Università Politecnica delle Marche

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica e dell'Automazione

Progettazione e realizzazione di un'applicazione per il supporto alla vita universitaria

Corso di Programmazione Mobile

Prof: Domenico Ursino

Autori:

Lapi Denaldo

Monini Marco

Politano Antonio

Indice

Elenco delle figure

Listings

Introduzione

La vita dello studente universitario è meno regolare e scandita rispetto a quella di uno studente liceale o di altri istituti. Il periodo di quarantena, dovuto alla pandemia da Coronavirus, in particolare, ci ha fatto notare ancora di più come sia facile, in alcune situazioni, perdere il senso del tempo e dei giorni. L'unica cosa dalla quale, ormai, non ci si separa praticamente mai è il proprio cellulare. Abbiamo pensato, quindi, di sviluppare un'applicazione che supporti lo studente nel percorso di studio universitario e che permetta di gestire in maniera semplice ed intuitiva le problematiche relative all'organizzazione della vita studentesca.

Gli obiettivi dell'applicazione sono dunque quelli di permettere allo studente di inserire e gestire gli orari delle lezioni, i corsi seguiti con eventuali appunti, gli appelli previsti e le eventuali attività che lo studente si prefigge di svolgere entro una data scadenza.

Il progetto verrà realizzato in due versioni, utilizzando due delle tre diverse tecniche di programmazione mobile, ovvero app nativa e app ibrida. In particolare, l'app nativa verrà sviluppata per il sistema operativo Android, utilizzando Android Studio come IDE e Java come linguaggio di programmazione. L'app ibrida, anch'essa particolarizzata per Android, verrà, invece, realizzata mediante il framework Xamarin, utilizzando Visual Studio come IDE e C# come linguaggio di programmazione.

Capitolo 1

Analisi dei requisiti

In questo capitolo ci occuperemo dell'analisi dei requisiti, un'attività fondamentale che precede la fase di progettazione vera e propria.

Questa fase consiste nel descrivere le funzionalità dell'applicazione utilizzando, inizialmente, un linguaggio naturale e, successivamente, un linguaggio di modellazione in grado di illustrare tali funzionalità in maniera schematica.

Prima di addentrarci nei dettagli dell'analisi dei requisiti, illustriamo brevemente le caratteristiche principali dell'applicazione che si intende sviluppare, al fine di estrarne i requisiti stessi.

1.1 Descrizione del progetto

L'applicazione da sviluppare dovrà permettere allo studente di organizzare la propria attività di studio, ovvero dovrà consentirgli una semplice gestione del materiale didattico, della frequentazione delle lezioni e del sostenimento degli esami.

Ogni utente dovrà registrarsi al primo utilizzo dell'applicazione e, per ogni utilizzo successivo, il login sarà automatico.

All'interno dell'app si potranno inserire i corsi seguiti con relativi orari di lezione e professori che tengono il corso. Consentiremo inoltre la possibilità di inserire appunti o altro materiale didattico visualizzabile all'interno dell'applicazione.

Prevediamo di inserire, anche, una sezione nella quale lo studente organizzerà la propria settimana, scegliendo quali attività svolgere ogni giorno inserendole nella propria *To Do list*.

Infine, abbiamo intenzione di permettere all'utente di inserire gli appelli di esame da svolgere, con relativa data ed orario.

Dovremo fornire un'app con un'interfaccia leggera, minimale ma gradevole, in modo da fornire un utilizzo semplice e veloce ad ogni utente dato che l'app verrà utilizzata più volte al giorno.

In conclusione, l'utilità dell'applicazione sarà quella di aumentare la produttività dello studente e di aiutarlo nella gestione del tempo dedicato allo studio.

1.2 Requisiti funzionali e non funzionali

In questa sezione verranno illustrati i requisiti funzionali e non funzionali dell'applicazione che si vuole sviluppare.

I requisiti funzionali rappresentano l'insieme delle funzionalità che dovranno essere implementate nell'applicazione.

I requisiti non funzionali, invece, definiscono l'insieme dei vincoli e delle regole, sia di tipo realizzativo che di tipo tecnico.

Analizziamo, ora, nel dettaglio, tali requisiti.

1.2.1 Requisiti funzionali

Le funzionalità principali che l'app si propone di garantire sono le seguenti:

- *Attività CRUD per i corsi*: si deve avere la possibilità inserire, leggere, modificare e cancellare i corsi seguiti con relativi orari di lezione, professore e appunti di riferimento.
- *Attività CRUD per gli appunti*: si deve avere la possibilità inserire, leggere, modificare e cancellare appunti relativi ai vari corsi.
- *Attività CRUD per la To Do list*: si deve avere la possibilità inserire, leggere, modificare e cancellare le attività settimanali da svolgere nella *To Do list*.

- *Attività CRUD per gli appelli*: si deve avere la possibilità inserire, leggere, modificare e cancellare gli appelli d'esame, con relativa data ed orario.

- *Attività CRUD per l'orario*: si deve avere la possibilità inserire, leggere, modificare e cancellare gli orari delle lezioni, o comunque, gli orari relativi alle attività da svolgere durante la settimana.

1.2.2 Requisiti non funzionali

Poiché l'applicazione dovrà essere usata dagli studenti, abbiamo la necessità di renderla intuitiva e con un design accattivante, ma allo stesso tempo semplice. Inoltre, questa, dovrà risultare il più possibile fluida durante l'esecuzione.

I requisiti non funzionali sono:

- L'app dovrà avere un design moderno e una grafica accattivante.

- L'app dovrà essere user friendly ed intuitiva.

- L'app dovrà avere delle buone performance ed essere reattiva.

- L'app dovrà essere il più affidabile possibile. In particolare, si dovranno evitare crash improvvisi o errori di varia natura.

- L'app dovrà gestire i comportamenti anomali degli utenti (inserimenti errati durante le procedure di login e di registrazione, campi vuoti negli inserimenti di attività, corsi, appunti, etc.).

- All'avvio dell'app viene effettuato automaticamente il login se all'ultimo utilizzo un utente loggato non ha effettuato la procedura di logout.

- L'app utilizzerà, per quanto riguarda la gestione della procedura di autenticazione degli utenti e il salvataggio dei dati inseriti da questi, un database esterno fornito dal servizio Firebase di Google. Nelle sezioni successive vedremo nel dettaglio l'implementazione di questo strato di persistenza dei dati.

1.3 Diagramma dei casi d'uso

Cerchiamo, ora, di illustrare i requisiti appena descritti mediante una rappresentazione schematica. In particolare, in questa fase ci occuperemo di individuare gli attori e i casi d'uso dell'applicazione, ovvero di creare il diagramma dei casi d'uso (use case diagram).

I diagrammi dei casi d'uso sono dei diagrammi definiti in UML (*Unified Modeling Language*) che permettono di analizzare i requisiti di un sistema fornendo una rappresentazione grafica delle funzionalità (*casi d'uso*) messe a disposizione degli utenti (*attori*) che con esso interagiscono.

In particolare, gli elementi che costituiscono un diagramma dei casi d'uso sono attori, casi d'uso e relative associazioni.

Come prima cosa, è necessario stabilire i cosiddetti *Attori* della nostra applicazione, ovvero gli utilizzatori della stessa. Nel nostro caso, gli utenti dell'applicazione saranno esclusivamente studenti, quindi non avremo la necessità di creare diverse tipologie di utenze.

Il secondo passo da affrontare per la rappresentazione schematica dei requisiti dell'applicazione consiste nella costruzione dei diagrammi dei casi d'uso.

Possiamo, ora, costruire un diagramma che indica i differenti casi d'uso della nostra applicazione, relativi all'unica tipologia di utenza prevista (Figura 1.1).

