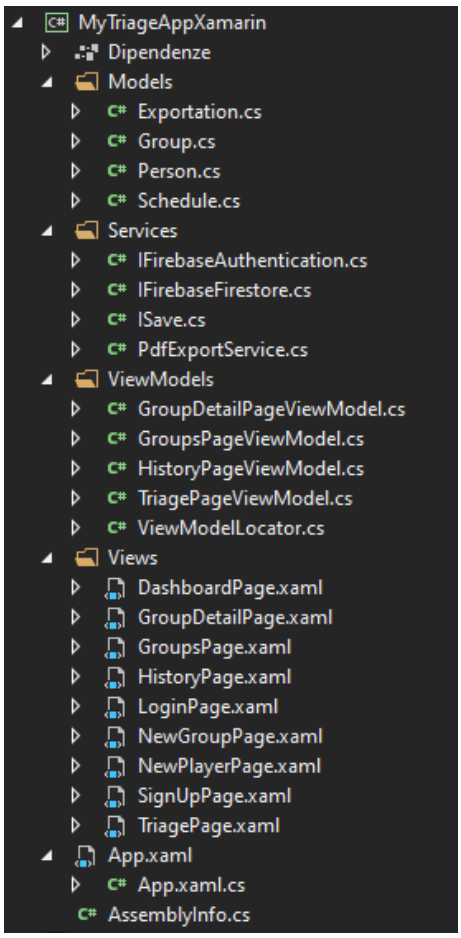


Figura 4.1: Struttura progetto .NET shared

All'interno del nuovo progetto, nel sottoprogetto di libreria .NET (figua 4.1), creiamo quattro cartelle:

- ***Models***, che contiene i file `Person.cs`, `Schedule.cs`, `Group.cs` e `Exportation.cs`.
- ***Services***, che contiene i file `IFirebaseAuthentication.cs`, `IFirebaseFirestore.cs`, `ISave.cs` e `PdfExportService.cs`.
- ***ViewModels***, che contiene i file `TriagePageViewModel.cs`, `HistoryPageViewModel.cs`, `GroupsPageViewModel.cs`, `GroupDetailPageViewModel.cs`, e `ViewModelLocator.cs`.

- ***Views***, contiene i file per le interfacce, ovvero `DashboardPage.xaml`, `LoginPage.xaml`, `SigUpPage.xaml`, `TriagePage.xaml`, `HistoryPage.xaml`, `GroupsPage.xaml`, `GroupDetailPage.xaml`, `NewPlayerPage.xaml` e `NewGroupPage.xaml`.

Inoltre, sempre all'interno del progetto condiviso .NET, è necessario installare tramite il *gestore di pacchetti NuGet* di Visual Studio i package `refactoring.MVVMhelpers` [12] (contiene classi che aiutano a usare l'architettura Model-View-ViewModel) e `Xamarin.Syncfusion.Pdf` [14] (libreria per generare e strutturare file pdf).

4.1.2 Progetto Xamarin.Android

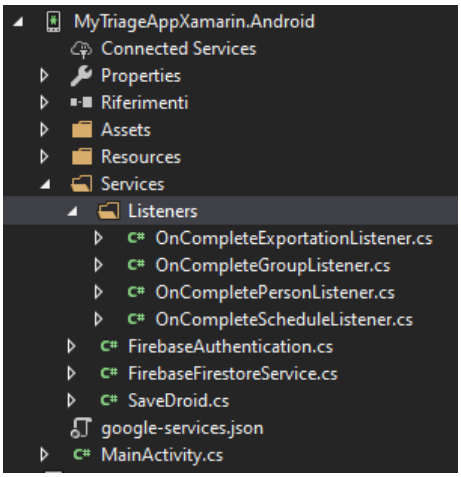


Figura 4.2: Struttura progetto Xamarin.Android

A completare la struttura del progetto Xamarin.Forms troviamo il sottoprogetto Xamarin.Android (figura 4.2) che contiene il codice specifico alla compilazione nativa Android. Al suo interno popoliamo la cartella `../Services/` con i seguenti file:

- `FirebaseAuthentication.cs` e `FirebaseFirestoreService.cs`, classi per l'autenticazione e la comunicazione con il database di Firebase.

- `SaveDroid.cs`, classe per salvare file nella memoria di archivazione interna o esterna di dispositivi Android.

Creiamo anche una cartella `../Services/Listeners/` che contiene delle classi, utilizzate da `FirebaseFirestoreService.cs`, per modellare i dati recuperati dal database all'interno degli oggetti *"Models"* dell'applicazione. Ultima classe di questo sottoprogetto che vale la pena menzionare è la `MainActivity.cs`, ovvero l'entry point d'esecuzione dell'applicazione Android. Entreremo nei dettagli implementativi e di utilizzo di tutti questi file nelle sezioni successive.

4.2 Models

In questa cartella troviamo la classi che definiscono la struttura, la logica di accesso e le regole dei dati dell'applicazione. Partiamo dunque dalla classe che rappresenta una persona all'interno della scheda triage, la classe Person. Oltre che le proprietà per mantenere le informazioni di una persona, sono presenti due metodi, `GoOut()` e `IsOut()` (vedi codice 4.1) . Il primo viene utilizzato quando la persona deve essere segnata come uscita dall'impianto, il secondo per sapere se l'istanza della classe su cui viene chiamato è di una persona già uscita.

```
1  public class Person {
2      public string Name { get; set; }
3      public string Surname { get; set; }
4      public string Temp { get; set; }
5      public DateTime EnterTime { get; set; }
6      public DateTime ExitTime { get; set; }
7      public Color BackgroundColor { get; set; }
8      public Person(string name, string surname, string temp){
9          Name = name;
10         Surname = surname;
11         Temp = temp;
12         BackgroundColor = Color.LightGreen;
13         EnterTime = DateTime.Now;
14     }
15     public void GoOut() {
16         BackgroundColor = Color.Orange; // imposta colore arancione
17         ExitTime = DateTime.Now; // imposta orario di uscita
18     }
19     public bool IsOut(){
20         return (ExitTime > EnterTime) ? true : false;
21     }}
```

Codice 4.1: File Person.cs

Passiamo ora al modello che identifica una scheda triage, la classe Schedule, di cui possiamo vedere le proprietà nell'estratto di codice 4.2.

La proprietà `List` rappresenta la lista di persone inserite nella scheda e non utilizza il tipo generico `List<T>`, bensì il tipo `ObservableRangeCollection<T>`, incluso dal pacchetto *mvvmhelpers*.

La proprietà `CountExited` serve per recuperare il numero di persone uscite all'interno della lista, per farlo utilizza una caratteristica importante delle proprietà in C#: la personalizzazione dei metodi di accesso (*getter* e *setter*) al campo.

Nella classe sono presenti anche due metodi: `IsValid()` che restituisce vero se la lista della scheda non è vuota e quindi pronta per essere esportata, `GetSlug()` che genera e restituisce un nome univoco, che servirà per nominare il file pdf relativo all'esportazione della scheda.

```
1  public class Schedule {
2      public ObservableRangeCollection<Person> List { get; set; }
3      public DateTime Date { get; set; }
4      public int CounterExited
5      { get {
6                  int i = 0;
7                  foreach (var player in List){
8                      if (player.IsOut())
9                          i++;
10                 }
11                 return i;
12             }
13         set { this.CounterUsciti = value; }
14     }
15     public bool IsValid() {
16         return (this.List == null || this.List.Count == 0) ? true : false;
17     }
18     public string GetSlug() {
19         return "output_" + Date.Day.ToString() + "-"
20                 + Date.Month.ToString() + "-"
21                 + Date.Year.ToString() +
22                 "_(" + Date.Minute.ToString() + ").pdf";
23     }
```

Codice 4.2: Proprietà classe Schedule

Arriviamo dunque alle ultime due classi *model*, Exportation e Group, che rappresentano rispettivamente le esportazioni e i gruppi (completa definizione nel codice 4.3).

```
public class Group{
    public string Id { get; set; }
    public string Name { get; set; }
    public string BriefDesc { get; set; }
    public ObservableRangeCollection<Person> People { get; set; }
}
public class Exportation{
    public string Id { get; set; }
    public string Path { get; set; }
    public DateTime Timestamp { get; set; }
}
```

Codice 4.3: Classi Group e Exportation

4.3 Services

4.3.1 Classe FirebaseAuthentication.cs

Il primo step dei diagrammi realizzati durante la fase di progettazione è l'autenticazione dell'utente. Procediamo dunque con i dettagli implementativi della classe FirebaseAuthentication: come accennato nella sezione 2.2.3, in Xamarin.Forms le classi dei servizi vengono prima definite come **interfacce** nel progetto .NET condiviso e poi implementate all'interno dei progetti specifici Xamarin.Android e Xamarin.iOS. Nel caso dell'autenticazione, nella cartella `Services/..` del progetto condiviso è definita la classe interfaccia IFirebaseAuthentication in questo modo:

```
public interface IFirebaseAuthentication {
    Task<string> LoginWithEmailAndPassword(string email, string password);
    Task<string> SignUpWithEmailAndPassword(string email, string name,
                                            string surname, string birthday,
                                            string password);
    bool LogOut();
    bool IsLoggedIn(); }
```

Codice 4.4: Intefaccia IFirebaseAuthentication

L'interfaccia definisce l'intestazione di tre metodi per le operazioni di autenticazione (login, registrazione e logout) e di un metodo per verificare se l'utente è già autenticato. La classe FirebaseAuthentication.cs all'interno del progetto Xamarin.Android eredita questa interfaccia e definisce obbligatoriamente ogni suo metodo. Grazie alla classe statica `FirebaseAuth` (della libreria *Xamarin.Firebase.Auth*) e alla sua proprietà `FirebaseAuth.Instance` possiamo effettuare operazioni d'autenticazione asincrone. Nell'estratto di codice 4.5 è visibile il comportamento del metodo `LoginWithEmailAndPassword`: tramite l'operatore await effettua una chiamata che sospende l'esecuzione del metodo asincrono. Se non vengono rilevate eccezioni (anch'esse fornite dalla libreria precitata), il metodo restituisce lo *uid* dell'utente appena autenticato.

```
public async Task<string> LoginWithEmailAndPassword(string email,
                                                    string password) {
    try {
        var user = await FirebaseAuth.Instance // using Xamarin.Firebase.Auth
                        .SignInWithEmailAndPasswordAsync(email,password);
        return user.User.Uid; // restituisco user uid
    }
    catch (FirebaseAuthInvalidUserException e) {
        // utente non valido, gestisco errore
    }
    catch (FirebaseAuthInvalidCredentialsException e) {
        // credenziali errate, gestisco errore
    }
}
```

Codice 4.5: Metodo di Login asincrono classe FirebaseAuthentication.cs

I metodi `SignUpWithEmailAndPassword` e `Logout` sono implementati in modo analogo a quello di login, differenziano solamente nella chiamata tramite operatore await dei metodi della proprietà `FirebaseAuth.Instance` (precisamente utilizzano `CreateUserWithEmailAndPasswordAsync` e `SignOut`).

4.3.2 Classe FirebaseFirestoreService.cs

Analogamente alla classe per l'autenticazione, la classe per l'interazione con il database **Cloud Firestore** eredita l'interfaccia IFirebaseFirestoreService.cs definita nel progetto .NET condiviso. Come si vede dall'estratto di codice 4.6, nell'interfaccia raggruppiamo i metodi in due tipi: i *GET Method*, ovvero i metodi

per recuperare informazioni dal database, e gli *ADD Method*, ovvero i metodi per aggiungere o aggiornare nuove informazioni al database.

```
public interface IFirebaseFirestore{
    //GET METHOD
    Task<List<Group>> GetGroups();
    Task<List<Person>> GetPeopleFromGroup(string teamId);
    Task<List<Exportation>> GetExportationList();
    Task<Schedule> GetScheduleFromExportation(string exportId);
    //ADD METHOD
    Task AddGroup(string groupId, string name, string briefDesc);
    Task AddPersonToGroup(string groupId, string name, string surname);
    Task AddEsportazione(string path, string timestamp, Schedule schedule);
}
```

Codice 4.6: Definizione interfaccia IFirebaseFirestore.cs

Prima di passare ai dettagli implementativi dei metodi della classe FirebaseFirestoreService.cs, definiamo al suo interno delle proprietà molto importanti per poter interagire con il database.

```
class FirebaseFirestoreService : IFirebaseFirestore
{
  private static string user_collection_name = "users";
  private static string groups_collection_name = "groups";
  private static string exports_collection_name = "exportations";
  private static DocumentReference userDocument =
              FirebaseFirestore
              .Instance
              .Collection(user_collection_name)
              .Document(FirebaseAuth.Instance.CurrentUser.Uid)
    ...
```

Codice 4.7: Proprietà della classe FirebaseFirestoreService

Nella definizione mostrata nel codice 4.7, i primi 3 campi contengono i nomi delle raccolte del database, mentre l'ultimo campo è il riferimento al documento (all'interno della raccolta *users*) relativo all'utente attualmente loggato. In questo modo, è possibile interrogare il database partendo già dal documento dell'utente che effettua la richiesta ed esplorare le relative sottoraccolte.

Dato che i metodi di questa classe si differenziano solamente per il tipo di informazione recuperata o aggiunta al database, entreremo nei dettagli implentativi di un solo metodo GET e di un solo metodo ADD.

Mettiamo sotto la lente di ingrandimento il metodo `AddGroup` (vedi estratto di codice 4.8), che crea un nuovo gruppo nella raccolta *groups* di un utente. Questo metodo crea solamente il gruppo, senza popolarlo. Per aggiungere le persone al gruppo creato si utilizza il metodo `AddPersonToGroup`.

Prima di poter aggiungere delle informazioni al database, è necessario strutturarle all'interno di un dizionario che prende come chiave degli elementi di tipo stringa e come valore degli oggetti. Successivamente si crea un Task che ha come compito quello di aggiungere, all'interno della sottoraccolta "groups" del documento dell'utente loggato, un nuovo documento che ha come nome il parametro `groupId` e come campi le coppie chiavi-valori del dizionario precedentemente creato. Il metodo infine restituisce la Task creata, permettendo a chi vuole chiamare questa funzione di farlo anche in modo **asincrono**.

```
1  public Task AddGroup(string groupId, string name, string briefDesc){
2      var dict = new Dictionary<string, object> {
3          {"name", name },
4          {"brief_desc", briefDesc }
5      };
6      var tcs = Task.Run(() => (
7          userCollection.Collection(groups_collection_name)
8          .Document(groupId).Set(new HashMap(dict))
9          ));
10     return tcs;
11 }
```

Codice 4.8: Metodo `AddGroup` della classe FirebaseFirestoreService

L'implementazione dei metodi GET è leggermente diversa. Prendiamo come esempio il metodo `GetPeopleFromGroup(groupId)` (vedi codice 4.9) che recupera la lista di persone appartenenti a un gruppo usando come parametro l'id di quest'ultimo. Interrogare il database è semplice: in modo analogo a quello visto nel metodo `AddGroup`, si accede (partendo dal documento dell'utente) alla sottoraccolta *groups*. In questa sottoraccolta si trova il documento del gruppo con id uguale a `groupId` e si recupera, tramite il metodo `Get()`, uno `snapshot` della sottoraccolta *people* di questo gruppo. Le informazioni in firestore vengono salvate come coppia di chiavi-valori, dunque lo snapshot recuperato deve essere in qualche modo letto e tradotto nei *models* dell'applicazione. A tal proposito introduciamo le classi OnCompleteListener: sono oggetti collegati ad una Task che, al momento della sua terminazione, eseguono un metodo chiamato

`OnComplete(Task task)`. In questo metodo, i *listeners* recuperano il risultato della task (in questo caso uno snapshot), ne leggono i dati e li inseriscono all'interno di istanze delle classi *models* dell'applicazione. Infine, restituiscono il risultato attraverso l'oggetto `tcs` (istanza della classe `TaskCompletionSource`) che gli è stato passato nel costruttore. Nell'estratto di codice 4.10 troviamo la definizione della metodo OnComplete della classe OnCompletePersonListener, utilizzata dal metodo GetPeopleFromGroup.

```
public Task<List<Person>> GetPeopleFromGroup(string teamId) {
    var tcs = new TaskCompletionSource<List<Person>>();
    userCollection.Collection(groups_collection_name).Document(teamId)
        .Collection("people").Get()
        .AddOnCompleteListener(new OnCompletePersonListener(tcs))
    return tcs.Task;
}
```

Codice 4.9: Metodo GetPeopleFromGroup della classe FirebaseFirestoreService.cs

```
public void OnComplete(Task task) {
    if (task.IsSuccessful) {
        if (task.Result is QuerySnapshot snapshot) {
            var list = new List<Person>();
            if (!snapshot.IsEmpty) {
                foreach (DocumentSnapshot item in snapshot.Documents) {
                    string name = item.Get("name")
                    string surname = item.Get("surname")
                    string temp = item.Get("temperature")
                    list.Add(new Person(name, surname, temp));
                }
                tcs.TrySetResult(list);
                return;
            }
        }
    }
    tcs.TrySetResult(null);// qualcosa e' andato storto
}
```

Codice 4.10: Metodo OnComplete Classe OnCompletePersonListener.cs

4.3.3 Esportazione in PDF

L'esportazione di una scheda triage avviene in due fasi: la prima è la **generazione** del pdf in un oggetto chiamato `MemoryStream`, la seconda è il **salvataggio** di questa stream come file pdf all'interno della memoria del dispositivo.

Per la generazione del pdf utilizziamo la classe statica PdfExportService.cs: questa classe sfrutta il pacchetto `Xamarin.Syncfusion.Pdf` per trasformare la scheda triage in una tabella del pdf. Per farlo espone un metodo statico chiamato `ExportSchedule` che prende come argomento un oggetto di tipo Schedule e restituisce un oggetto di tipo MemoryStream. Come notiamo dal codice 4.11, la tabella viene preparata tramite il metodo SetUpTable definito nella stessa classe. Successivamente si crea, per ogni persona all'interno della scheda, una riga della tabella. Infine, le righe appena create vengono inserite nella sorgente dati della tabella e il documento viene salvato nella stream.

```
1  public static MemoryStream ExportSchedule(Schedule schedule){
2      try {
3          string header = "Scheda Triage in data"+ schedule.Date.ToString();
4          PdfLightTable table = this.SetUpTable(header);
5          List<TableObject> objectList = new List<TableObject>();
6          //Scrive una riga in tabella per ogni persona nella lista
7          foreach (Person g in schedule.List){
8              objectList.Add(new TableRow { Nome = g.Name, Cognome = g.Surname,
9                                  Temperatura = g.Temp + " C",
10                                 Entrato = g.EnterTime.ToString(),
11                                 Uscito = g.ExitTime.ToString() });
12         };
13         table.DataSource = objectList;
14         MemoryStream stream = new MemoryStream();
15         document.Save(stream);// salva documento nella stream
16         document.Close(true);
17         return stream;
18     }
19     catch (Exception ex){// qualcosa e' andato storto
20     return null;
21     }
```

Codice 4.11: Metodo ExportSchedule classe PdfExportService.cs

Lo step successivo alla generazione della tabella nella stream è il salvataggio di quest'ultima in un file pdf. A seconda che si tratti di un sistema Android o

iOS l'implementazione cambia, dunque creiamo una classe interfaccia chiamata `ISave`. É un interfaccia molto semplice, definisce solo un metodo, chiamato `SaveAsFile` che prende come argomenti il nome del file, il tipo e la stream relativa. Successivamente, nel sottoprogetto Xamarin.Android, definiamo la classe `SaveDroid` che eredita l'interfaccia e implementa il metodo `SaveAsFile` (codice 4.12).

Quando si sviluppa un'applicazione mobile è importante ricordare che si sta sviluppando un software che, in principio, lavora in modo isolato dal resto del sistema in cui esegue. Di default, l'applicazione non ha accesso alle funzionalità del sistema. Per accedervi è necessario richiedere i **permessi** all'utente e, per farlo, devono essere dichiarati espressamente all'interno del file *AndroidManifest.xml*, nel tag `<user-permission>`. Xamarin offre delle API per richiedere i permessi all'utente, per questo all'interno della classe SaveDroid è definito il metodo ManageWritePermission, che utilizza queste API per controllare che l'applicazione abbia i permessi per scrivere sulle memoria di archiviazione esterna del dispositivo. Dopo aver controllato i permessi, viene creata una cartella *"MyTriageApp"* al cui interno verrà salvato il file. Nella figura 4.3 viene mostrato il file pdf generato da un'esportazione.

```
1  public async Task<string> SaveAsFile(string fileName, string contentType,
2                                                          MemoryStream stream){
3      this.ManageWritePermission(); // Gestisco permessi
4      var root=Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
5      Java.IO.File myDir = new Java.IO.File(root + "/MyTriageApp");
6      myDir.Mkdir();
7      Java.IO.File file = new Java.IO.File(myDir, fileName);
8      if (file.Exists()) file.Delete();
9      try{ //Scrivo file
10         FileOutputStream outs = new FileOutputStream(file);
11         outs.Write(stream.ToArray());
12         outs.Flush();
13         outs.Close();
14         return file.AbsolutePath;
15     }
16     catch (Exception e) {// qualcosa e' andato storto}
```

Codice 4.12: Metodo SaveAsFile della classe SaveDroid.cs

Created with a trial version of Syncfusion Essential PDF

Scheda Triage delle persone entrate e uscite in data 9/16/2021 2:17:04 PM

Nome	Cognome	Temperatura	Entrato	Uscito
Federico	Bianco	36°C	9/16/2021 2:13:32 PM	9/16/2021 2:15:19 PM
Mario	Rossi	35.5°C	9/16/2021 2:13:48 PM	9/16/2021 2:14:57 PM
Teresa	Verdi	35.7°C	9/16/2021 2:14:30 PM	9/16/2021 2:15:13 PM

Figura 4.3: Esempio file pdf generato dall'esportazione

4.4 Interfaccia Utente

Le **view** costruiscono l'interfaccia utente dell'applicazione attraverso file `.xaml`. Inoltre, in Xamarin.Forms, ad ogni nuovo file interfaccia creato viene associato un file `.xaml.cs` che definisce, in linguaggio C#, una **partial class** collegata all'interfaccia. Direttamente in questa classe si possono implementare metodi per la logica di interazione dell'utente, oppure (se si utilizza un'architettura Model-View-ViewModel) si può delegare questo compito ad una classe **ViewModel**. Per testare entrambe le soluzioni, ho deciso di non utilizzare il pattern MVVM nelle pagine relative all'autenticazione, ovvero `LoginPage` e `SignUpPage` (vedi figure 4.4b e 4.4a), e nella pagina del menu principale, `DashBoardPage` (vedi figura 4.5).

4.4.1 Views Indipendenti

Procediamo ad analizzare le interfacce che ho definito **indipendenti**, ovvero che non utilizzano classi viewmodel per gestire la logica di business. Prendiamo come esempio l'implementazione dell'interfaccia di login nel file `LoginPage.xaml`. Nell'estratto di codice 4.13 è riportata la parte della sua definizione che crea i

due campi di inserimento (email e password) e i tasti per autenticarsi oppure passare alla pagina di registrazione.

```
<ContentPage.Content>
    <StackLayout Padding="10,0,10,0"
                 VerticalOptions="Center">
        <Label Text="Log in:"
            HorizontalOptions="Center"
            TextColor="#4682B4"
            FontSize="30" />
        <Entry Placeholder="E-mail"
            Keyboard="Email"
            x:Name="EmailInput"/>
        <Entry Placeholder="Password"
            IsPassword="True"
            x:Name="PasswordInput"/>
        <Button Text="Enter"
            Clicked="LoginClicked"
            Margin="60,40"
            BackgroundColor="#4682B4"
            TextColor="White"/>
        <Button Text="Create a New Account"
            Clicked="SignUpClicked"
            HorizontalOptions="Center"
            Margin="60,40"
            BackgroundColor="Transparent"/>
    </StackLayout>
</ContentPage.Content>
</ContentPage>
```

Codice 4.13: Implementazione file LoginPage.xaml

Notiamo come, nei tag Button, sia presente un attributo chiamato `Clicked`: in questo attributo inseriamo il nome del metodo, definito all'interno della partial class associata all'interfaccia, che vogliamo venga eseguito quando il tasto viene premuto. In questo caso, all'interno della partial class LoginPage.xaml.cs (vedi implementazione in codice 4.14) dovranno essere definiti due metodi chiamati `LoginClicked` e `SignUpClicked`. Ma questi metodi come accedono ai valori inseriti dall'utente? Notiamo che i tag *Entry*, che rappresentano le caselle di input di email e password, hanno un attributo chiamato `x:Name`: tramite questo attributo andiamo a definire i nomi delle proprietà che verranno create nella partial class e che conterranno i valori inseriti dall'utente.

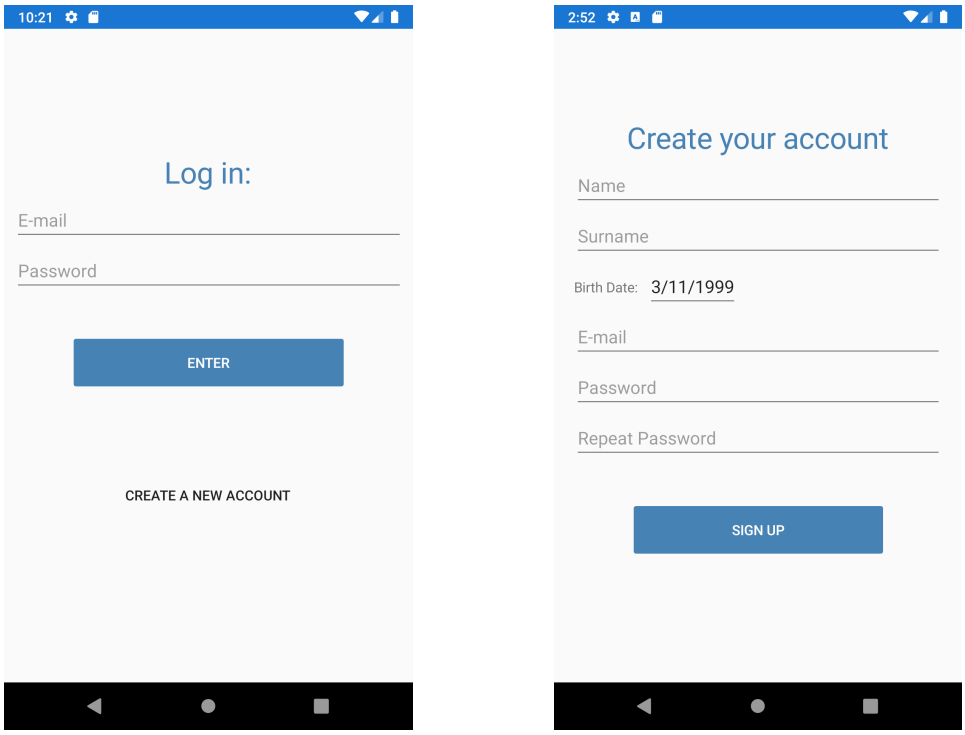
```
1  public partial class LoginPage : ContentPage
2  {
3      private readonly IFirebaseAuthentication auth;
4      public LoginPage() //costruttore
5      {
6          InitializeComponent();
7          auth = DependencyService.Get<IFirebaseAuthentication>();
8      }
9      protected override void OnAppearing()
10     {
11         base.OnAppearing();
12         if (Xamarin.Essentials.Preferences.Get("my_token", "") != "")
13             Application.Current.MainPage =
14                             new NavigationPage(new DashboardPage());
15     }
16     async void LoginClicked(object sender, EventArgs e)
17     {
18         string email = EmailInput.Text; // x:Name attribute reference
19         string pwd = PasswordInput.Text; // x:Name attribute reference
20         string token = await auth.LoginWithEmailAndPassword(email, pwd);
21         if (token != String.Empty)
22         {
23             Xamarin.Essentials.Preferences.Set("my_token", token);
24             Application.Current.MainPage =
25                             new NavigationPage(new DashboardPage());
26         }
27         else
28             await DisplayAlert("Failed", "Email or Password incorrect!", "Ok");
29     }
30     void SignUpClicked(object sender, EventArgs e)
31     {
32         var signOut = auth.SignOut();
33         if (signOut) //manda utente a pagin signup
34             Application.Current.MainPage = new SignUpPage();
35     }
36 }
```

Codice 4.14: Partial class LoginPage.xaml.cs

Notiamo come, a riga 12 e 23 dello stralcio di codice 4.14, si utilizzino dei metodi della classe `Preferences`: questa classe, del pacchetto `Xamarin.Essentials` (solitamente incorporato di default nei progetti Xamarin.Forms), funge da in-

terfaccia per il salvataggio e recupero di ***preferenze di sistema*** attraverso le API native del sistema su cui esegue l'applicazione. In questo caso vengono utilizzate per salvare una stringa `token` e per controllare, all'avvio dell'applicazione (ovvero quando viene chiamato il metodo `OnAppearing()`), se l'utente si sia già autenticato l'ultima volta che ha aperto l'applicazione, ed in quel caso reindirizzarlo direttamente alla dashboard.

La SignUpPage e la DashboardPage sono definite in modo simile a quello visto per la pagina di Login: per la prima, sono presenti altri tag Entry per l'inserimento di nome, cognome e ripetizione password, ed un tag DatePicker per la data di nascita. Mentre per la seconda sono presenti 3 tasti che reindirizzano alle attività principali dell'applicazione (*Visualizzazione Gruppi*, *Cronologia Esportazioni* e *Scheda Triage*).



(a) *Login Page* (b) *Sign Up Page*

Figura 4.4: Screenshot pagine per l'autenticazione - *MyTriageAppXamarin*

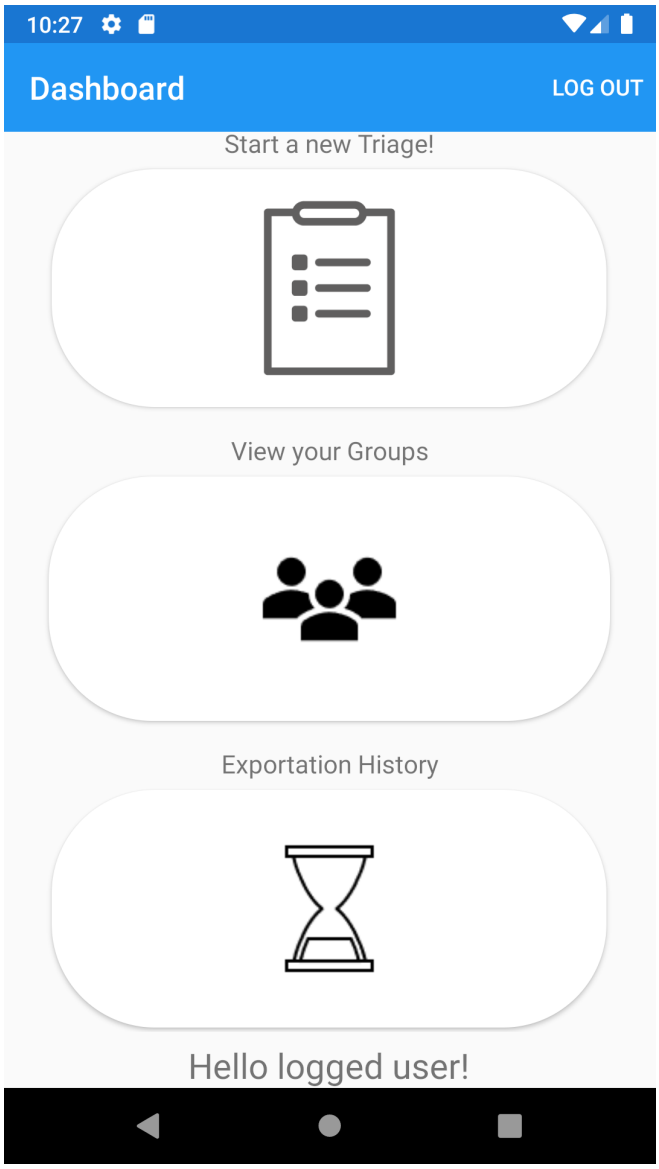


Figura 4.5: Screenshot pagina navigazione - *MyTriageAppXamarin*

4.4.2 Views con ViewModels

Passiamo ora all'implementazione delle restanti view che utilizzano l'architettura MVVM: queste interfacce delegano la gestione della logica d'interazione con l'utente a delle classi chiamate *viemodel*. Per farlo, all'interno della partial class relativa alla view si passa all'attributo `BindingContext` una nuova istanza della classe viemodel. In questo modo, quando si dovranno risolvere i riferimenti definiti all'interno della view, verranno cercati direttamente nella viewmodel passata al contesto.

Vediamone meglio i meccanismi prendendo come esempio la view GroupPage, che rappresenta la pagina di "Visualizzazione Gruppi".

Come si vede nell'estratto di codice 4.15, definiamo un elemento chiamato CollectionView che rappresenta una lista di elementi. Per popolare questa lista si utilizza l'attributo `ItemSource` e lo si associa tramite **binding** alla proprietà che contiene gli elementi che popoleranno la CollectionView . Analogamente, tramite binding si definisce il comando `SelectionChangedCommand`, ovvero l'azione/metodo da eseguire quando un elemento della lista viene premuto.