Si noti l'utilizzo dell'etichetta *extend*, la quale esprime una dipendenza tra casi d'uso. In particolare, essa descrive la relazione tra un caso d'uso *subordinato* che estende il comportamento di un caso d'uso *principale*, e il caso d'uso *principale* stesso.

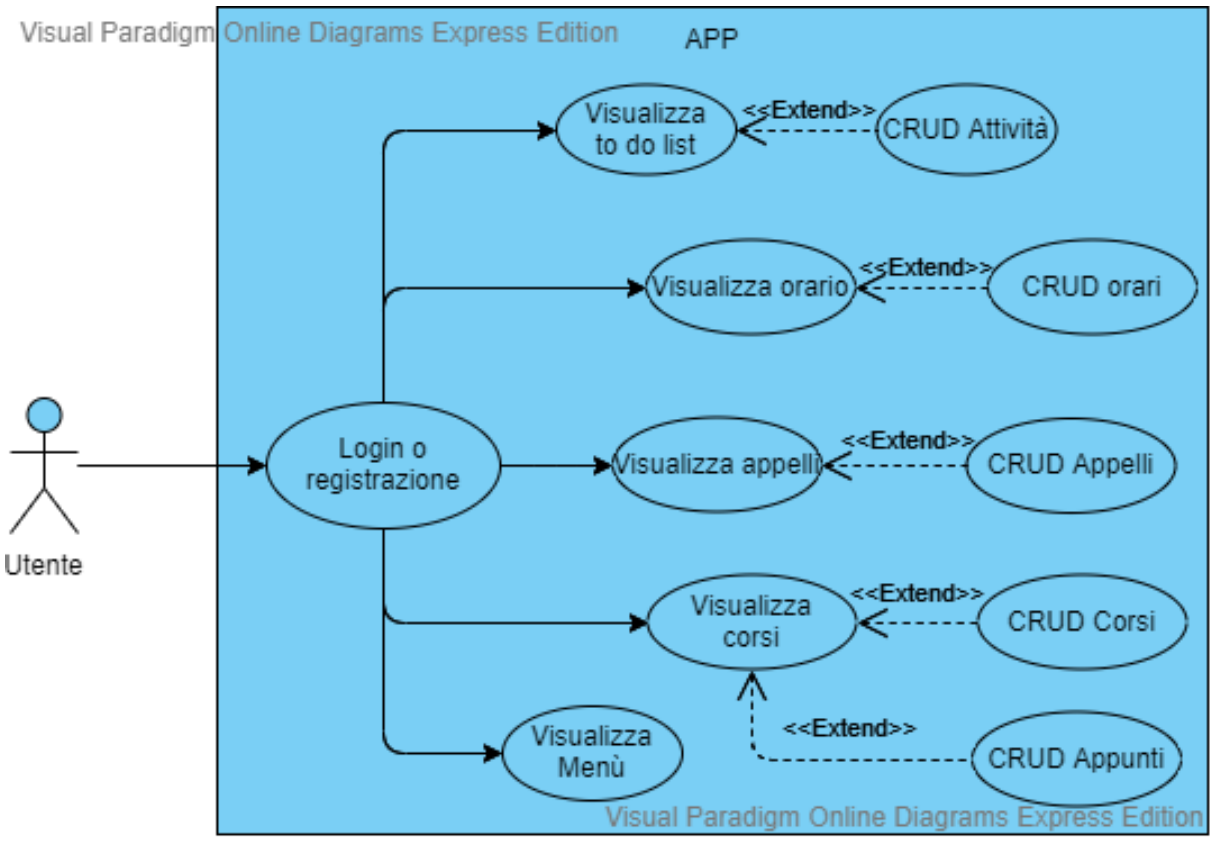

Figura 1.1: Diagramma dei casi d'uso

Capitolo 2

Progettazione

La fase di progettazione della componente applicativa di un'applicazione mobile consiste nella progettazione vera e propria delle funzionalità che l'app offre e che sono state descritte nel capitolo precedente.

I primi passi della progettazione consistono nella realizzazione delle seguenti componenti:

- mappa dell'applicazione;
- mockup.

Andiamo, ora, a vedere nel dettaglio la realizzazione di queste due componenti, in modo da dare un'idea di come funzioni l'app e di che aspetto debba avere.

2.1 Mappa dell'applicazione

La mappa dell'applicazione è una rappresentazione schematica della struttura dell'app, che consente di visualizzare immediatamente come i contenuti forniti siano organizzati nelle diverse pagine. L'organizzazione dei contenuti, in particolare, deve essere tale da permettere una navigazione intuitiva all'interno dell'app e, quindi, essa deve contribuire a garantire una user experience soddisfacente.

In Figura 2.1 riportiamo la mappa relativa alla nostra applicazione, la quale rappresenta schematicamente la struttura dell'app stessa.

Si noti come, nella mappa, vengono utilizzati colori differenti in base ai diversi livelli di navigazione tra le pagine dell'app.

Lo scopo di tale mappa è, perciò, quello di illustrare, in maniera ancora informale, il flusso delle azioni da compiere per usufruire delle varie funzionalità offerte dall'app.

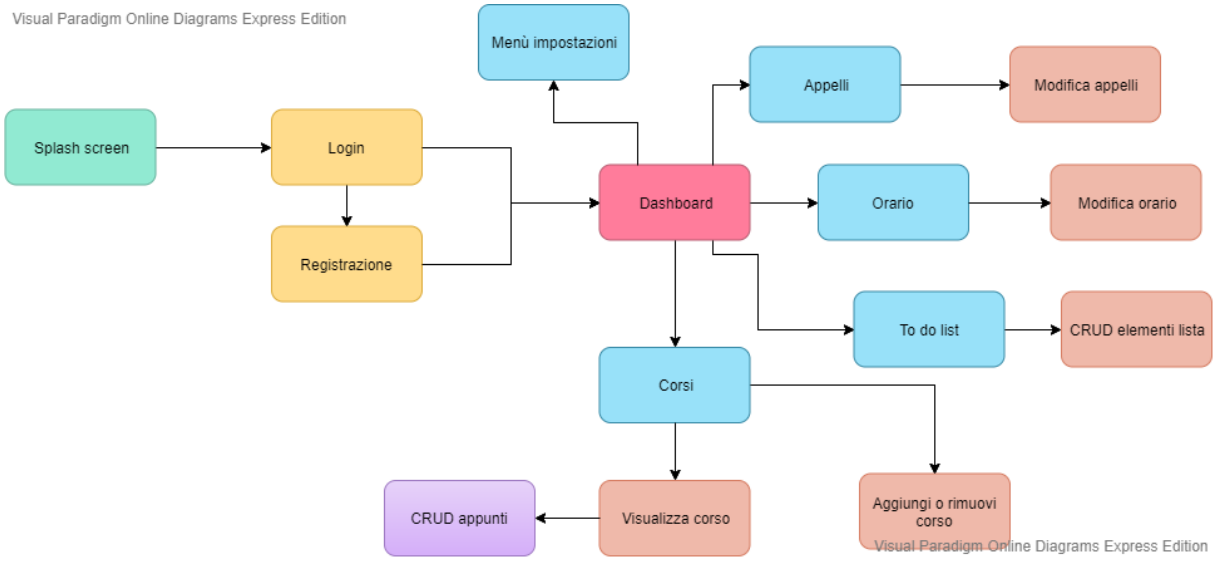

Figura 2.1: Mappa dell'applicazione

In particolare, il grafico sopra riportato illustra come, all'avvio dell'app, viene mostrata, inizialmente, una *Splash screen* che, nel caso del primo avvio dell'applicazione, porta ad una schermata per la procedura di login/registrazione. Quest'ultima permette, poi, all'utente di identificarsi accedendo ad un account esistente, o di registrarsi creando un nuovo profilo. Inoltre, l'autenticazione viene mantenuta per i successivi utilizzi dell'app.