Inoltre, tramite il tag interno `CollectionView.ItemTemplate` possiamo definire lo stile grafico degli elementi della lista: dentro questo template si può fare riferimento (sempre tramite binding) ai campi dei singoli elementi della lista.

```
1  <CollectionView x:Name="CollectionView"
2      SelectionChangedCommand="{Binding DetailGroupCommand}"
3      ItemsSource="{Binding Groups}"
4      ItemsLayout="HorizontalList">
5      <CollectionView.ItemTemplate>
6          <DataTemplate>
7                  <StackLayout Orientation="Horizontal">
8                      <Label Text="Team Name:" FontAttributes="Bold"
9                              VerticalOptions="Center" />
10                     <Label Text="{Binding Name}" VerticalOptions="Center" />
11                     <Label Text="Description:" FontAttributes="Bold"
12                             VerticalOptions="Center" />
13                     <Label Text="{Binding BriefDesc}" VerticalOptions="Center" />
14                 </StackLayout>
15         <DataTemplate>
16     </CollectionView.ItemTemplate>
17 </CollectionView>
```

Codice 4.15: Implementazione CollectionView nell'interfaccia GroupPage.xaml

Per completare il quadro dell'architettura MVVM, vediamo come si implementa la classe GroupPageViewModel.cs (vedi codice 4.16).

Ogni viewmodel eredita dalla classe BaseViewModel della libreria *MvvmHelpers*. Nel caso della visualizzazione gruppi, dobbiamo recuperare la lista dei gruppi relativa all'utente da Firestore, quindi aggiungiamo un campo di tipo `IFirebaseFirestore` che ci aiuti a svolgere questo compito, fornendo i metodi che abbiamo visto nella sezione 4.3.2.

Inoltre, definiamo le due proprietà a cui facciamo riferimento nel file GroupPage.xaml, ovvero la lista `Groups` e il comando asincrono `DetailGroupCommand`. Questo comando asincrono viene inizializzato nel costruttore con un parametro che rappresenta la funzione che deve essere eseguita: in questo caso il metodo asincrono `DetailGroup`. Questo metodo riceve un oggetto come argomento, ne effettua il cast ad oggetto Group e apre una nuova pagina che ne mostra i dettagli.

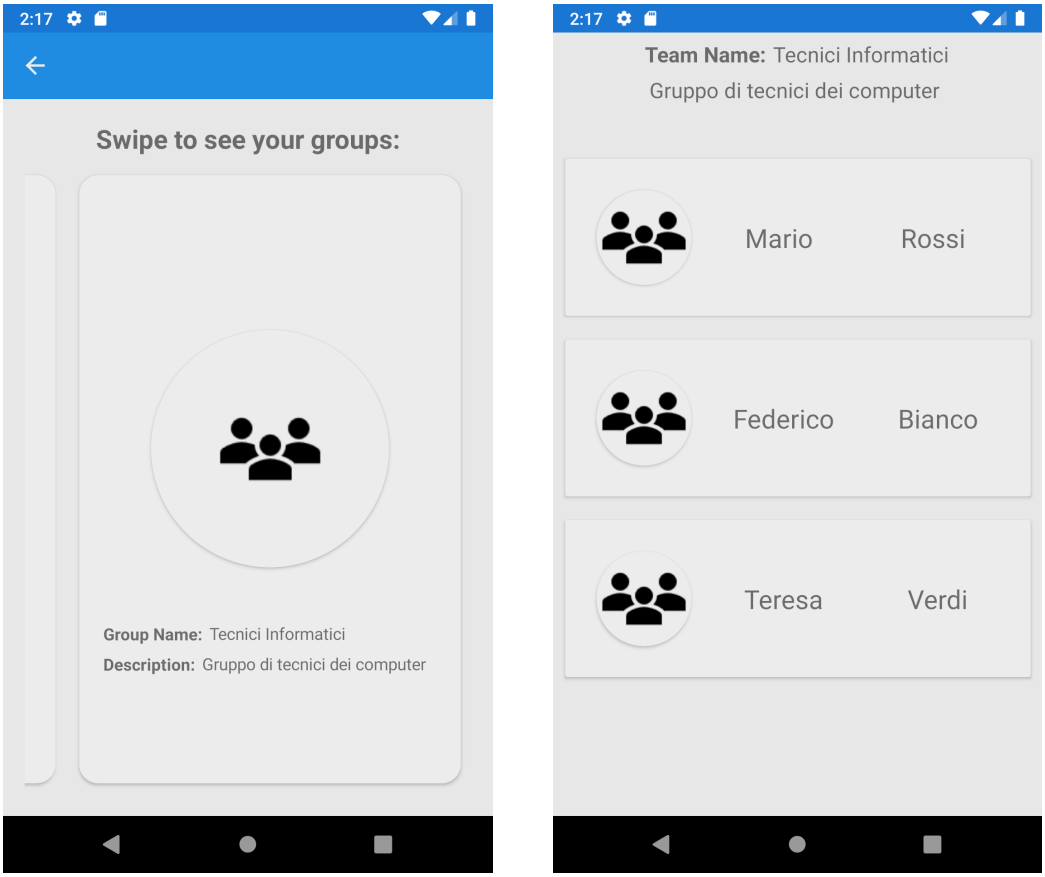
```
class GroupsPageViewModel : BaseViewModel {
    private readonly IFirebaseFirestore firestore;
    public ObservableRangeCollection<Group> Groups { get; set; }
    public AsyncCommand<object> DetailGroupCommand { get; set; }
    public GroupsPageViewModel() {
        Title = "Your Groups";
        firestore = DependencyService.Get<IFirebaseFirestore>();
        Groups = new ObservableRangeCollection<Group>();
        DetailGroupCommand = new AsyncCommand<object>(DetailGroup);
        GetGroupsFromFirestoreAsync();
    }
    private async Task DetailGroup(object arg) {
        var group = arg as Group;
        if (group == null) return;
        await Navigation.PushModalAsync(new GroupDetailPage(group));
    }
    private async void GetGroupsFromFirestoreAsync(){
        var groups = await firestore.GetGroups();
        if (groups != null){
            foreach (Group group in groups)
                group.People = await firestore.GetPeopleFromGroup(group.Id);
            Groups.AddRange(groups);
        }
    }}
```

Codice 4.16: Implementazione classe GroupsPageViewModel.cs

Per quanto riguarda la pagina di Cronologia Esportazioni, utilizza come view l'interfaccia HistoryPage.xaml e come viewmodel la classe HistoryPageViewModel.cs. Entrambe hanno una struttura molto simile a quelle usate per la visualizzazione dei gruppi (vedi figura 4.7). La principale differenza è il comando eseguito quando si preme un elemento della lista: il metodo eseguito si chiama `RegenScheduleExportation` e si occupa di rigenerare il file pdf della relativa esportazione, utilizzando gli strumenti visti nella sezione 4.3.3.

```
private async Task RegenScheduleExportation(object arg){
    var export = arg as Exportation;
    if (export == null) return;
    var schedule = await firestore.GetScheduleFromExportation(export.Id);
    if (schedule == null) return; // scheda non trovata
    bool answer = await App.Current.MainPage.DisplayAlert(
                        "Regenerate pdf file?",
                        "Press yes if you want to regenerate file", "Yes", "No");
    if (answer) {
        var stream = PdfExportService.ExportScheduleToPath(schedule);
        if (stream != null)
            var path = await Xamarin.Forms.DependencyService
                            .Get<ISave>()
                            .SaveAsFile("regen_" + schedule.GetSlug(),
                                        "application / pdf", stream);
    }
}
```

Codice 4.17: RegenScheduleExportation classe HistoryPageViewModel.cs



(a) *Group Page* (b) *Detail Group Page*

Figura 4.6: Pagine di visualizzazione gruppi e di dettaglio di un gruppo - *MyTriageAppXamarin*

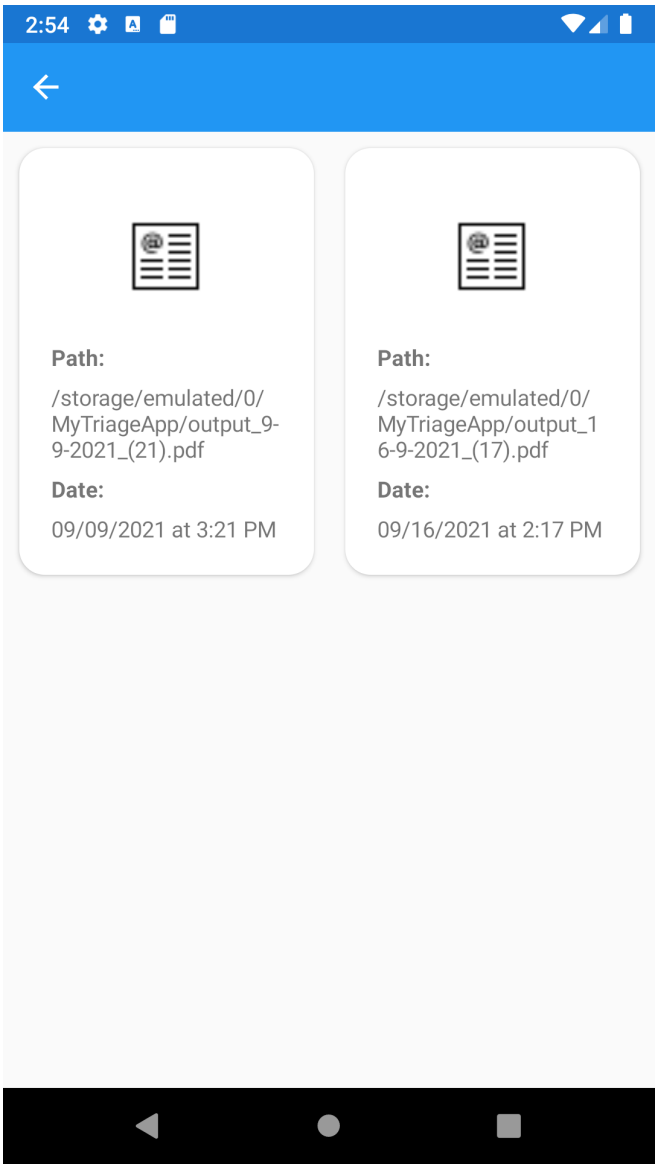


Figura 4.7: Screenshot pagina Cronologia Esportazioni - *MyTriageAppXamarin*

Passiamo ora all'implementazione dell'attività più significativa dell'applicazione: la pagina di creazione di una scheda triage, mostrata nella figura 4.8. L'interfaccia utente di questa pagina viene definita nella view TriagePage.xaml: come le pagine dei gruppi e della cronologia esportazioni, anche questa implementa una lista con il tag CollectionView che rappresenta le persone aggiunte alla scheda. Gli elementi verdi della lista rappresentano persone entrate ma non ancora uscite, mentre quelli arancioni rappresentano persone già uscite.

Infine sono presenti due contatori, che mostrano il totale di persone entrate e il totale di persone uscite, e tre `FloatingActionButton`, ovvero tre tasti sopraelevati che sono rispettivamente collegati ai comandi di aggiunta persona alla scheda, creazione gruppo dalla scheda ed esportazione della scheda.

In questa pagina sfruttiamo una feature molto interessante aggiunta di recente alla classe CollectionView: la ***SwipeView***. Attraverso il tag `CollectionView .SwipeView` possiamo associare dei comandi all'evento di "swipe" sinistro o destro di un elemento. Più precisamente, all'evento SwipeLeft viene collegato il comando di eliminazione di un elemento, mentre all'evento SwipeRight viene associato il comando di uscita di una persona.

L'implementazione dei comandi è pressoché identica a quella vista precedentemente per la pagina di visualizzazione gruppi, fatta eccezione per due comandi: `AddGroupCommand` e `AddPersonCommmand`.

Entriamo nei dettagli del primo di essi: l'idea iniziale era che l'utente, dopo aver premuto il tasto di creazione di un nuovo gruppo, dovesse compilare un popup-form (ovvero una finestra che si sovrappone alla pagina corrente) con il nome e una breve descrizione del gruppo. Purtroppo, Xamarin.Forms non mette a disposizione strumenti per la creazione di popup-form (al contrario di Flutter), dunque l'alternativa è creare un altra pagina per compilare il form.

Da qui nascono le pagine `NewPersonPage.xaml` e `NewGroupPage.xaml`, che contengono semplicemente il form da compilare per l'aggiunta di una persona alla scheda o di un nuovo gruppo alla lista gruppi, come mostrato nelle figure 4.9a e 4.9b.

Utilizzare delle pagine separate però fa scaturire un problema: come faccio a utilizzare, all'interno della `TriagePageViewModel`, delle informazioni che sono state inserite in una pagina separata? Più generalmente il problema si può riassumere in come collegare più views alla stessa viewmodel.

La prima soluzione che ho pensato è quella di passare come `BindingContext`,

nelle partial class NewPersonPage.xaml.cs e NewGroupPage.xaml.cs, la classe TriagePageViewModel.cs, nello stesso modo visto per la pagina di visualizzazione gruppi.

Questo metodo si è rivelato errato e non funzionante, dato che crea e collega una nuova istanza della classe viewmodel invece che collegare quella a cui fa riferimento la pagina di triage. Dunque la pagina di triage, che era già creata, non può ricevere alcun evento e informazione dalla pagina del form, e viceversa.

Dopo alcune ricerche, ho trovato una seconda soluzione, apparentemente non troppo efficiente, ma funzionante: definire una nuova classe chiamata `ViewModel Locator` che, attraverso delle proprietà statiche, permette di accedere alla stessa istanza di una classe viewmodel. A questo punto, sia alla TriagePage (vedi codice 4.19) che alla NewGroupPage (vedi codice 4.20) possiamo passare la stessa istanza di TriageViewModel attraverso la proprietà statica di questa nuova classe.

```
class ViewModelLocator{
    static TriagePageViewModel _viewModel = new TriagePageViewModel();
    public static TriagePageViewModel TriageViewModel
    {
        get { return _viewModel;}
    }
}
```

Codice 4.18: Classe statica ViewModelLocator.cs

```
public partial class TriagePage : ContentPage{
    public TriagePage(){
        InitializeComponent();
        BindingContext = ViewModelLocator.TriageViewModel;
    }
}
```

Codice 4.19: Partial class TriagePage.xaml.cs

```
public partial class NewGroupPage : ContentPage{
    public NewGroupPage(){
        InitializeComponent();
        BindingContext = ViewModelLocator.TriageViewModel
    }
}
```

Codice 4.20: Partial class NewGroupPage.xaml.cs

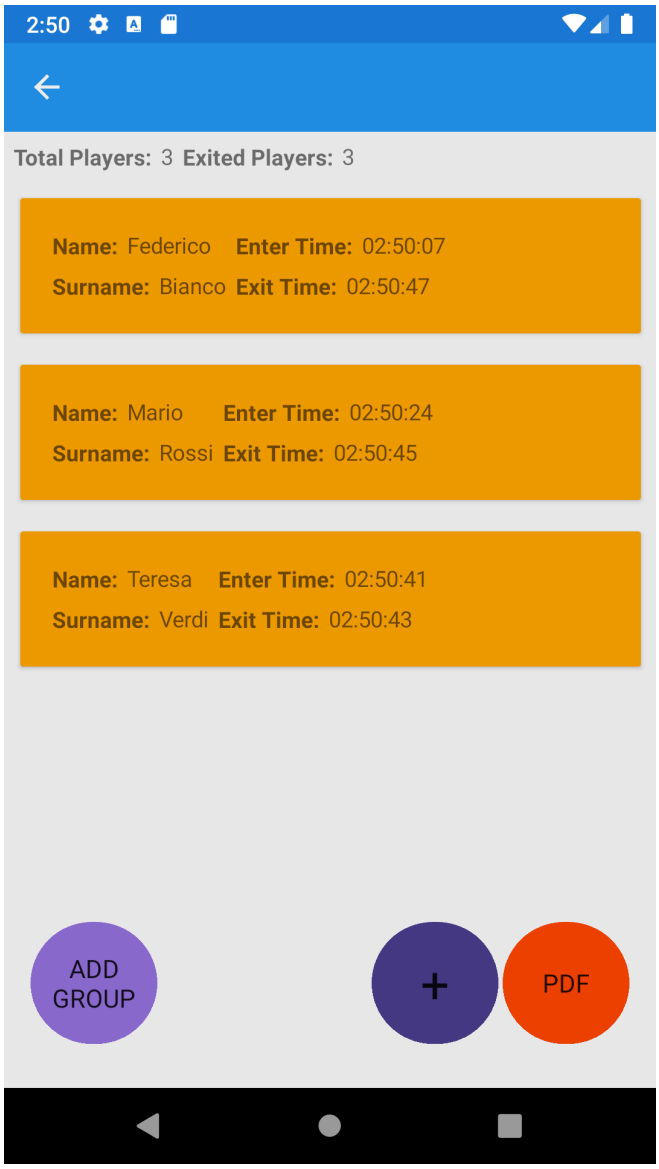
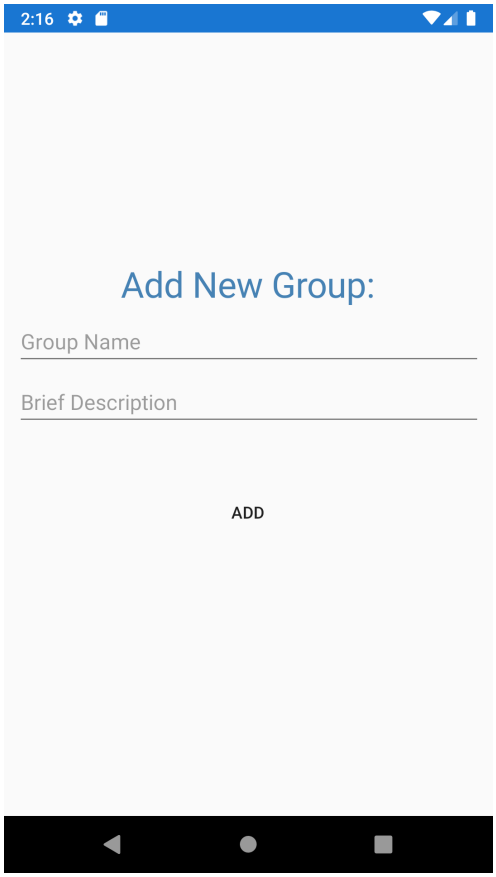
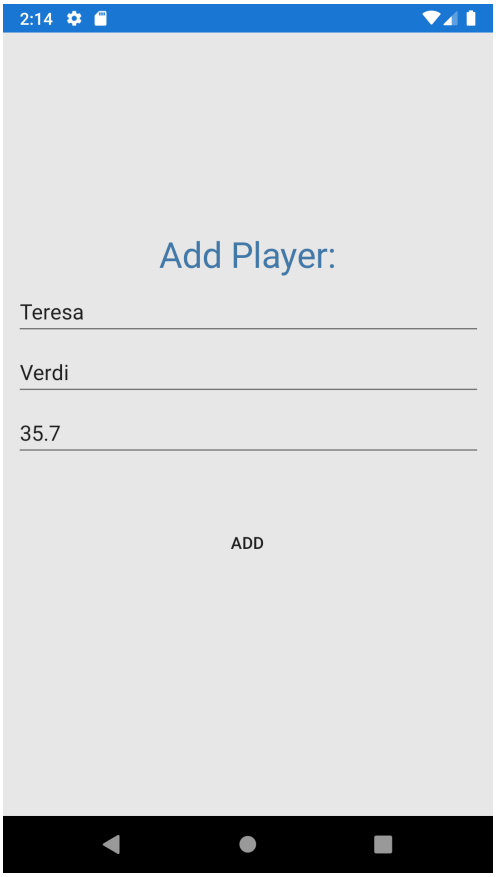


Figura 4.8: Screenshot pagina creazione scheda triage - *MyTriageAppXamarin*



(a) *New Group Page*



(b) *New Person Page*

Figura 4.9: Pagine per form di aggiunta gruppo e di aggiunta persona - *MyTriageAppXamarin*

Capitolo 5

Implementazione Flutter

In questo capitolo sono elencati i dettagli implementativi del progetto *MyTriageAppFlutter*, sviluppato nel framework Flutter.

Per programmare *MyTriageAppFlutter* ho utilizzato come ambiente di sviluppo Visual Studio Code [21]: versione più *"light"* di Visual Studio 2019, adatta anche a sistemi operativi non Windows, offre un ampia gamma di plug-in, chiamati *extensions*, per sviluppare in diversi linguaggi. Le *extensions* che ho utilizzato per sviluppare l'app sono `dart-code.dart` (per supportare il linguaggio Dart) e `dart-code.flutter` (per refactoring, IntelliSense e debugging di applicazioni Flutter).

Anche per questo progetto ho scelto come sistema target Android ed ho eseguito e testato l'applicazione, tramite l'emulatore di Android Studio, sullo stesso dispositivo virtuale che ho usato per il progetto in Xamarin.

5.1 Struttura

Per creare un nuovo progetto in Flutter si esegue da terminale il comando "`flutter create` *`NomeProgetto`*". Con questo comando viene creata la repository principale del progetto con al suo interno la struttura di cartelle vista nella sezione 2.3.3. Procediamo dunque a popolare la cartella `../lib/` con i file `.dart` che compongono l'applicazione (vedi figura 5.1):

- cartella `models/.`, contiene le 4 classi per i modelli dell'applicazione, `person.dart`, `schedule.dart`, `exportation.dart` e `group.dart`.

- cartella `services/.`, contiene le classi `auth.dart`, `firestore.dart` e `pdfexport.dart`.

- cartella `screens/.`, è la cartella più complessa del progetto e contiene le classi che definiscono le pagine dell'applicazione. A sua volta è organizzata nelle seguenti sottocartelle:

  - `/screens/auth/.`, contiene le classi per le pagine di *login* e *sign up*, ovvero i file `authentication.dart`, `registration.dart` e `sign_in.dart`.

  - `/screens/home/.`, contiene le classi per le pagine delle attività dell'applicazione post-login. I file di questa cartella vengono raggruppati a loro volta in sottocartelle che rappresentano le funzionalità dell'applicazione, che sono:

    * `/screens/home/triage`, contiene i file `person_dialog.dart`, `triage_widget.dart` e `triage_page.dart`.
    * `/screens/home/groups`, contiene i file `group_form.dart`, `group_list.dart` e `group_page.dart`
    * `/screens/home/history`, contiene i file `export_list.dart` e `history_page.dart`.
    * `/screens/home/home.dart`, classe che rappresenta la *dashboard* principale dell'applicazione.

  - `/screens/wrapper.dart`, classe che definisce il *root widget* dei widget della cartella `/screen`.

- cartella `widget/.`, contiene i *custom widget* utilizzati nell'applicazione. Al suo interno troviamo i file `group_card.dart`, `person_card.dart`, `home_card.dart`, `myappbar.dart`, `loading.dart` e `person_list.dart`.

- cartella `shared/.`, contiene il file `constants.dart` che definisce variabili costanti che rappresentano aspetti statici dell'applicazione.

Per installare all'interno del progetto i package esterni, elenchiamo le dipendenze nel file `pubspec.yaml`, come mostrato in figura 5.2.

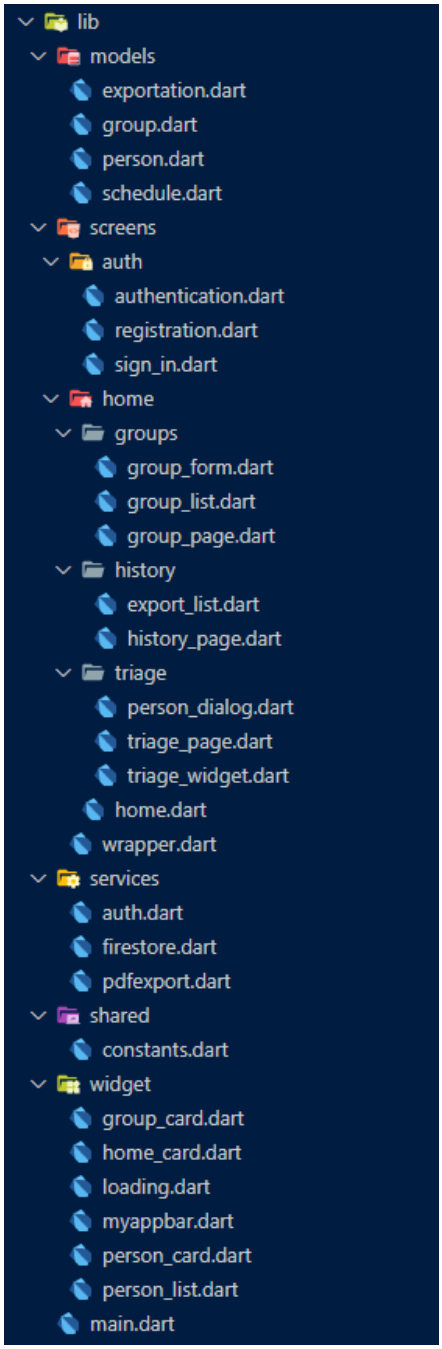


Figura 5.1: Struttura cartella *lib* progetto *MyTriageAppFlutter*

```yaml
dependencies:
  flutter:
    sdk: flutter

  permission_handler: ^5.0.1+1
  provider: ^3.1.0

  # The following adds the Cupertino Icons font to your application.
  # Use with the CupertinoIcons class for iOS style icons.
  cupertino_icons: ^1.0.0

  # ------Firebase packages------
  firebase_auth: ^0.18.1+2
  firebase_core: "^0.5.0+1"
  cloud_firestore: "^0.14.1+3"

  # Spinning load bar package
  flutter_spinkit: "^4.0.0"

  # Package to generate and save pdf file
  pdf: ^1.11.2
  ext_storage: ^1.0.3

  # other google fonts
  google_fonts: ^1.1.1
```

Figura 5.2: Dipendenze a pacchetti esterni nel file *pubspec.yaml*

5.2 Models

Procediamo con le definizioni delle classi nella cartella models. Le proprietà delle classi sono le stesse viste nella sezione 4.2 e la loro definizione è pressoché identica, cambiano solo i costrutti iterativi e i tipi di dato che differenziano il linguaggio C# dal linguaggio Dart.

Partendo dalla classe Person, vediamo dal codice 5.1 che le differenze implementative dalla classe in Xamarin sono solamente due nuovi campi di tipo `get` utilizzati per recuperare in un format specifico gli orari di uscita ed entrata della persona.

```
class Person {
  String name, surname, temperature;
  final DateTime entryTime;
  DateTime exitTime;
  Person({this.name, this.surname, this.temperature, this.entryTime,
                                                        this.exitTime});
  String get getEntryTime {
    return (entryTime == null) ? "--.--" :
                            this.entryTime.hour.toString() + ":" +
                            this.entryTime.minute.toString() + ':' +
                        this.entryTime.second.toString();
  }
  String get getExitTime {
    //...
  }
  void goOut() {
    if (this.exitTime != null) {
      this.exitTime = DateTime.now();
    }
  }
  bool get isOut {
    return this.exitTime.isAfter(this.entryTime);
  }
}
```

Codice 5.1: Definizione classe in person.dart

Per quanto riguarda la classe Schedule, notiamo dal codice 5.2 che, per definire proprietà che rappresentano liste di oggetti, non si utilizza una classe simile a `ObservableRangeCollection<T>` vista in Xamarin, ma si utilizza il costrutto generico `List<T>`.

```
class Schedule {
  final String id;
  List<Person> entriesList;
  final DateTime scheduleTime;
  Schedule({this.id, this.scheduleTime});
  int get counterExited {
    int i = 0;
    for (Person entry in entriesList) {
      if (entry.isOut) i++;
    }
    return i;
  }
  bool isValid() {
    return (this.entriesList != null && this.entriesList.length != 0);
  }
  String get slug {
    return this.scheduleTime.day.toString() +
        "-" +
        this.scheduleTime.month.toString() +
        "-" +
        this.scheduleTime.year.toString() +
        "(" +
        this.scheduleTime.second.toString() +
        ").pdf";
  }
}
```

Codice 5.2: Definizione classe in schedule.dart

Concludendo i modelli, le definizioni delle classi Group ed Exportation sono analoghe a quelle viste nel codice 4.3 del capitolo precedente.

5.3 Services

5.3.1 Classe AuthService

La classe AuthService viene definita all'interno del file auth.dart, nella cartella services. Questa classe serve per effettuare operazioni di autenticazione dell'utente con il servizio Firebase Authentication fornito da Firebase. La classe implementa i tre "classici" metodi per login, registrazione di un nuovo utente e

logout, utilizzando i metodi della proprietà `FirebaseAuth.Instance` del package `firebase_auth.dart`, con un'implementazione molto simile a quella vista in Xamarin. Inoltre, dall'estratto di codice 5.3, notiamo la presenza di un campo di tipo `get` chiamato `user`: questo campo serve per notificare, alle pagine che lo sfrutteranno, i cambi di stato d'autenticazione dell'utente.

```
import 'package:firebase_auth/firebase_auth.dart';
class AuthService {
  final FirebaseAuth _auth = FirebaseAuth.instance;
  Stream<User> get user {
    return _auth.authStateChanges();
  }
  Future signInWithEmailAndPassword(String email, String password) async {
    try {
      UserCredential result = await _auth.signInWithEmailAndPassword(
          email: email, password: password);
      User user = result.user;
      return user;
    } catch (e) {
      print(e.toString());
      return null;
    }
  }
  Future registerWithEmailAndPassword(String email, String password, String
    name, String surname, DateTime birthdate) async {
      // Metodo per registrare un nuovo utente
  }
  Future signOut() async {
      // Log out
  }
}
```

Codice 5.3: Classe AuthService in auth.dart

5.3.2 Classe DatabaseService

La classe DatabaseService definita all'interno del file firestore.dart è utilizzata per recuperare, aggiungere e aggiornare informazioni al database **Cloud Firestore**.

DatabaseService definisce tre tipi di metodi: i metodi *GET* per recuperare informazioni dal database, i metodi *ADD* per aggiungere o aggiornare informazioni sul database ed infine i metodi di *MAP*. Questi ultimi sono funzioni private

che svolgono un compito simile alle classi OnCompleteListener viste in Xamarin: trasformano le informazioni restituite dai metodi GET da una struttura JSON alla struttura delle classi models dell'applicazione.

```
class DatabaseService {
  final String uid;
  DatabaseService({this.uid});
  static final String userCollectionName = "users";
  static final String groupCollectionName = "groups";
  static final String exportCollectionName = "exportations";
  final CollectionReference userCollection =
      FirebaseFirestore.instance.collection(userCollectionName);

  //ADD
  Future addGroup(String id, String groupName, String briefDesc) async{}
  Future addPersonToGroup(String groupId, String name, String surname) async
    {}
  Future addPdfExport(String exportPath) async{}
  //GET
  Stream<List<Group>> get userGroups{}
  Stream<List<Exportation>> get userExportations{}
  Stream<List<Person>> getPeopleFromGroup(String groupName){}
  Stream<Schedule> getScheduleFromExportation(String exportId){}
  //MAP
  List<Exportation> _exportationListFromSnapshot(QuerySnapshot event){}
  List<Person> _peopleFromSnapshot(QuerySnapshot event){}
  List<Group> _groupListFromSnapshot(QuerySnapshot event) {}
  Schedule _scheduleFromSnapshot(DocumentSnapshot event){}
}
```

Codice 5.4: Classe DatabaseService in firestore.dart

Per i metodi *GET*, l'implementazione differenzia nel caso in cui sia necessario un argomento: come si può notare dal codice 5.4, i metodi per recuperare i gruppi e le esportazioni dell'utente sono definiti come dei campi `get`, perché non necessitano alcun parametro fuorché lo user id (*uid*) che viene passato nel costruttore della classe. Mentre i metodi per recuperare le persone di un gruppo (`getPeopleFromGroup`) e la scheda triage di un esportazione (`getScheduleFromExportation`) sono definiti come funzioni, perché necessitano di un ulteriore parametro.

Nonostante la loro diversa definizione, tutti i metodi *GET* restituiscono una `Stream`, che deve essere poi "mappata" tramite i metodi *MAP* defini-

ti nella classe. Nel codice 5.5 viene riportata l'implementazione del metodo getPeopleFromGroup e del relativo metodo di mapping _peopleFromSnapshot.

```
Stream<List<Person>> getPeopleFromGroup(String groupName) {
return userCollection
    .doc(uid)
    .collection(groupCollectionName)
    .doc(groupName)
    .collection('people')
    .snapshots().map(_peopleFromSnapshot);
}
List<Person> _peopleFromSnapshot(QuerySnapshot event) {
    return event.docs.map((doc) {
      return Person(
        name: doc['name'] ?? '',
        surname: doc['surname'] ?? '');
    }).toList();
  }
```

Codice 5.5: Implementazione getPeopleFromGroup e _peopleFromSnapshot

Per i metodi *ADD*, l'implementazione non differisce molto da quella vista in Xamarin, se non nel tipo di oggetto ritornato. Tutti i metodi *ADD* infatti restituiscono un oggetto di tipo `Future`: gli oggetti di questa classe servono per rappresentare un valore (o un errore) che sarà disponibile in futuro. Di seguito, nell'estratto di codice 5.6, è riportata l'implementazione del metodo addGroup, che aggiunge (ma non popola) un nuovo gruppo alla lista di gruppi dell'utente.

```
Future addGroup(String id, String groupName, String briefDesc) async {
    await userCollection.doc(uid)
          .collection(groupCollectionName)
          .doc(id).set({
              'name': groupName,
              'brief_desc': briefDesc,
            });
}
```

Codice 5.6: Implementazione addGroup

Concludendo, possiamo notare come l'implementazione di questo servizio differenzi relativamente poco da quella vista in Xamarin: l'unico cambiamento sostanziale è la presenza dei metodi per il mapping.

5.3.3 Classe PdfExporter

All'interno del file pdfexport.dart viene definita la classe PdfExporter, utilizzata per generare e salvare da una scheda triage un file pdf. Al contrario di quanto visto in Xamarin, sia la creazione della tabella che il successivo salvataggio in file pdf avvengono attraverso lo stesso metodo.