Dopo l'eventuale autenticazione, viene aperta la schermata di *Dashboard* nella quale, utilizzando una semplice barra di navigazione (*bottom navigation view*), si può navigare tra le diverse sezioni dell'applicazione, potendo così accedere alle vere e proprie funzionalità della stessa. Inoltre, per accedere al menù *Impostazioni* (che comprende pagine per le *Faq*, l'*Informativa sulla privacy* e i *Contatti*) è presente un apposito menù nella toolbar (*Overflow Menù*).

2.2 Mockup

Il mockup è la rappresentazione grafica, pulita e stilizzata, delle schermate che costituiscono l'applicazione da sviluppare. In particolare, esso serve per rappresentare i vari contenuti e per illustrare l'aspetto grafico del progetto.

In base al grado di dettaglio, si possono avere tre differenti tipologie di mockup:

- *Mockup di Livello 0*: in cui ci si concentra sui contenuti, senza perdersi in dettagli grafici.
- *Mockup di Livello 1*: in cui si arricchisce il Livello 0 con dei dettagli grafici.
- *Mockup di Livello 2*: in cui si realizza una sorta di copia del progetto finale. In tale mockup la grafica gioca un ruolo fondamentale e si evidenziano colori e forme.

Riportiamo i mockup di livello 0, rappresentanti la prima progettazione grafica effettuata. Più avanti presenteremo direttamente gli screen dell'app.

Nello specifico, vedremo i mockup relativi alle seguenti schermate:

- Schermata di registrazione e login (Figura 2.2).
- Schermata relativa alla lista delle attività della *To do* (Figura 2.3).
- Schermata relativa alla visualizzazione della lista dei corsi, all'inserimento di un nuovo corso e alla visualizzazione della lista degli appunti di un determinato corso (Figura 2.4).
- Schermata relativa alle pagine di visualizzazione dell'orario e di inserimento di un nuovo evento (Figura 2.5).
- Schermate relative alla visualizzazione della lista degli appelli ed inserimento di un nuovo appello (Figura 2.6).
- Visualizzazione dell'*Overflow menu* con relative opzioni (Figura 2.7), che consentono l'accesso alle pagine secondarie dell'applicazione.

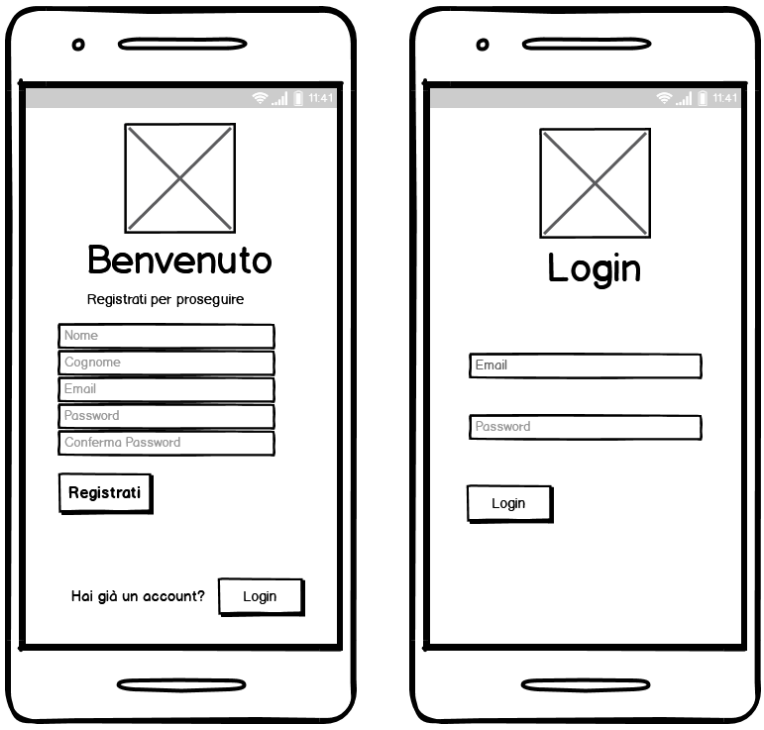

Figura 2.2: Mockup riguardante le schermate di registrazione e di login

Figura 2.3: Mockup riguardante la lista delle attività e l'inserimento di una nuova attività

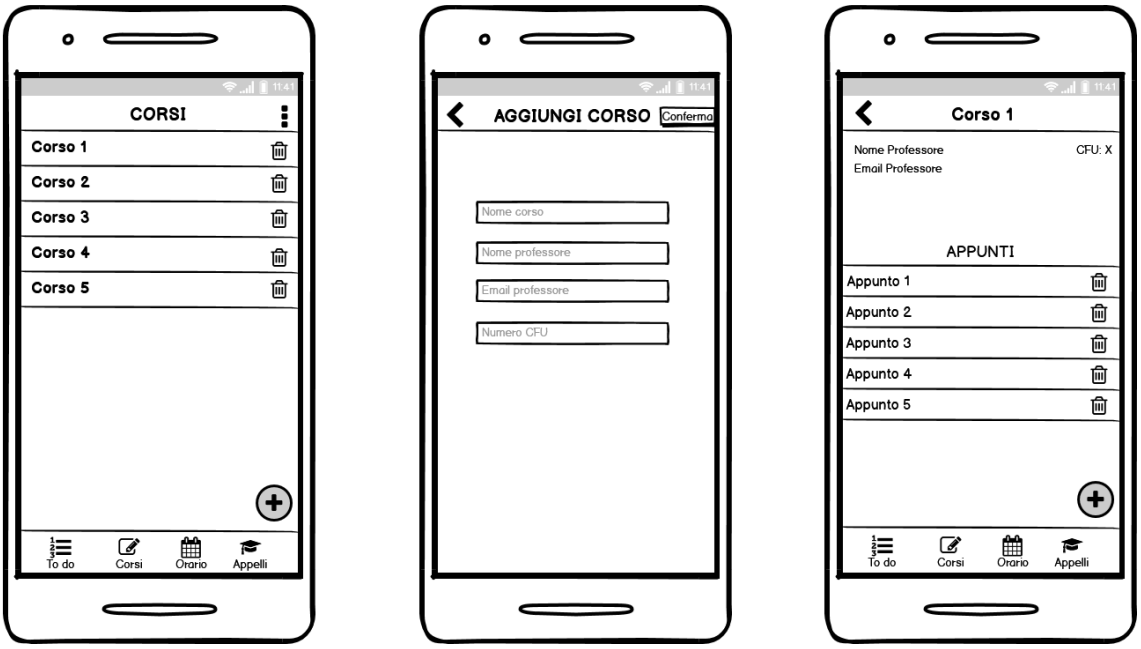

Figura 2.4: Mockup riguardante la lista dei corsi, l'inserimento di un appunto e la lista degli appunti di un corso

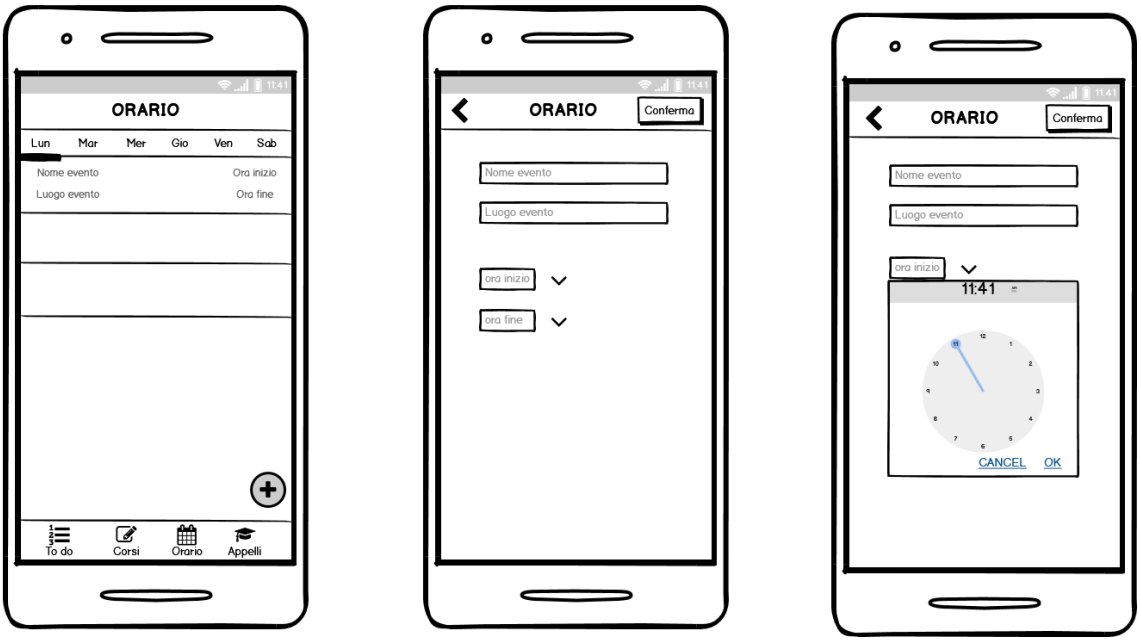

Figura 2.5: Mockup riguardante la visualizzazione dell'orario e l'inserimento di un nuovo evento

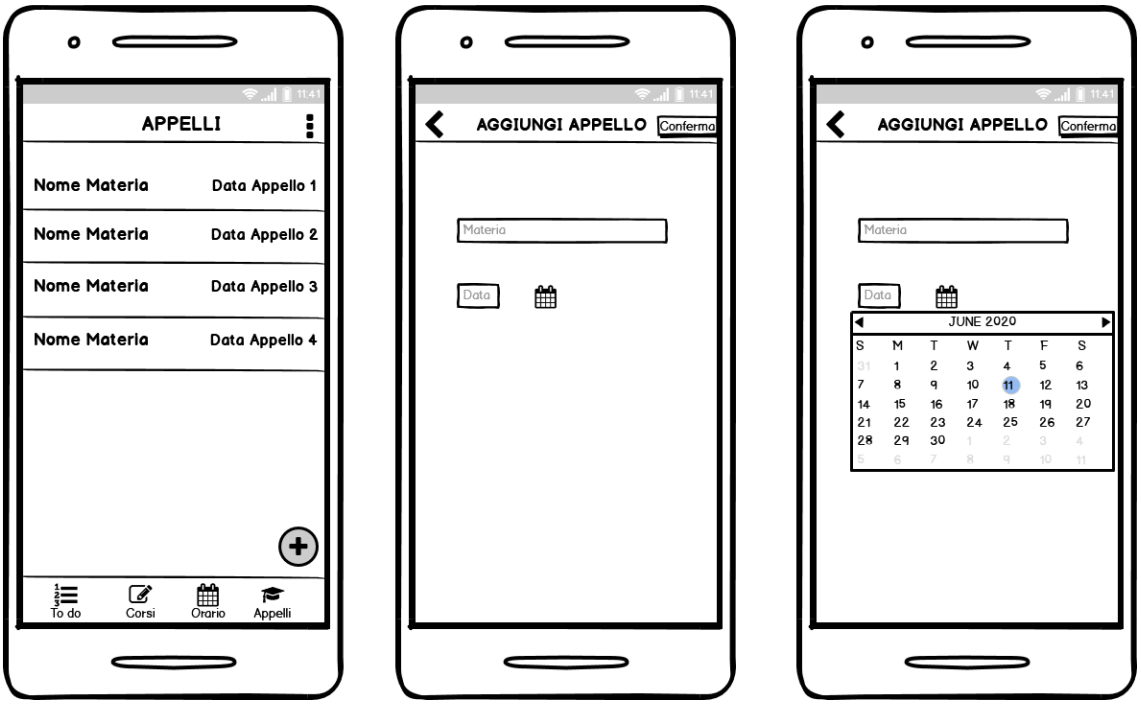

Figura 2.6: Mockup riguardante la lista degli appelli e l'inserimento di un nuovo appello

Figura 2.7: Mockup riguardante la visualizzazione dell'*Overflow menu*

2.3 Progettazione del database

Un'altra fase fondamentale della progettazione di qualsiasi applicazione mobile è l'implementazione dello strato di persistenza dei dati.

Andiamo, ora, ad analizzare come è stata implementata questa funzionalità.

2.3.1 Firebase

Abbiamo deciso di utilizzare, per lo strato di permanenza dei dati, un servizio di backend realizzato da Google, ovvero Firebase. Quest'ultimo offre servizi di database, di autenticazione, di hosting web, di analisi del prodotto e di diagnostica degli utenti, oltre a moltissime altre funzioni. Inoltre, il servizio è ben documentato, oltre che largamente supportato da Android, IOS e dalle applicazioni Web.

2.3.2 Motivo della scelta

Abbiamo optato per l'utilizzo di Firebase per la sua semplicità di utilizzo e anche perché le funzionalità offerte rientrano perfettamente nei nostri requisiti.

Infatti, con l'utilizzo dell'autenticazione offerta da Firebase, abbiamo la possibilità di far mantenere ad ogni utente i propri dati indipendentemente dal dispositivo utilizzato.

Inoltre, il fatto di poter utilizzare il server offertoci da Firebase, invece di doverlo implementare, oltre a farci risparmiare tempo, ci garantisce una maggiore robustezza ed affidabilità.

2.3.3 Considerazioni sul database

Firebase offre due diverse opzioni di database: Cloud Firestore e Realtime Database. Questi sono entrambi database non relazionali NoSQL.

Un database NoSQL (*Not Only SQL*) non prevede la conservazione dei dati in tabelle definite a priori, ma utilizza una struttura basata su raccolte di collezioni e documenti, reperibili tramite un identificatore e le cui informazioni sono organizzate attraverso una serie di campi.

Il database in questione, a differenza del classico database relazionale, non è caratterizzato da una strutturazione rigida dei contenuti. Questa caratteristica, però, permette di ottenere

migliori prestazioni quando si lavora con enormi quantità di dati, poiché, a differenza di un database SQL, non è necessario verificare la correttezza dei dati (integrità dei dati) e quella delle relazioni fra tabelle (integrità referenziale).

2.3.4 Implementazione database

Per quanto riguarda la progettazione del database, abbiamo comunque scelto di seguire i pattern classici della progettazione per i database SQL tradizionali. Però, data la semplicità della nostra base di dati, siamo passati direttamente alla progettazione dello schema logico.

In particolare, le raccolte ed i loro campi sono stati definiti secondo lo schema ad albero illustrato in Figura 2.8.