La scheda triage che deve essere esportata e salvata viene passata come parametro al costruttore della classe. Successivamente, alla chiamata del metodo `exportSchedule` vengono chiesti e controllati i permessi di scrittura e lettura sulla memoria esterna del dispositivo, tramite la libreria esterna `permission_handler` [1]. Dopo aver controllato i permessi, per recuperare il percorso esatto della memoria esterna utilizziamo il package `ext_storage` [29]. Dopodiché, tramite la libreria `flutter_pdf` [13] generiamo e salviamo il pdf nella memoria esterna, come mostrato nel codice 5.7. Il risulta dell'esportazione è analogo a quello visto in figura 4.3.

```
import 'package:pdf/widgets.dart' as pw;
import 'package:permission_handler/permission_handler.dart';
import 'package:ext_storage/ext_storage.dart';
class PdfExporter {
  final Schedule schedule;
  final doc = pw.Document();
  PdfExporter(this.schedule);
  Future<String> exportSchedule() async {
    if (await Permission.storage.request().isGranted) {
      final String path = await ExtStorage.getExternalStoragePublicDirectory(
          ExtStorage.DIRECTORY_DOWNLOADS);
      final timestamp = this.schedule.scheduleTime.toString();
      final finalPath = path + '/output_' + this.schedule.slug + '.pdf';
      doc.addPage( pw.MultiPage(   pageFormat: PdfPageFormat.a4,
                                    build: (pw.Context context) {
                                      // Qui compongo la tabella del pdf
                                    }),);
      final file = File(finalPath);
      file.writeAsBytesSync(doc.save());
      return finalPath;
    } else {
      return null;
    }}}
```

Codice 5.7: Implementazione classe PdfExporter

5.4 Screens

La cartella screens contiene tutte i file utili a rappresentare le varie attività dell'applicazione. La struttura di questa cartella segue la struttura delle pagine dell'applicazione: le prime due sottocartelle, *auth* e *home*, sono la prima diramazione del flusso di attività dell'applicazione. La classe Wrapper, definita nel file wrapper.dart, è un widget stateless che si occupa di indirizzare l'utente nella diramazione giusta, a seconda che sia già autenticato oppure no.

5.4.1 Auth

La cartella auth contiene la definizione di tre classi per la creazione delle pagine di login e di registrazione.

Il file authentication.dart implementa lo stateful widget `Authenticate`: questo widget serve per navigare tra la pagina di login e la pagina di registrazione (vedi codice 5.8). Per farlo, nella classe stato del widget viene definita una variabile booleana che può essere modificata solo tramite la funzione toggleView. Questa funzione ribalta il valore della variabile attraverso il metodo SetState. Infine, nell'override del metodo build, tramite operatore ternario viene costruito il widget di login o di registrazione, passando come parametro la funzione di toggle.

```
class Authenticate extends StatefulWidget {
  @override
  _AuthenticateState createState() => _AuthenticateState();
}
class _AuthenticateState extends State<Authenticate> { // classe per lo stato
  bool showSignIn = true;
  void toggleView() { // funzione che cambia pagina
    setState(() => showSignIn = !showSignIn);
  }
  @override
  Widget build(BuildContext context) {
    return Container(
      child: showSignIn ? SignIn(toggleView: toggleView)
                        : Register(toggleView: toggleView),
    );
  }}
```

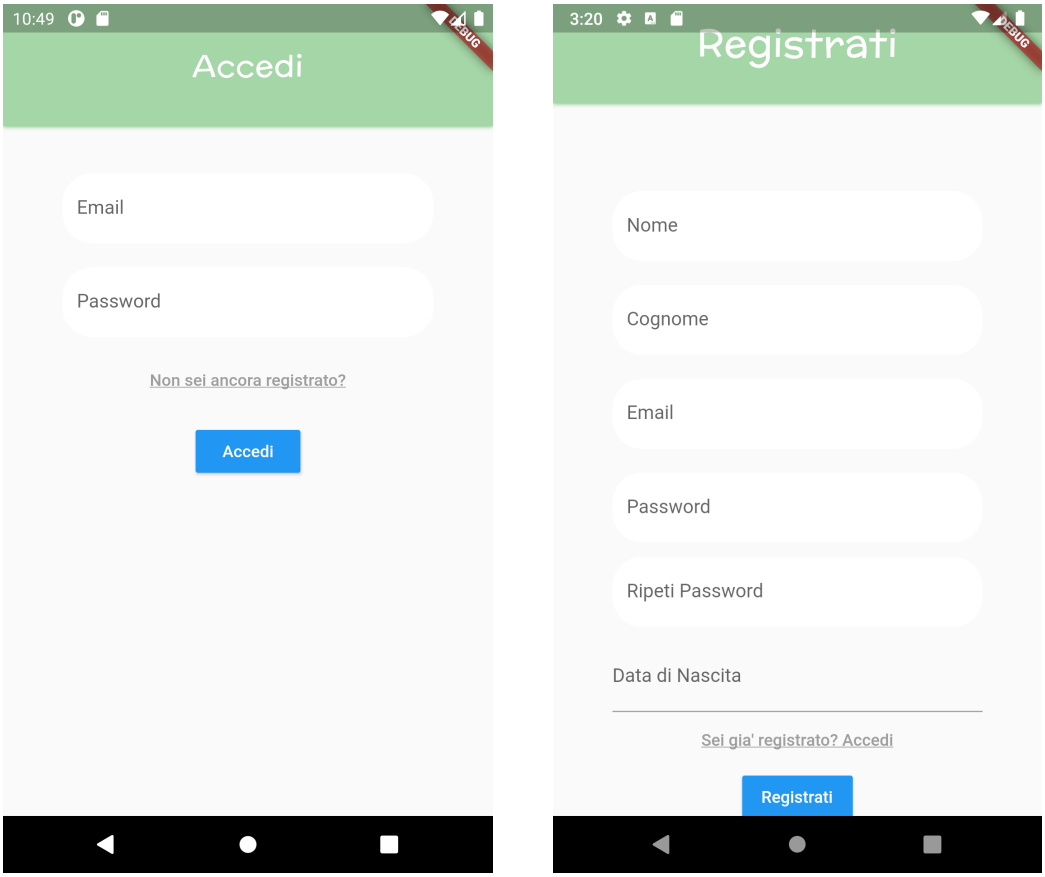
Codice 5.8: Implementazione classe Authenticate

Le pagine SignIn e Register sono definite rispettivamente nei file sign_in.dart e registration.dart. Entrambe hanno un'implementazione simile: presentano all'utente un form per inserire le credenziali e tramite la classe AuthService autenticano o registrano l'utente.

Nel codice 5.9 entriamo nei dettagli di gestione e implementazione del form di login (quello di registrazione è quasi identico): generiamo una chiave per il form tramite la classe `GlobalKey<FormState>`, questa chiave servirà per la validazione finale. A questo punto creiamo il form vero e proprio e inseriamo al suo interno i campi tramite il widget `TextFormField`. Questo widget fornisce gli attributi `validator` e `onChanged` che definiscono i criteri per validare l'input e l'azione da eseguire quando il dato di input cambia. Infine, per inviare il form, è presente un tasto finale che controlla lo stato del form tramite la chiave generata precedentemente e, nel caso in cui sia valido, autentica l'utente con il metodo `signInWithEmailAndPassword`. Come si nota dalle figure 5.3a e 5.3b, la pagina di registrazione differenzia dalla pagina di login per la presenza di campi aggiuntivi e per il diverso metodo della classe `AuthService` chiamato.

```
Form(// Form di login
key: _formKey, //chiave
child: Column(children: <Widget>[
  TextFormField(// campo email
    decoration: textInputDecoration.copyWith(hintText: 'Email'),
    validator: (val) => val.isEmpty ? "Inserisci una mail valida" : null,
    onChanged: (val) {
      setState(() => email = val);
    },),
  //definizione altri campi
  RaisedButton( // Tasto accedi
      onPressed: () async {
        if (_formKey.currentState.validate()) {
          dynamic result = await _authService.signInWithEmailAndPassword(
                  email, password);
          if (result == null) {
            setState(() {
              error = 'Accesso non riuscito!';
            });
        }}}
)]))
```

Codice 5.9: Implementazione form classe SignIn



(a) *Login Page* (b) *Registration Page*

Figura 5.3: Pagine di login e registrazione - *MyTriageAppFlutter*

5.4.2 Triage

L'attività di Creazione Scheda Triage è composta da tre classi: TriagePage, definita nel file triage_page.dart, TriageList definita nel file triage_widget.dart ed infine la classe PersonDialog definita nel file person_dialog.dart.

La classe TriagePage è un widget stateless che rappresenta la pagina dell'attività. Racchiude al suo interno il widget TriageList, che contiene la lista delle persone nella scheda e i tasti per aggiungere una persona, esportare la scheda e creare un nuovo gruppo dalla scheda, come visibile in figura 5.4.

All'interno del widget TriageList viene creata una lista di `PersonCard` tramite lo strumento `ListView.builder()` di Flutter. La classe PersonCard rappresenta una persona della lista ed è definita nel file person_card.dart, nella cartella widget. Come si può vedere dal codice 5.10, al costruttore di questo widget

viene passata un'istanza della classe Person e le due funzioni `exitPerson` e `removePerson`: la prima viene eseguita quando l'utente preme su un elemento e si occupa di segnare l'orario di uscita, mentre la seconda rimuove la persona dalla lista quando un utente vi tiene premuto a lungo.

```
Widget build(BuildContext context) {
  final user = Provider.of<User>(context);
  return Column(
    children: [
      Expanded( // lista persone
        child: ListView.builder(
            itemCount: schedule.entriesList.length,
            itemBuilder: (BuildContext context, int index) {
              return PersonCard(
                person: schedule.entriesList[index],
                index: index,
                onClickHandler: exitPerson,
                onLongClickHandler: removePerson,
              );
            }),
      ),
      Container( // definisco tasti
          child: Row(
            children: [
              FloatingActionButton(// tasto aggiungi persona ...
              FloatingActionButton(// tasto esporta ...
              RaisedButton(// aggiungi gruppo ...
                  })
  ],),),),],);
}
}
```

Codice 5.10: Implementazione widget TriageList

Il tasto di esportazione genera e salva il pdf della scheda triage utilizzando la classe PdfExporter vista nella sezione 5.3.3.

Il tasto per aggiungere una persona utilizza la classe PersonDialog per mostrare un form in sovrimpressione alla pagina (come si vede in figura 5.5a) ed inserire le informazioni della persona che si vuole aggiungere.

Il tasto per aggiungere un nuovo gruppo utilizza la funzione `showModalBottomSheet` (presente in tutte le classi di widget stateless) per mostrare in uscita dal

basso (vedi figura 5.5b) il form `GroupForm`, definito nel file group_form.dart nella cartella group.

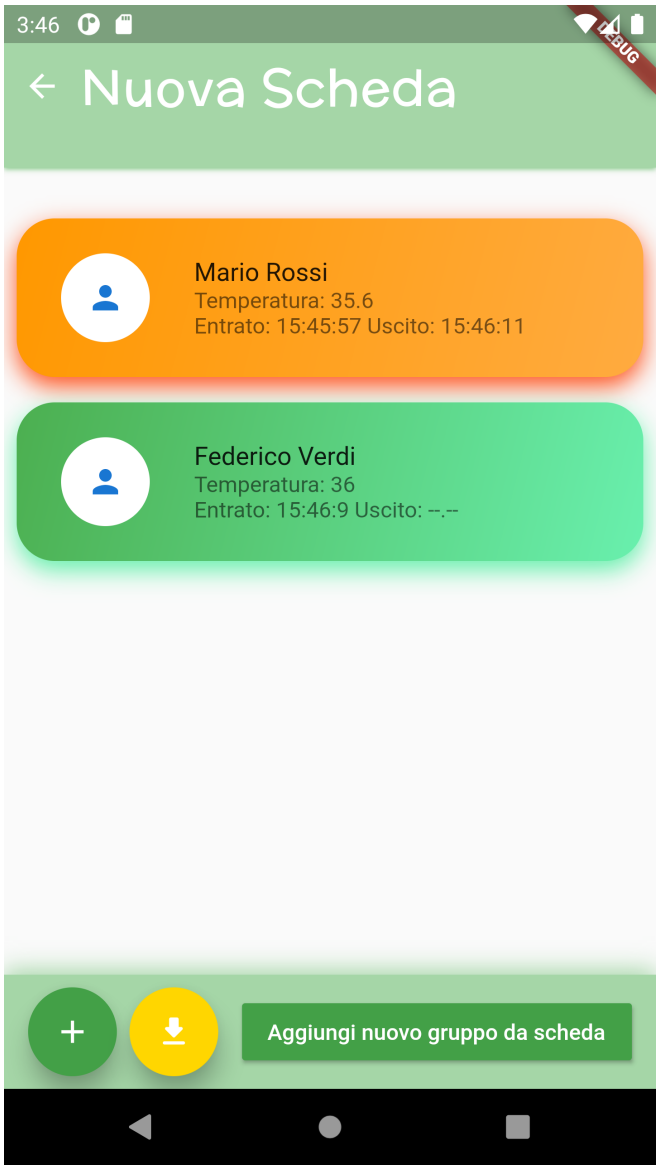
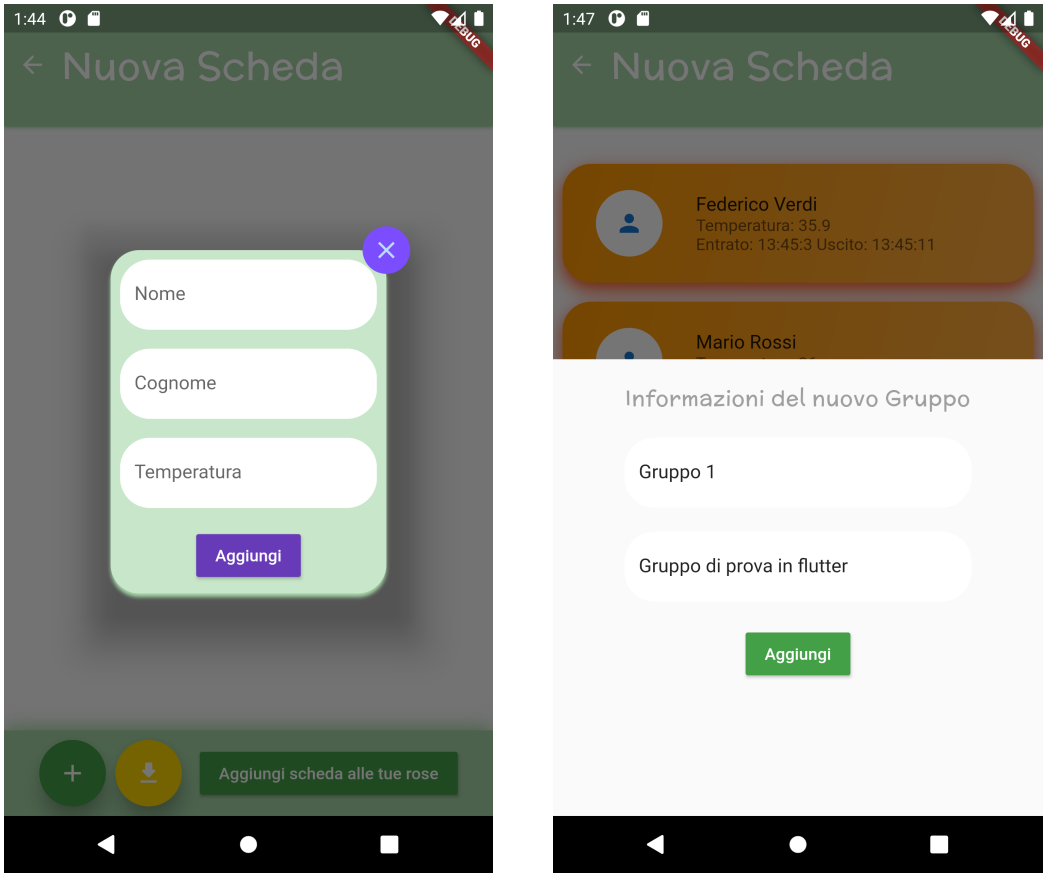


Figura 5.4: Pagina creazione di una scheda di Triage - *MyTriageAppFlutter*



(a) *Person Dialog Form* (b) *Group Form*

Figura 5.5: Screenshot form di aggiunta persona a scheda e di nuovo gruppo - *MyTriageAppFlutter*

5.4.3 Group e History

La attività di Visualizzazione Gruppi e di Cronologia Esportazioni sono implementate in modo simile: entrambe forniscono una pagina con una lista di elementi popolata con delle informazioni recuperate dal database.

Per creare la pagina di visualizzazione gruppi usiamo il widget stateless GrouPage definito nel file group_page.dart. Questa classe recupera le informazioni dell'utente loggato, crea un'istanza di DatabaseService relativa a questo utente e poi, grazie allo strumento `StreamProvider.value()`, costruisce l'oggetto `GroupList` passando nel contesto i gruppi dell'utente .