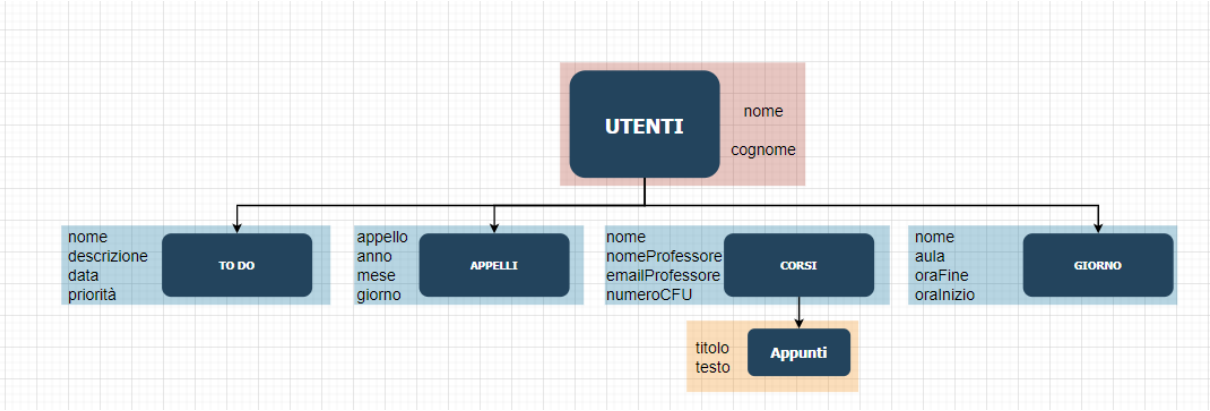

Figura 2.8: Schema Database

Andiamo, ora, ad analizzare le singole collezioni e i singoli documenti del nostro database di Firebase Firestore.

NB: se non è specificato il tipo dell'attributo si intende di tipo stringa. Inoltre l'identificatore è il nome stesso del file rappresentante il documento e viene determinato automaticamente.

Utenti (nome, cognome)

La collezione*Utenti* (Figura 2.9) è la radice del nostro database e contiene ogni altra raccolta all'interno dei propri documenti, i quali rappresentano i singoli utilizzatori dell'app.

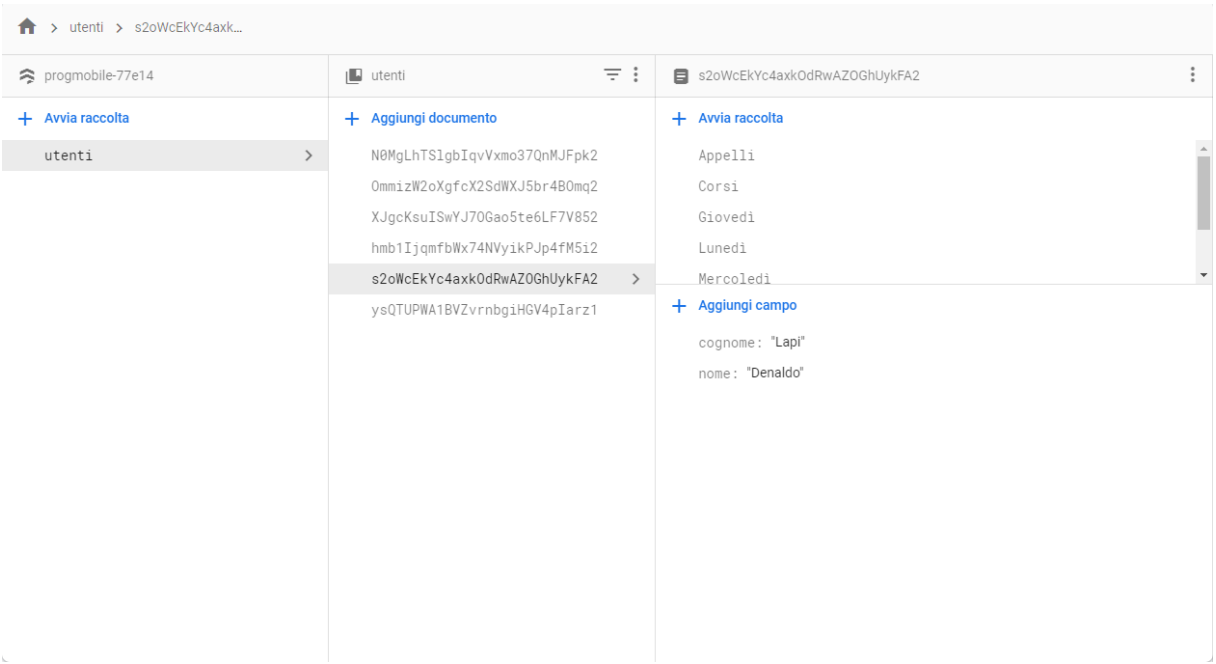

Figura 2.9: Collezione *Utenti*

ToDo (data, descrizione, nome, priorita:*numero*)

La collezione *ToDo* (Figura 2.10) contiene all'interno di ogni suo documento un'attività da svolgere con relativo nome, descrizione, data e un attributo che rappresenta la priorità.

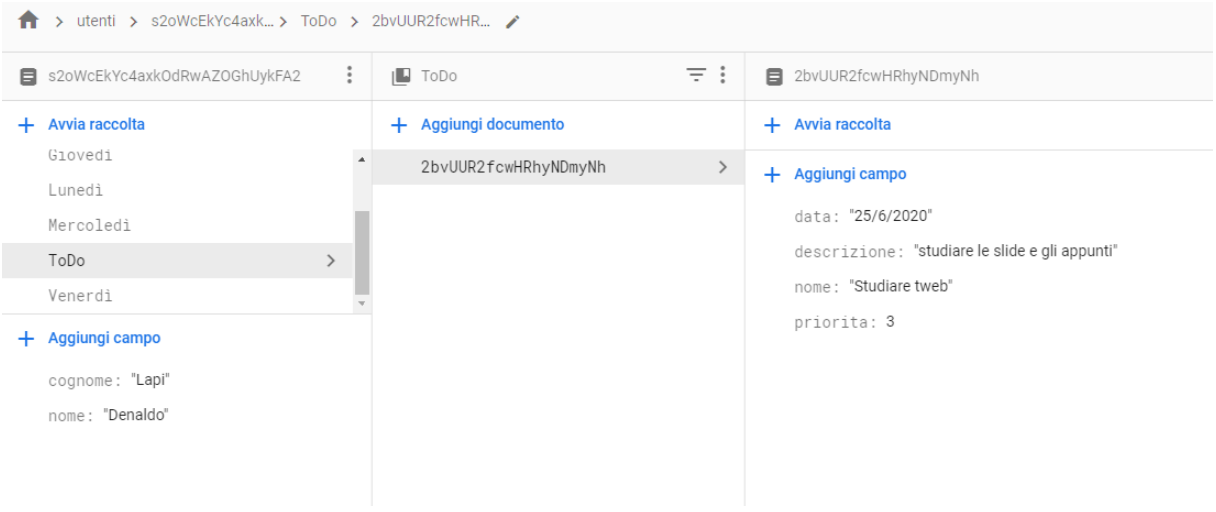

Figura 2.10: Collezione *ToDo*

Appelli (materia, giorno:*numero*, mese:*numero*, anno:*numero*)

La collezione *Appelli* (Figura 2.11) contiene all'interno di ogni suo documento un appello di esame con relativa materia, giorno, mese e anno.

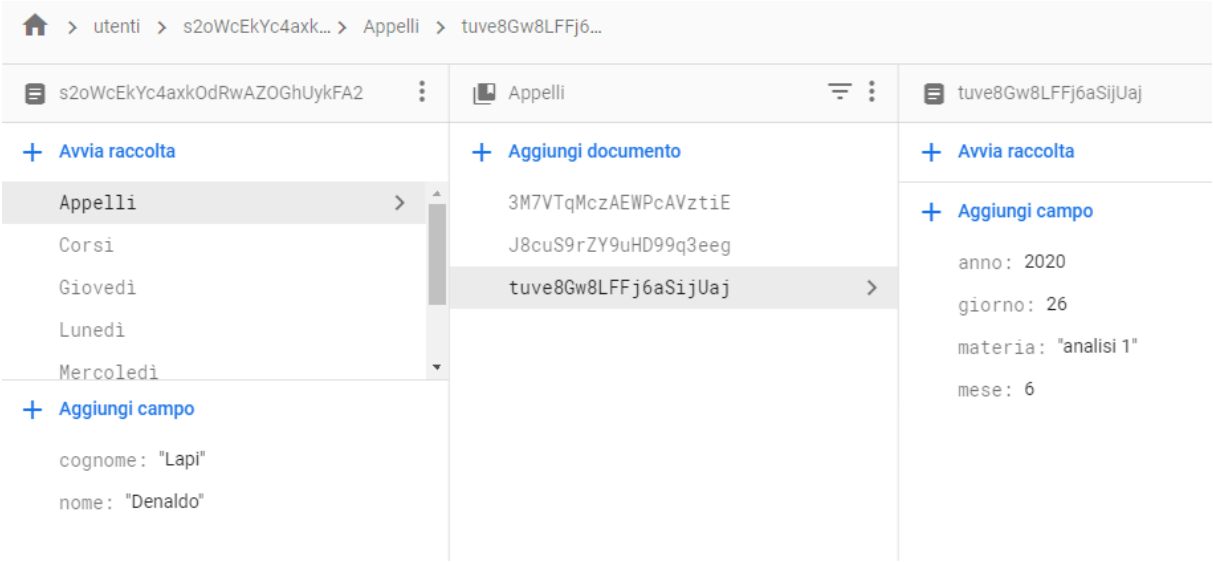

Figura 2.11: Collezione *Appelli*

Giorno (nome, aula, oraInizio, oraFine)

Esiste una collezione per ogni giorno della settimana, ma essendo queste molto simili, ne riportiamo solo una per esempio (*Lunedi*).

La collezione *Lunedi* (Figura 2.12) contiene all'interno di ogni suo documento un evento con relativo nome, aula, ora di inizio ed ora fine.

Corsi (nome, nomeProfessore, emailProfessore, numeroCFU:*numero*)

La collezione *Corsi* (Figura 2.13) contiene all'interno di ogni suo documento un corso con i seguenti campi: nome, nome professore, email professore e numero CFU. Inoltre ogni documento della collezione può contenere all'interno un'ulteriore collezione *Appunti* che ovviamente conterrà gli appunti relativi a tale corso.

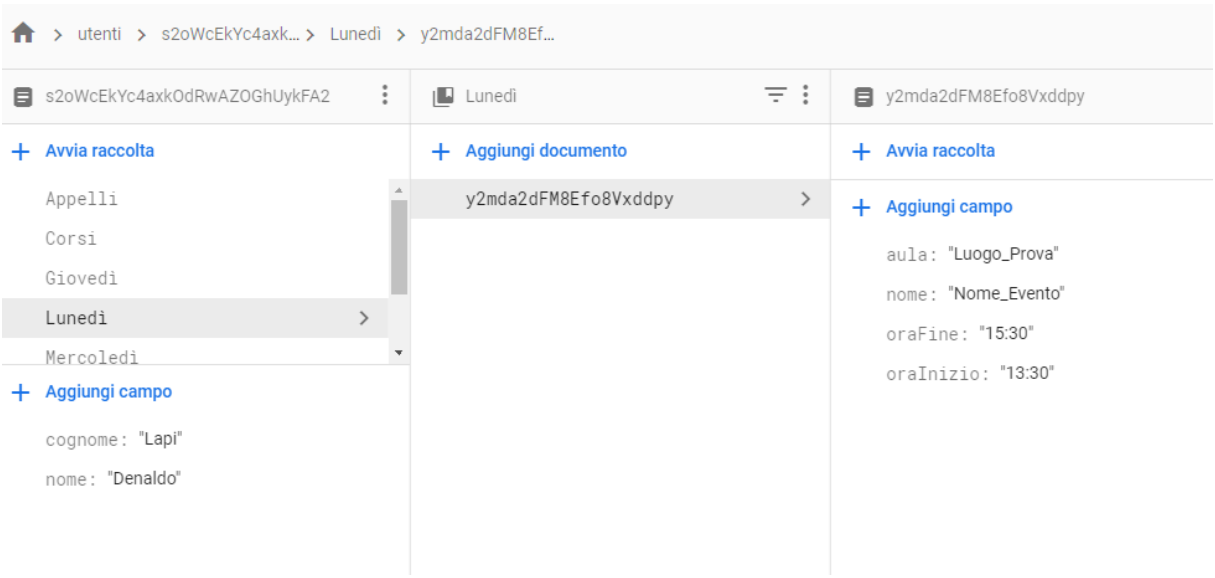

Figura 2.12: Collezione Giorno

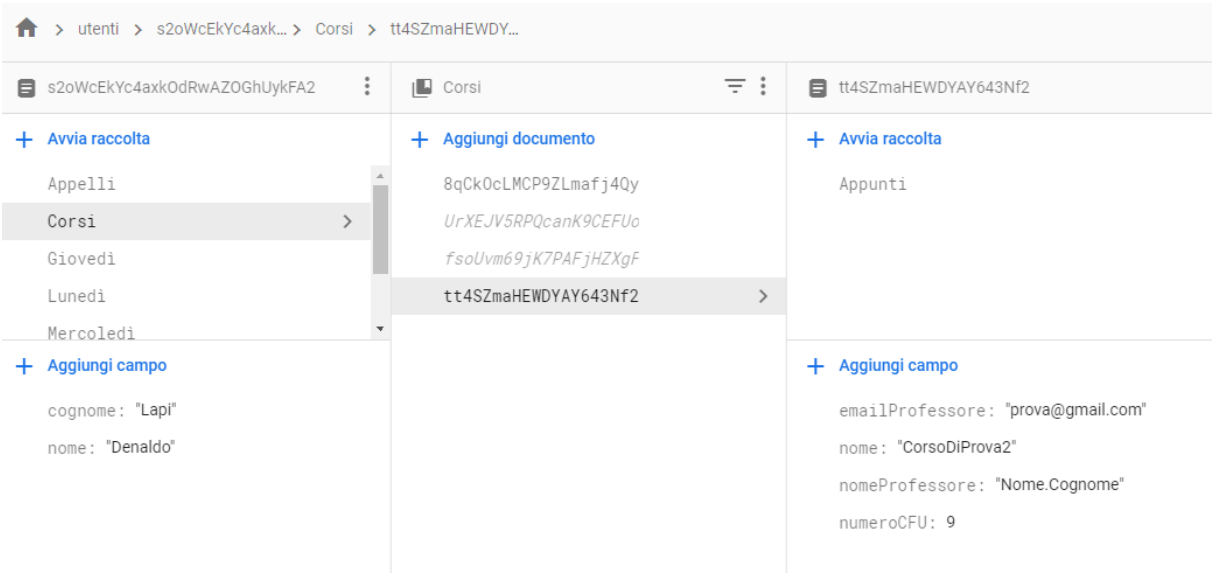

Figura 2.13: Collezione *Corsi*

Appunti (titolo, testo)

La collezione *Appunti* (Figura 2.14), come anticipato, si trova ad un livello inferiore rispetto alle altre, sempre all'interno di un documento appartenente alla collezione *Corsi*. Ogni documento della collezione *Appunti* presenta i campi relativi al titolo dell'appunto stesso e al suo testo.