A questo punto, nella GroupList possiamo recuperare tramite `Provider` i gruppi dell'utente e creare la lista di `GroupCard` (classe definita nel file omonimo della cartella widget) tramite ListView.builder(). L'ultimo passo è recuperare le persone appartenenti ad ogni gruppo e passarle ad ogni elemento della lista: utilizziamo sempre lo StreamProvider e passiamo nel contesto, per ogni elemento, il valore recuperato dalla funzione `getPeopleFromGroup(groupName)`, come si vede a riga 17 dell'estratto di codice 5.11.

```
1  class GroupList extends StatefulWidget {
2    @override
3    _GroupListState createState() => _GroupListState();
4  }
5  class _GroupListState extends State<GroupList> {
6    @override
7    Widget build(BuildContext context) {
8      final userGroups = Provider.of<List<Group>>(context);
9      final user = Provider.of<User>(context);
10     if (userGroups == null) {
11       return Loading();
12     } else {
13       return Expanded(
14         child: ListView.builder(
15             itemCount: userGroups.length,
16             itemBuilder: (context, index) {
17               return StreamProvider<List<Person>>.value(
18                   value: DatabaseService(uid: user.uid)
19                       .getPeopleFromGroup(userGroups[index].name),
20                   child: GroupCard(group: userGroups[index]));}
21     ));}}}
```

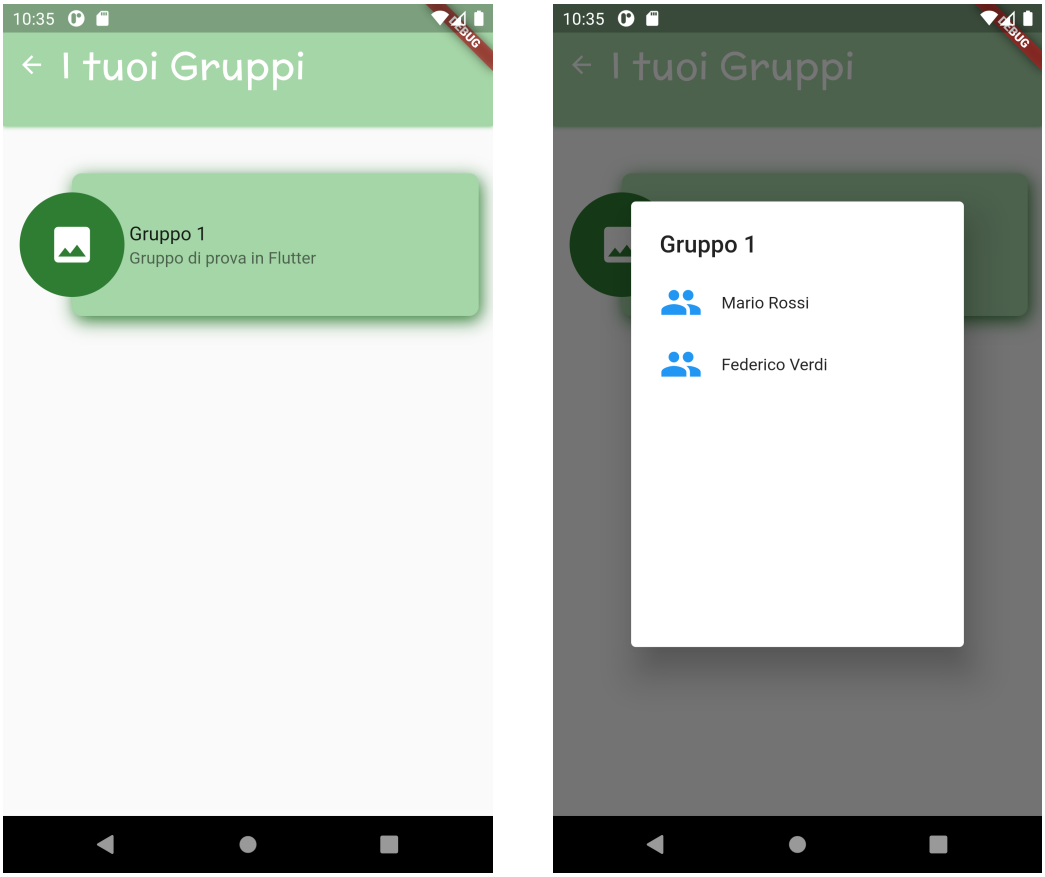
Codice 5.11: Implementazione classe GroupList

La classe GroupCard definisce anche il comportamento di un elemento della lista nel caso in cui venga selezionato: mostra all'utente una `PersonListDialog`, widget definito nel file person_list.dart che genera in sovrimpressione una finestra (stile alert) con la lista di persone appartenenti al gruppo (vedi figura 5.6b).

L'attività di Cronologia Esportazioni, mostrata in figura 5.7, viene definita in modo analogo a quella dei gruppi: la classe `HistoryPage` definisce la pagina, recupera le esportazioni relative all'utente dal database e, tramite StreamProvider, le fornisce alla classe PdfExportList. Quest'ultima, implementata nel file export_list.dart, costruisce la lista e implementa la funzione `regenExportation`, che viene chiamata alla selezione di un elemento della lista e rigenera il pdf dell'esportazione. Come si vede dall'estratto di codice 5.12, la funzione recupera dal database, tramite l'id dell'esportazione, la scheda relativa. Dopodiché, ne riesegue l'esportazione tramite la classe PdfExporter e informa l'utente, attraverso un messaggio *snackbar*, se questa è andata a buon fine.

```
void regenExportation(string exportId) async {
    var schedule = db.getScheduleFromExportation(exportId);
    String result = await PdfExporter(schedule).exportSchedule();
    if (result != null) {
        Scaffold.of(context).showSnackBar(SnackBar(
            content: Text("Rigenerato esportazione pdf in $result")));
    }else{
        Scaffold.of(context).showSnackBar(SnackBar(
            content: Text("Errore! Esportazione fallita")));
    }
}
```

Codice 5.12: Metodo regenExportation classe PdfExportList



(a) *Group Page* (b) *PersonListDialog*

Figura 5.6: Screenshot pagina visualizzazione gruppi e dettaglio di un gruppo - *MyTriageAppFlutter*

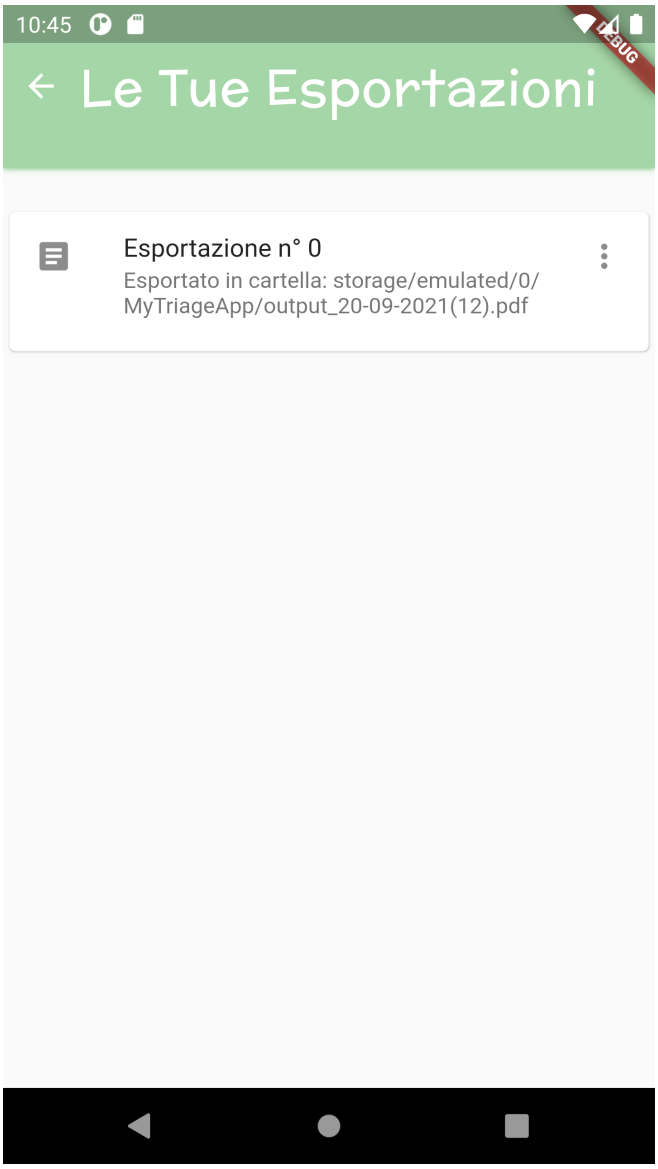


Figura 5.7: Pagina cronologia esportazioni - *MyTriageAppFlutter*

5.5 Shared e Widget

All'interno della cartella widget vengono definite solamente classi di widget stateless, che possono essere utilizzate in modo indipendente nelle pagine dell'applicazione.

Nella cartella shared vengono inseriti file statici, che non definiscono classi o widget, ma solamente variabili di tipo `final` (ovvero non modificabili) utili a configurare l'aspetto grafico dell'app. Nel caso del progetto *MyTriageApp-Flutter*, all'interno della cartella shared è presente un unico file constants.dart che definisce solamente variabili di **decoration** per vari widget dell'applicazione. Queste variabili non fanno altro che impostare colori, ombre ed altri aspetti grafici dei widget in cui vengono utilizzate. Nell'estratto di codice 5.13 possiamo vedere due decoration, utilizzate nella pagina di creazione di una scheda triage, che definiscono l'aspetto di una persona entrata (verde) e di una persona uscita (arancione), come visto in figura 5.4.

```
final personInCardDecoration = BoxDecoration(
    gradient: LinearGradient(
        colors: [Colors.green, Colors.greenAccent],
        begin: Alignment.topLeft,
        end: Alignment.bottomRight),
    borderRadius: BorderRadius.circular(24),
    boxShadow: [
    BoxShadow(color: Colors.greenAccent, blurRadius: 12, offset: Offset(0, 6)
    )
    ]);

final personOutCardDecoration = BoxDecoration(
    gradient: LinearGradient(
        colors: [Colors.orange, Colors.orangeAccent],
        begin: Alignment.topLeft,
        end: Alignment.bottomRight),
    borderRadius: BorderRadius.circular(24),
    boxShadow: [
    BoxShadow(color: Colors.deepOrange, blurRadius: 12, offset: Offset(0, 6))
    ]);
```

Codice 5.13: Esempio di decoration nel file constants.dart

5.5.1 Card

Abbiamo già visto l'utilizzo delle classi GroupCard e PersonCard all'interno degli elementi di una lista. Vediamo ora come questi tipi di widget vengono implementati.

Entriamo nei dettagli implementativi della classe `HomeCard`, che rappresenta i tasti di navigazione della dashboard principale (vedi figura 5.8). La classe definisce diverse proprietà per salvare il titolo, sottotitolo, icona e la route, ovvero l'indirizzo all'interno dell'applicazione a cui navigare quando si preme sulla card. Per funzionare, le routes devono essere definite all'avvio dell'applicazione, nel file main.dart.

```
class HomeCard extends StatelessWidget {
  final String text, subtext, routeName;
  final IconData icon;
  HomeCard({Key key, this.text, this.subtext, this.icon, this.routeName})
      : super(key: key);
  @override
  Widget build(BuildContext context) {
    final heightDevice = MediaQuery.of(context).size.height;
    return Container(
        padding: EdgeInsets.all(12),
        height: heightDevice / 2 * 0.7,
        decoration: homeCardDecoration, // definita in constants.dart
        child: InkWell(
          onTap: () { // quando seleziono navigo a route
            Navigator.pushNamed(context, routeName);
          },
          child: Padding(
            padding: const EdgeInsets.only(top: 20.0),
            child: Column(children: [
              Icon( icon, size: 50, color: Colors.lightGreen),
              SizedBox(height: 20),
              ListTile(Text(subtext, textAlign: TextAlign.center))
            ]),),),),);
  }
```

Codice 5.14: Implementazione widget HomeCard

Figura 5.8: Screenshot dashboard con `HomeCard` - *MyTriageAppFlutter*

5.5.2 AppBar e Loading

Due widget molto importanti in ogni applicazione mobile sono l'*appbar*, ovvero la barra superiore presente in tutte le pagine, e il widget di caricamento, che ho chiamato `Loading`, ovvero il widget che compare quando una pagina è in fase di caricamento dato che non ha ancora ricevuto tutti i dati dal database. In figura 5.9 sono mostrati entrambi i componenti.

Per quanto riguarda la appbar, Flutter mette a disposizione degli sviluppatori classi di default come `AppBar` (in `flutter/material/app_bar.dart`) che peccano però di flessibilità e personalizzazione. É meglio dunque sfruttare l'estrema componibilità di Flutter e dei suoi widget per creare una appbar specifica all'applicazione. Così nasce la classe `MyAppBar`, che non fa altro che costruire "manualmente" la barra superiore con un semplice titolo e un tasto di ritorno (`BackButton`) che esegue la funzione passata nel costruttore come attributo `onPopHandler`.

Per realizzare il widget di caricamento invece, ho utilizzato il pacchetto esterno `flutter_spinkit` [10], che fornisce diverse animazioni da eseguire durante il caricamento della pagina. L'implementazione del file loading.dart è molto semplice: un widget stateless che incorpora un `Container` con al suo interno la classe della libreria esterna `SpinKitFadingCircle`.

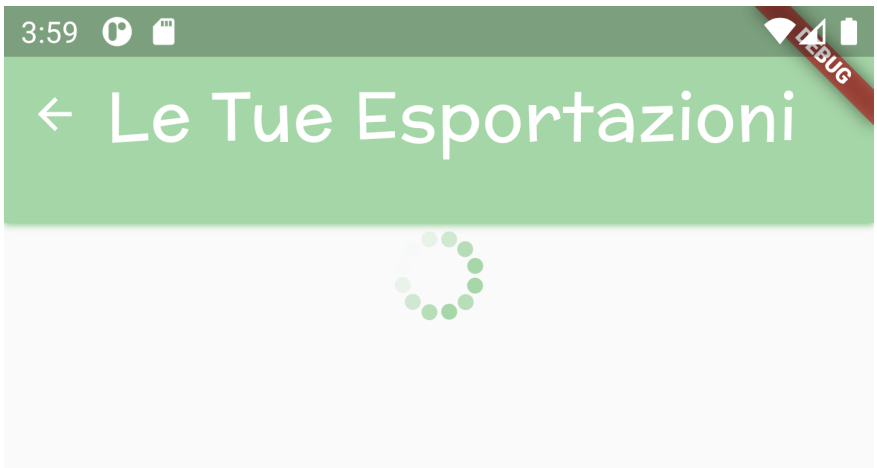


Figura 5.9: Widget di barra superiore e di caricamento in *MyTriageAppFlutter*

Capitolo 6

Confronto

In questo capitolo è riportato il confronto esteso tra le due piattaforme di sviluppo. L'obiettivo è analizzare differenze e somiglianze tra i framework, studiandone la natura e capendo i vantaggi o svantaggi che ne conseguono. Per strutturare meglio il confronto, i criteri di comparazione vengono raggruppati in tre categorie:

- **caratteristiche dei framework**, ovvero gli aspetti intrinsechi che definiscono le piattaforme;

- **developer experience**, comprende i parametri che descrivono l'esperienza avuta dallo sviluppatore nell'utilizzare i framework;

- **performance**, l'insieme di criteri che analizzano l'efficienza del prodotto finale generato dalla piattaforma.

Lo scopo di quest'analisi non è determinare un vincitore tra le piattaforme, ma fornire le informazioni necessarie ad uno sviluppatore per scegliere quale framework si presta maggiormente al suo progetto.

Proprio per questo, a concludere il capitolo è presente una tabella di confronto finale, composta da tutti i parametri considerati durante l'analisi, che riassume brevemente il paragone.

6.1 Caratteristiche dei Framework

6.1.1 Linguaggio di Programmazione

Il vantaggio principale di utilizzare un framework di sviluppo cross-platform è la possibilità di usare un unico linguaggio di programmazione per creare applicazioni destinate a sistemi operativi diversi.

Di conseguenza, il linguaggio utilizzato assume una rilevanza importante nell'ecosistema della piattaforma, ragione per cui è il primo aspetto analizzato nel confronto.

C# nasce nei primi anni duemila all'interno del progetto *.NET* di Microsoft. Supporta tutti i concetti della programmazione a oggetti, la sua sintassi e struttura ricordano molto quelle del *C++* e di *Java*.

Utilizzato principalmente per il *web-developement*, è diventato popolare per le sue importanti caratteristiche come **meta-programming**, **programmazione funzionale** e **portabilità**.

Tutti i programmi scritti in C# vengono eseguiti in .NET, dunque è possibile utilizzare il set di librerie fornito da .NET per eseguire compiti semplici come connettersi a internet, operazioni di I/O, leggere file, ecc.