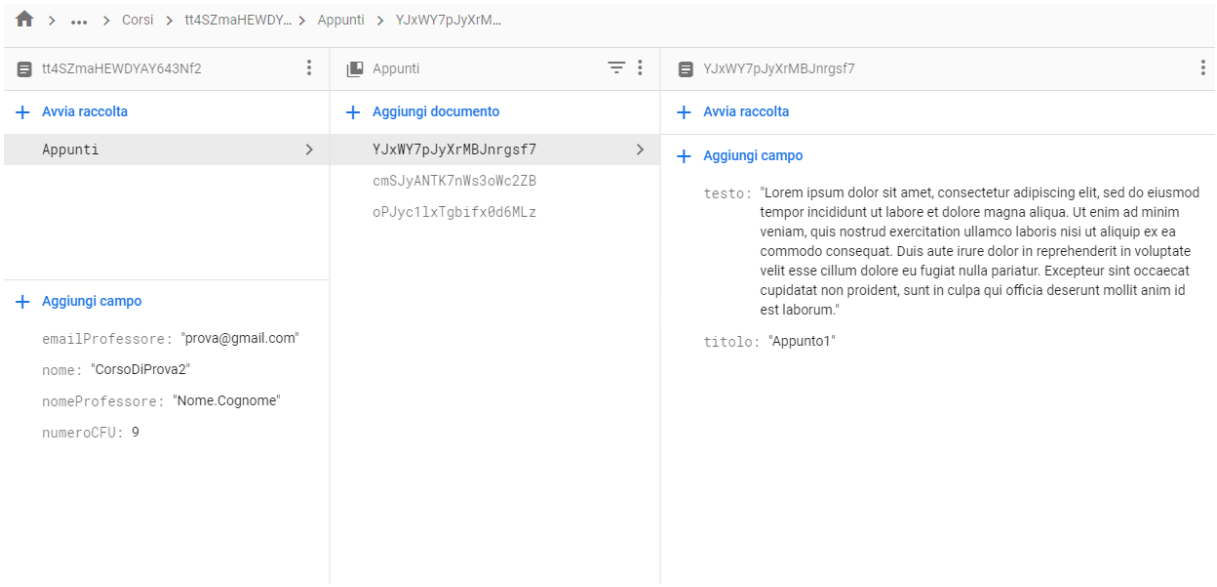

Figura 2.14: Collezione *Appunti*

Capitolo 3

Implementazione Android

L'app nativa per Android è stata sviluppata utilizzando l'IDE *Android Studio*, in modo da essere compatibile con Marshmallow e le versioni successive del sistema operativo Android (23 è infatti la versione minima dell'SDK impostata alla creazione del progetto).

3.1 Struttura del progetto in Android Studio

La cartella di progetto in Android Studio, ha la struttura riportata in Figura 3.1.

Si noti come i diversi elementi del progetto sono raggruppati in cinque differenti packages:

- *activities*: package che contiene tutte le activity dell'applicazione.
- *dialogs*: package che contiene i dialogs dell'applicazione, utilizzati principalmente per la visualizzazione e la modifica dei singoli elementi delle diverse liste.
- *entities*: package che contiene i modelli dei dati rappresentanti le entità della nostra applicazione.
- *fragments*: package che contiene i fragment utilizzati principalmente per la visualizzazione delle differenti liste.
- *uiutilities*: package che contiene le classi di utilità e di servizio alla nostra applicazione. In particolare, in questo package troviamo le classi relative agli *Adapter*, utilizzati per il popolamento delle varie liste dell'applicazione.

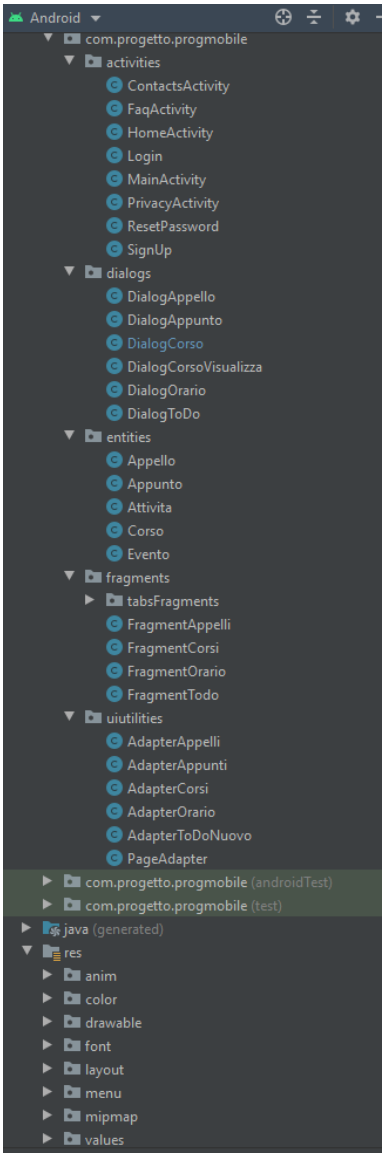

Figura 3.1: Struttura porgetto in *Android Studio*

Per quanto riguarda, invece, la cartella *Res*, in quest'ultima troviamo i file relativi alle risorse della nostra applicazione. In particolare, tale cartella contiene la cartella *layout* che, a sua volta, contiene i file di layout delle activities e dei fragments dell'applicazione. Inoltre, all'interno della cartella *Res* troviamo anche la cartella *menu*, che contiene i file relativi ai layout dei due menù presenti all'interno dell'applicazione, ovvero l'*Overflow menu* e la *BottomNavigationView*.

3.2 Componenti

3.2.1 Le Activity

Analizziamo, ora, una per una, le activity che compongono la nostra applicazione, illustrando, per ciascuna di esse, anche lo screen relativo alla visualizzazione.

Tali activity sono:

- *Splash Activity*: è la schermata di passaggio (*Splash screen*) che viene visualizzata per pochi secondi durante l'avvio (Figura 3.2).
- *SignUp Activity*: activity d'avvio dell'app (Figura 3.3). Permette ad un nuovo utente di registrarsi immettendo i dati richiesti. Abbiamo scelto tale activity come activity di avvio (piuttosto che quella di login) poiché, avendo un bacino di utenti ristretto, è molto più probabile che ad aprire l'app dopo averla scaricata sia un nuovo utente piuttosto che un utente già registrato.
- *Login Activity*: activity che permette di effettuare il login immettendo username e password (Figura 3.4).
- *ResetPassword Activity*: activity che permette all'utente, nel caso in cui questo abbia dimenticato la propria password, di procedere attraverso una procedura di *Reset*, completamente gestita da Firebase (Figura 3.5).
- *Home Activity*: activity che implementa una *BottomNavigationView* nella parte bassa dello schermo, utilizzata come menù principale dell'app. Tale menù permette di scorrere tra le quattro sezioni dell'applicazione (*ToDo*, *Corsi*, *Orario* e *Appelli*). Abbiamo optato per un menù di questo tipo piuttosto che per l'utilizzo di un'intera schermata come menu principale, poiché una *BottomNavigationView* facilita l'utilizzo dell'app e permette di passare più rapidamente da una sezione ad un altra, invece di dover passare da un menu principale. Il file di layout di questa activity presenta un placeholder che verrà popolato con i diversi fragment della nostra applicazione, come vedremo in seguito.
- *Contacts Activity*: activity che mostra come contattare i membri del gruppo (Figura 3.6).

- *Faq Activity*: activity che mostra le domande più frequenti degli utenti (Figura 3.7).
- *Privacy Activity*: activity che mostra le condizioni relative al trattamento dei dati personali degli utenti (Figura 3.8).