Anche *Dart* nasce come linguaggio di programmazione per il web, orientato però più al lato client. Rilasciato da Google, aveva come obiettivo quello di sostituire *Javascript* nel mondo della programmazione web. Dotato di una sintassi simile a *Java*, una delle sue caratteristiche principali è quella di poter essere compilato in quattro modalità diverse: **compilazione web**, ovvero direttamente dal browser in stile *Javascript*, **compilazione indipendente**, tramite una macchina virtuale fornita dal Dart SDK, **compilazione Ahead-of-Time**, utilizzata in Flutter per compilare in anticipo le applicazioni, ed infine la **compilazione nativa** tramite il compilatore *dart2native*.

Proprio per questa varietà di compilazioni, Dart adotta una **tipizzazione** leggermente meno **forte** di C#: entrambe definite come ***type safe***, sono controllate sia staticamente che dinamicamente, ma C# ha regole più rigide sul casting e sul tipo di operazioni consentite, mentre Dart usa una tipizzazione che definisce **opzionale**, soprattutto nel dedurre i tipi di dato (***type inference***).

La grande popolarità di C# genera un vantaggio rispetto ai linguaggi concorrenti: la community di .NET è molto grande, dunque il supporto che si può trovare online è enorme e molto utile in diverse fasi di scrittura del codice.

Nonostante la fama di Dart sia in costante crescita, è comunque un linguaggio relativamente giovane e il supporto della community che ha intorno non è ancora comparabile a quello del linguaggio di casa Microsoft.

Concludendo questo primo confronto, non credo ne risulti un chiaro vincitore: sono entrambi linguaggi con sintassi e strutture moderne, probabilmente i migliori nel campo della programmazione cross-platform. Dovendomi sbilanciare, direi che C# ha un leggero vantaggio dal punto di vista della richiesta in ambito lavorativo, dato che è più utilizzato nelle aziende.

Parametro	**Dart**	**C#**
Struttura e Sintassi	Simile *Javascript*	Simile *C++, Java*
Compilazione	Quattro diverse modalità	Nell'ambiente .NET
Community Support	Relativamente giovane	Ampio e consolidato
Paradigmi	*OOP*, funzionale, imperativa e riflessione	*OOP*, funzionale, imperativa ed a eventi
Tipizzazione	***type safe*** ed opzionale	***type safe*** e forte

Tabella 6.1: Riassunto confronto linguaggi di programmazione C# e Dart

6.1.2 Componenti UI e Portabilità

La frase slogan con cui il team di sviluppatori di Google ha presentato Flutter al mondo, *"Is all about widgets"* - ovvero *"è tutta una questione di widget"*, racchiude perfettamente uno dei vantaggi principali di questo framework.

Flutter utilizza una struttura di composizione aggressiva in cui ogni elemento, ogni componente UI dell'applicazione, anche il più piccolo, è un widget. Ogni widget diventa parte di una struttura *top-level* ad albero, utilizzata per renderizzare sul display l'applicazione nei minimi dettagli.

Flutter offre una vasta libreria di widget completamente personalizzabili per coprire ogni aspetto dell'applicazione: animazioni, layout, navigazione ecc. Pro-

prio per questo, spesso tutto ciò che ti serve per sviluppare la tua applicazione è già presente nella libreria di Flutter, rimuovendo o comunque limitando di molto l'utilizzo di librerie di terze parti.

Questo approccio permette a Flutter di garantire consistenza, flessibilità, una buona percentuale di riutilizzo del codice e un miglioramento delle performance.

In Xamarin invece, i componenti dell'interfaccia utente di un'applicazione progettata in Xamarin.Forms sono mappati direttamente ai componenti nativi, permettendo all'utente la scrittura di codice specifico per la piattaforma. Essendo uno dei più vecchi SDK cross-platform, Xamarin fornisce delle API di sviluppo e dei componenti UI ben consolidati all'interno dell'ambiente mobile. Molto spesso però, bisogna ricorrere all'utilizzo di pacchetti o librerie di terze parti per poter sfruttare aspetti più nuovi e aggiornati della UI.

Inoltre, Xamarin supporta diverse piattaforme che Flutter non ancora raggiunge, come applicazioni per dispositivi *wearable* (*Android Wear* e *WatchOS*), che si traduce in una più ampia fornitura di componenti per gli sviluppatori e una maggiore **portabilità**, che Flutter non offre in quanto sviluppa applicazioni solo per Android e iOS.

Al contrario di Flutter, Xamarin non è confinato a un unico pattern architetturale: offre diverse scelte, come il pattern *Model-View-Presenter* (*MVP*) oppure *Model-View-ViewModel* (*MVVM*), molto utili per strutturare e riutilizzare codice in più progetti.

6.2 Developer Experience

6.2.1 Installazione, Documentazione e Supporto

Per poter valutare l'esperienza di uno sviluppatore che utilizza questi framework, bisogna anche considerare lo step iniziale dell'**installazione**, premettendo che, per quanto possibile, una buona installazione deve essere veloce, semplice e leggera.

Per quanto riguarda Xamarin, l'installazione deve avvenire all'interno dell'IDE *Visual Studio*, dunque è richiesta la precedente installazione di quest'ultimo. Purtroppo non esiste documentazione ufficiale riguardante l'installazione di Xamarin separatamente da Visual Studio.

Mentre per quanto riguarda Flutter, il processo di installazione consiste semplicemente nel scaricare il file `flutter.zip`, relativo alla piattaforma su cui deve eseguire, dalla rispettiva repository GitHub. Non sono richiesti altri step aggiuntivi, dato che Flutter è indipendente dall'IDE utilizzato.

Per quanto riguarda la procedura di installazione, Flutter si pone un passo avanti a Xamarin, con un'installazione leggera e intuitiva.

Passiamo ora ad analizzare un altro aspetto molto importante nell'esperienza di utilizzo di uno strumento di sviluppo: la **documentazione**. É il punto di partenza di ogni programmatore che approccia una nuova tecnologia e proprio per questo deve essere chiara e completa.

Nonostante Flutter sia relativamente giovane, la documentazione alle sue spalle non lascia a desiderare: chiara e ben organizzata, fornisce esempi e tutorial che coprono a 360 gradi gli aspetti di sviluppo di un'applicazione. Novità unica e interessante nella documentazione di Flutter sono i *CookBook*, dei veri e propri ricettari che accompagnano passo passo lo sviluppatore nel risolvere problemi comuni che si possono incontrare nella scrittura di app.

La documentazione a supporto di Xamarin è altrettanto completa e chiara, ma la trovo personalmente meno interattiva e peggio strutturata.

Detto questo, Xamarin ha il vantaggio di essere il framework cross-platform più anziano sul mercato e di conseguenza la sua community è la più grande in assoluto. Proprio per questo il supporto che si trova online è ampio, disponibile anche grazie al forum *Microsoft Support* che aiuta a postare problemi, aprire discussioni e inviare feedback direttamente a operatori Microsoft.

Anche se più piccola, la community di Flutter è molto attiva e in continuo aggiornamento: soprattutto negli ultimi anni, Flutter ha avuto un incremento esponenziale di developer che contribuiscono al suo sviluppo, diventando il principale concorrente di Xamarin.

6.2.2 Produttività e Curva di Apprendimento

La **produttività** è un fattore importante per poter progettare applicazioni velocemente, soprattutto perché uno sviluppatore lavora meglio quando è soddisfatto della tecnologia che utilizza.

Flutter in questo campo fornisce uno strumento tanto unico quanto efficace, chiamato ***hot reload***. É una caratteristica del framework che permette di compilare ed eseguire nuove modifiche al codice mentre l'applicazione è in esecu-

zione e vederne gli effetti immediatamente, senza dover performare alcun restart. Per funzionare, l'hot reload inietta i file sorgente aggiornati direttamente nella macchina virtuale di Dart in esecuzione (**Dart VM**). Dopo che la VM viene aggiornata con le nuove classi e le nuove funzioni, Flutter ricostruisce da capo il *widget-tree*, mostrando subito i cambiamenti all'utente.

Anche Xamarin fornisce uno strumento simile, ma non è altrettanto efficace: per i progetti Xamarin.Forms, è possibile sfruttare la feature di ***ricaricamento rapido XAML*** che, come si può intuire, esegue l'hot reload solamente dei file XAML. Dunque i cambiamenti che si possono vedere immediatamente nell'app in esecuzione sono solo quelli apportati alle view, e non alla business logic che vi è dietro.

Il ricaricamento che offre Xamarin è comunque utile e aumenta la produttività di chi lo utilizza, ma è comunque molto limitato se messo a confronto con l'hot reload di Flutter, che copre qualsiasi modifica in qualsiasi parte del progetto.

La **curva di apprendimento** (*learning curve*), in ambito di sviluppo software, indica il rapporto tra il **livello di conoscenza** del software maturato da un suo utilizzatore e il **tempo** impiegato per raggiungerlo. Spesso viene utilizzato come criterio per giudicare e analizzare la qualità di un programma o di uno strumento tecnologico, anche se deve essere sempre considerato come un valore non obiettivo e nemmeno scientifico, ma solamente rappresentativo e informale.

Per considerarsi "esperti" in Xamarin, bisogna sostanzialmente avere conoscenza di C#, della programmazione mobile e dei pattern che utilizza. Dato che C# è abbastanza popolare come linguaggio di programmazione, per sviluppatori con esperienza non dovrebbe essere complicato adattarsi, grazie anche all'aiuto dell'ecosistema che Visual Studio crea intorno. Non sono in grado di affermare le stesse cose per sviluppatori che non hanno familiarità con l'ambiente Microsoft, perché potrebbe volerci più tempo ad acquisire destrezza con il framework.

Dall'altra parte, Flutter è leggermente più semplice da imparare, anche per chi non ha mai avuto a che fare con linguaggi di programmazione come Dart o simili, nonostante a primo impatto possano sembrare un po' inusuali. Questo perché Flutter continua a rilasciare risorse per l'apprendimento: oltre alla documentazione di cui abbiamo già parlato (sezione 6.2.1), sono presenti video, articoli e podcast del developer team originale, che aiutano sia i principianti che i più esperti a rimanere aggiornati e ad utilizzare il framework secondo le sue *best-practice*. É anche vero però che, dopo un primo breve periodo di appren-

dimento iniziale, imparare cose più complesse e strutturate in Flutter non è un processo altrettanto rapido.

In conclusione, dovendo disegnare le due curve di apprendimento, per Xamarin opterei per un andamento abbastanza lineare nel tempo, mentre per Flutter una curva inizialmente ripida, ma che tende ad appiattirsi con l'aumentare delle conoscenze.

6.3 Perfomance

Per confrontare le **performance** dei due framework utilizzati andremo a misurare e analizzare diversi aspetti sia delle applicazioni implementate - *MyTriageAppXamarin* e *MyTriageAppFlutter* - sia di applicazioni create appositamente per il test.

Le funzionalità che andremo a testare sono tre: l'avvio dell'app (*start-up*), la comunicazione con Firebase e l'esportazione in PDF di una scheda. Per ognuna di queste funzionalità verranno misurati i tempi di esecuzione, ricavandone una media su cinque esecuzioni. Inoltre, per avere un quadro più completo, alcune di queste funzionalità verranno replicate e testate anche su un'applicazione nativa progettata in Xamarin.Android, per poterne confrontare le performance con quelle cross-platform.

Le misurazioni non verranno effettuate su un emulatore di dispositivi Android, perché vi è rischio di inconsistenze date dall'esecuzione virtuale dell'hardware. Per questo le funzionalità verranno testate su un dispositivo reale, le cui specifiche sono riportate di seguito:

- **Modello**: *Redmi Note 8T*
- **Versione OS**: *Android 9.0*
- **RAM**: *4 GB*
- **CPU**: *Octa-core Max 2.01 GHz*

Gli strumenti utilizzati per misurare i tempi di esecuzione sono la classe `Stopwatch` di `System.Diagnostic` in Xamarin e la classe `Stopwatch` di `dart:core` in Flutter.

6.3.1 Start-Up

Con fase di **start-up** indichiamo il periodo di tempo trascorso dall'avvio dell'applicazione fino al caricamento completo della pagina sul display. Misureremo la velocità di avvio in tre diversi casi:

- *Applicazione "HelloWorld"*, in cui rileveremo il tempo di avvio di un'app appena creata in Xamarin.Android, Xamarin.Forms e Flutter;
- *MyTriageApp senza precedente login*, in cui rileveremo il tempo di avvio delle app **MyTriageAppXamarin** e **MyTriageAppFlutter** senza che ci sia il login automatico di una sessione precedente;
- *MyTriageApp con login precedente*, in cui rileveremo il tempo di avvio delle app **MyTriageAppXamarin** e **MyTriageAppFlutter** con login automatico e re-indirizzamento alla dashboard;

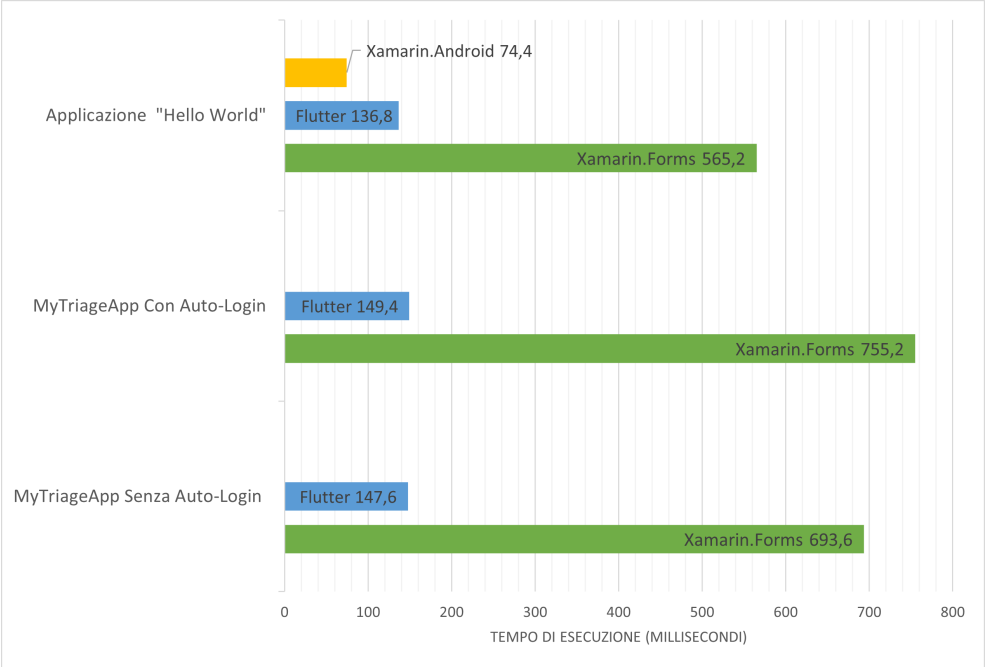


Figura 6.1: Grafico tempi di esecuzione fase di *start-up*

Dal grafico in figura 6.1 notiamo come tutte e tre le applicazioni sviluppate in Flutter impieghino approssimativamente lo stesso tempo per avviarsi, largamente inferiore a quello impiegato dalle applicazioni in Xamarin.Forms. Inoltre,

l'applicazione sviluppata nativamente in Xamarin.Android risulta la più veloce nello start-up, confermando la teoria dell'inefficienza del cross-platform.

6.3.2 Interazione con Firebase

Questo insieme di test punta ad analizzare la performance di comunicazione con i servizi di Firebase (*Authentication* e *Cloud Firestore*). Più precisamente, analizzeremo l'interazione rilevando il tempo di esecuzione della funzione di autenticazione (`LoginInWithEmailAndPassword`), di un metodo di tipo *GET* e di un metodo di tipo *ADD* (`GetGroups` e `AddGroup`).

Ognuna di queste rilevazioni verrà effettuata su tre applicazioni, sviluppate in Xamarin.Android, Xamarin.Forms e Flutter.

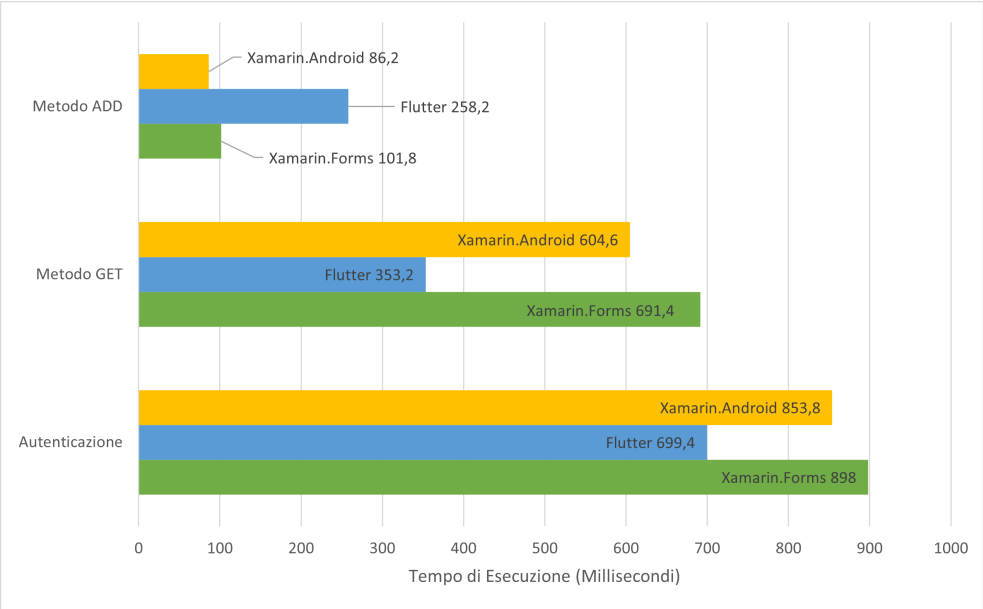


Figura 6.2: Grafico tempi di interazione con Firebase

Dal grafico in figura 6.2 possiamo notare come le latenze di interazione con i servizi Firebase per le applicazioni native Xamarin.Android e le applicazioni cross-platform Xamarin.Forms siano pressoché identiche. Questo perché l'implementazione è la stessa, dato che il progetto in Xamarin.Forms comprende al suo interno un sottoprogetto Xamarin.Android. Per quanto riguarda Flutter, è mediamente più veloce ad autenticarsi ed a recuperare informazioni che i suoi contendenti, mentre impiega quasi il triplo del tempo di Xamarin per inserire o aggiornare campi nel database Firestore.

6.3.3 Esportazione PDF

Arriviamo all'ultimo criterio di performance analizzato, l'**esportazione**. Più precisamente, rileviamo il tempo di esecuzione totale che le applicazioni impiegano per **generare il pdf** e **salvarlo** come file sulla memoria del dispositivo. Per avere dei dati congrui e consistenti, la composizione della scheda esportata e il percorso di salvataggio sono sempre gli stessi ad ogni rilevazione, per tutte le tre applicazioni (Xamarin.Android, Xamarin.Forms e Flutter).

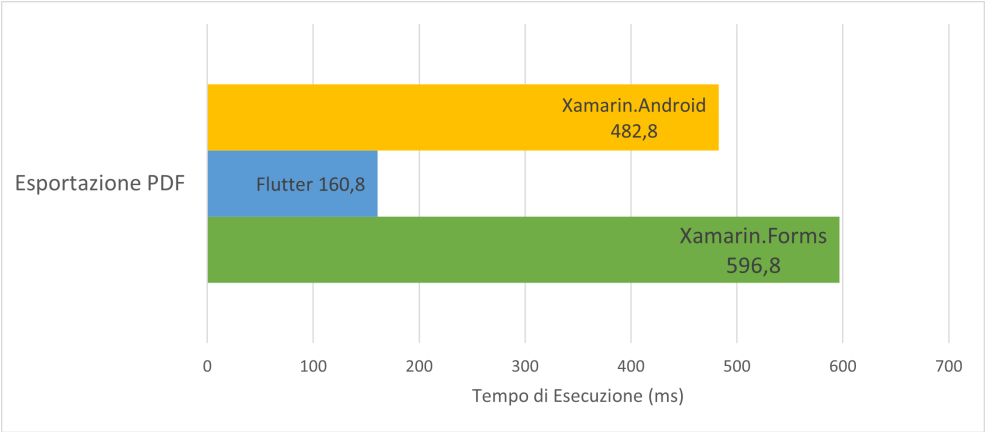


Figura 6.3: Grafico tempi di esportazione schede in PDF

I dati misurati, visibili nel grafico in figura 6.3, rimangono in linea con quelli visti finora: l'esportazione in Flutter è la più efficiente, mentre l'applicazione Xamarin nativa rimane, anche se non di molto, più veloce di quella cross-platform.

6.3.4 Performance Generali

Terminiamo il confronto delle performance tra Xamarin e Flutter con un riassunto generale delle prestazioni. Nel grafico in figura 6.4 sono riportate le medie dei tempi di esecuzione delle applicazioni prodotte in Xamarin.Forms e Flutter per le tre funzionalità analizzate.

Da questo confronto possiamo dedurre che le applicazioni cross-platform in Flutter siano più efficienti, con dei tempi di esecuzione che possono arrivare ad un quinto di quelli di Xamarin.Forms.

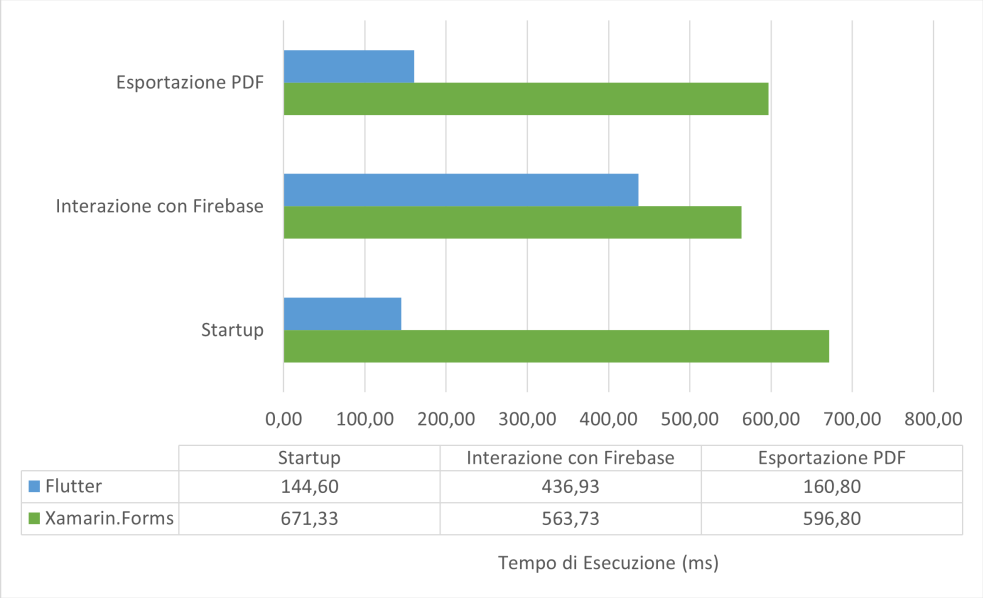


Figura 6.4: Grafico tempi di esecuzioni *Xamarin.Forms vs Flutter*

6.3.5 Dimensione Applicazioni

Un altro criterio da tenere in considerazione durante lo sviluppo di un'applicazione è la sua dimensione (*app size*). É un fattore molto importante che influenza anche la fase finale di rilascio nei vari app store.

Nella tabella 6.2 vengono paragonate le dimensioni delle app *Hello World* e *MyTriageApp* nei framework cross-platform Xamarin.Forms e Flutter, e nel framework nativo Xamarin.Android.

	Xamarin.Android	**Flutter**	**Xamarin.Forms**
Hello World	18.37 MB	32 MB	21.56 MB
MyTriageApp		60 MB	45.59 MB

Tabella 6.2: Tabella confronto dimensione applicazioni

Come tutti i framework cross-platform, anche Xamarin.Forms e Flutter producono applicazioni pesanti. Per quanto riguarda Xamarin, il prezzo pagato per il cross-platform sembra non essere eccessivo, poco più di 3 MB di differenza dal nativo nell'app *Hello World*. Flutter invece, produce applicazioni mediamente

più pesanti, soprattutto a causa della dimensione aggiuntiva che il Flutter *core engine* porta al file binario finale.

6.4 Tabella di Confronto

Per concludere il confronto tra le due piattaforme, analizziamo ancora alcuni parametri che, insieme a quelli già visti, compongono la **tabella di confronto** finale.

É importante ricordare che sia Flutter che Xamarin sono software ***open-source***, ma con differenti *price policy*. Nel caso di Flutter, la questione è molto semplice: è completamente gratuito, per qualsiasi tipo di utilizzo. Nel caso di Xamarin invece, la situazione cambia: il framework è tecnicamente open-source e quindi gratuito, ma la sua simbiosi con l'IDE di Visual Studio complica le cose. Per poter sviluppare software a *scopo commerciale*, Visual Studio richiede l'acquisto di una licenza, che può variare da 499$ a 2'999$, una cifra sicuramente da tenere in considerazione durante l'analisi dei costi di progettazione.

Inoltre, entrambe le piattaforme possono offrire performance a 60+ *frame-per-second* (fps) e, nel caso di Flutter, si possono raggiungere anche 120 fps nei dispositivi che lo supportano.

A ultimare i parametri di confronto della tabella troviamo la *riusabilità del codice*, dove Xamarin, con oltre il 96% di riusabilità, supera l'80% di Flutter, principalmente grazie all'utilizzo dei pattern architetturali di progettazione software (*MVVM* e *MVP*).

	Xamarin	**Flutter**
Rilasciato	Da *Microsoft*, Dicembre 2012	Da *Google*, Maggio 2017
Linguaggio di Programmazione	C# (C Sharp)	Dart
Open-Source	Sì	Sì
Documentazione	Ampia e completa	Ampia, ben strutturata e interattiva
Prezzo	Gratuito, ma per uso commerciale 499$ - 2'999$	Free-for-all
Dimensione App	Pesanti	Pesanti
Riusabilità del Codice	Fino al 96%	Fino al 80%
Community	Larga e consolidata	Relativamente giovane ma in crescita
Componenti UI	Componenti nativi	*Widget-Tree*, componenti built-in personalizzabili
60+ FPS	Sì	Sì
Piattaforme Supportate	*iOS, Android, MacOS, AndroidWear, AndroidTv, WatchOS, TvOS*	*iOS, Android, Linux, MacOS e WebApp*

Tabella 6.3: Tabella confronto finale *Xamarin vs Flutter*

Conclusione

Per quanto riguarda il progetto *MyTriageApp*, tramite entrambe le tecnologie è stato possibile implementare tutte le funzionalità preposte in fase di progettazione.

Come sviluppo futuro dell'app, sarebbe molto utile aggiungere la possibilità di salvare, quando si crea un gruppo, oltre che alle generalità di una persona anche le informazioni riguardanti la certificazione verde (ad es. la data di scadenza) nel caso in cui sia richiesta per entrare nell'edificio. In questo modo, anche nella pagina di visualizzazione del gruppo si potranno distinguere con colori diversi i membri che hanno certificazioni valide oppure scadute.

Il percorso che ha portato allo sviluppo delle due applicazioni, *MyTriageAppXamarin* e *MyTriageAppFlutter*, e che ha portato all'autoapprendimento delle tecnologie utilizzate, aveva come fine ultimo la raccolta di informazioni per studiare e confrontare le due piattaforme.

Bisogna premettere che la scelta del framework di sviluppo per un applicativo mobile non è mai semplice e immediata, "o bianca o nera", ma dipende da molti fattori anche non oggettivi, tra cui conoscenze e sensazioni del programmatore. Per questo nelle considerazioni finali verrà, per quanto possibile, tralasciato l'aspetto soggettivo.

Xamarin si è dimostrata una piattaforma completa che, integrata perfettamente a Visual Studio e all'ambiente Microsoft, fornisce supporto completo allo sviluppatore. La sua implementazione del cross-platform (Xamarin.Forms) non è altro che un livello di astrazione superiore ai progetti nativi (Xamarin.Android e Xamarin.iOS) che traduce i componenti UI in componenti nativi. Qui nasce il più grande svantaggio di questa multi-piattaforma: scrivere del codice specifico nei sotto progetti per Android e iOS (come spiegato nella sezione 2.2.3) richiede un'ulteriore conoscenza degli strumenti di sviluppo nativi destinati a entrambi i sistemi operativi.

Questo in Flutter non è richiesto, perché il framework sposa completamente il concetto di programmazione cross-platform, in cui la totalità del codice, senza sostanziali modifiche, può essere compilato per iOS oppure per Android.

Di conseguenza, per uno sviluppatore che per la prima volta approccia la programmazione mobile, la scelta più ragionevole è apprendere ed utilizzare Flutter. Chiaramente, per uno sviluppatore familiare con strumenti come Xamarin.Android e Xamarin.iOS, utilizzare Xamarin.Forms sembra più adatto che imparare ad usare un framework totalmente nuovo e diverso.

Lo svantaggio che la struttura di Xamarin.Forms comporta è ormai noto nell'ambito della programmazione multi-piattaforma e, anche per questo, il team di sviluppatori Microsoft ha annunciato il lancio di una nuova tecnologia cross-platform, NET MAUI (.NET Multi-platform App UI), che semplificherà lo sviluppo di app native multipiattaforma, strutturate in un solo progetto e con feature come l'*hot reload* totale (rilascio a Novembre 2021).

Bibliografia

[1] BaseFlow.com. *Permission plugin for Flutter*. URL: https://pub.dev/packages/permission_handler.

[2] Creately. *Diagramma dei casi d'uso in UML*. URL: https://creately.com/blog/it/uncategorized-it/tutorial-diagramma-di-attivita/.

[3] Apple Developers. *XCode 13*. URL: https://developer.apple.com/xcode/.

[4] Google Developers. *Android Studio*. URL: https://developer.android.com/studio.

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

[6] Google Developers. *Material Design*. URL: https://material.io.

[7] firebase.google.com. *Firebase Auth - A Flutter plugin to use the Firebase Authentication API*. URL: https://pub.dev/packages/firebase_auth.

[8] firebase.google.com. *Firebase Core - A Flutter plugin to use the Cloud Firestore API*. URL: https://pub.dev/packages/cloud_firestore.

[9] firebase.google.com. *Firebase Core - A Flutter plugin to use the Firebase Core API, which enables connecting to multiple Firebase apps*. URL: https://pub.dev/packages/firebase_core.

[10] Ayush Agarwal Jeremiah Ogbomo. *Flutter Spinkit - A collection of loading indicators animated with flutter*. URL: https://pub.dev/packages/flutter_spinkit.

[11] Amit Manchand. *The Ultimate Guide to Cross Platform App Development Frameworks in 2021*. URL: https://docs.microsoft.com/en-us/xamarin/get-started/. (accessed: 30.08.2021).

[12] James Montemagno. *Collection of MVVM helper classes for any application.* URL: https://github.com/jamesmontemagno/mvvm-helpers.

[13] nfet.net. *A pdf producer for Dart.* URL: https://pub.dev/packages/pdf.

[14] Syncfusion. *Create, Read, and Edit PDF Files with Syncfusion in Xamarin.* URL: https://www.syncfusion.com/pdf-framework/xamarin/pdf-library.

[15] Flutter Dev. Team. *Cupertino Design.* URL: https://flutter.dev/docs/development/ui/widgets/cupertino.

[16] Flutter Dev. Team. *Flutter.* URL: https://flutter.dev. (accessed: 1.09.2021).

[17] Flutter Dev. Team. *Flutter Architectural Overview.* URL: https://flutter.dev/docs/resources/architectural-overview. (accessed: 1.09.2021).

[18] Microsoft Team. *From Data Bindings to MVVM.* URL: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/xaml/xaml-basics/data-bindings-to-mvvm?WT.mc_id=dotnet-35129-website. (accessed: 1.09.2021).

[19] Microsoft Team. *iOS App Architecture.* URL: https://docs.microsoft.com/en-us/xamarin/ios/internals/architecture. (accessed: 31.08.2021).

[20] Microsoft Team. *Visual Studio 2019.* URL: https://visualstudio.microsoft.com/it/vs/.

[21] Microsoft Team. *Visual Studio Code.* URL: https://code.visualstudio.com/?wt.mc_id=DX_841432.

[22] Microsoft Team. *What is Xamarin?* URL: https://docs.microsoft.com/en-us/xamarin/get-started/. (accessed: 31.08.2021).

[23] Microsoft Team. *What is Xamarin.Forms?* URL: https://docs.microsoft.com/en-us/xamarin/get-started/what-is-xamarin-forms. (accessed: 31.08.2021).

[24] Microsoft Team. *Xamarin.Android Architecture.* URL: https://docs.microsoft.com/en-us/xamarin/android/internals/architecture. (accessed: 31.08.2021).

[25] Microsoft Team. *Xamarin.Android Bindings for Firebas.Firestore.* URL: https://www.nuget.org/packages/Xamarin.Firebase.Firestore.

[26] Microsoft Team. *Xamarin.Android Bindings for Firebase Auth.* URL: https://www.nuget.org/packages/Xamarin.Firebase.Auth/.

[27] Microsoft Team. *Xamarin.Android Core Bindings for Firebase.* URL: https://www.nuget.org/packages/Xamarin.Firebase.Core/.

[28] Vito La Vecchia. *I Diagrammi delle Attività in UML.* URL: https://vitolavecchia.altervista.org/diagramma-delle-attivita-uml/.

[29] yasukotelin. *ext_storage is a minimal flutter plugin that provides external storage path.* URL: https://pub.dev/packages/ext_storage.