Figura 3.2: Splash Activity

Figura 3.3: SignUp Activity

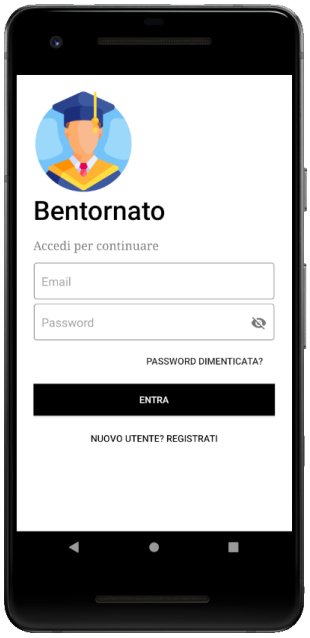

Figura 3.4: Login Activity

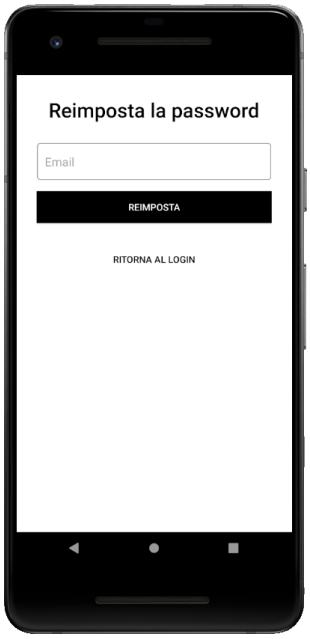

Figura 3.5: Reset Password

Figura 3.6: Contatti

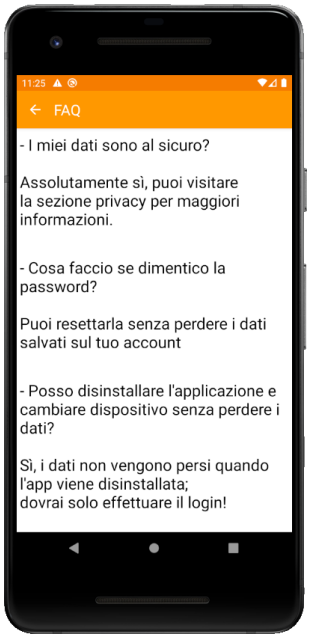

Figura 3.7: FAQ

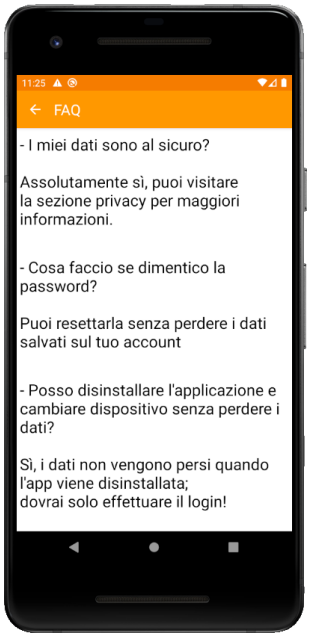

Figura 3.8: Privacy Activity

3.2.2 Fragments

Una parte molto importante della nostra applicazione sono i fragment. Come precedentemente detto, la *BottomNavigationView* presente nel layout della pagina principale dell'applicazione, permette di navigare tra i vari fragment, ciascuno dei quali contiene una *RecyclerView* per realizzare le diverse liste previste dalla nostra applicazione.

Analizziamo, ora, i singoli fragment:

- *Fragment ToDo*: è il fragment contenente la lista di attività *To do* da svolgere (Figura 3.9). In particolare viene mostrato il nome dell'attività, l'inizio della descrizione, un'immagine che rappresenta la priorità e la data di scadenza.
- *Fragment Corsi*: è il fragment contenente la lista di corsi inseriti dall'utente (Figura 3.10). Vengono mostrati solamente il nome del corso ed il numero di CFU.
- *Fragment Orario*: è il fragment che contiene l'orario settimanale dell'utente (Figura 3.11). Questo in particolare è un *tabbed* fragment e permette di navigare attraverso i diversi giorni (Ognuno con la lista di eventi contenuta in un apposito fragment).
- *Fragment Appelli*: è il fragment contenente la lista degli appelli inseriti dallo studente (Figura 3.12). Per ogni elemento si mostra la materia e la data dell'esame.

É importante ricordare come i layout dei signoli elementi delle liste presenti nei fragment dell'applicazione vengano definiti all'interno della directory *Res*, in particolare all'interno della cartella *layout*.

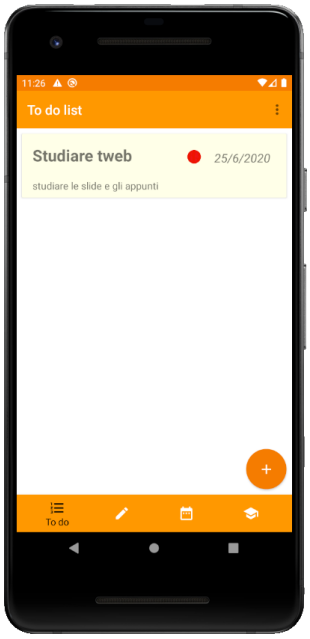

Figura 3.9: Fragment ToDo

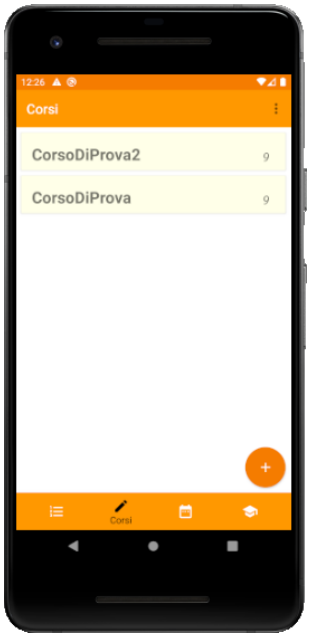

Figura 3.10: Fragment Corsi

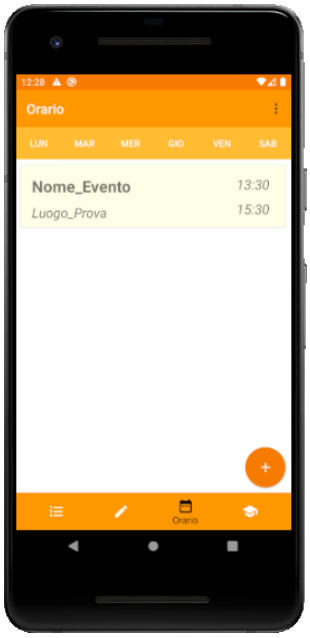

Figura 3.11: Fragment Orario

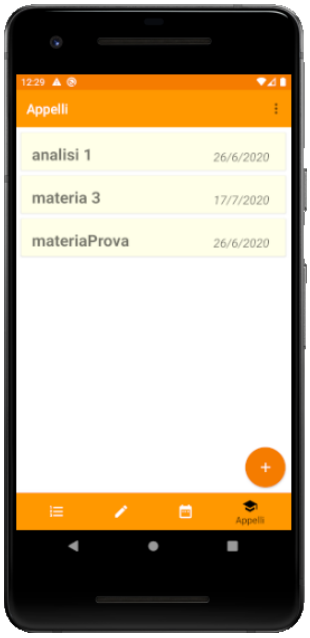

Figura 3.12: Fragment Appelli

3.2.3 Dialogs

Abbiamo utilizzato una serie di schermate che permettono all'utente di inserire le informazioni da salvare e visualizzare per ciascuna delle liste presenti all'interno dell'applicazione.

Per realizzare queste funzionalità abbiamo optato per l'utilizzo dell'elemento *Dialog*, in particolare abbiamo utilizzato i *DialogFragment*, che sono dei dialog a schermo intero. Questo ci ha permesso di adattarli all'inserimento dei dati per ogni sezione.

I dialogs realizzati sono i seguenti:

- *Dialog ToDo*: dialog che permette l'inserimento o la modifica di un elemento della categoria *Attività* (Figura 3.13). Nel dialog verranno richiesti all'utente il nome, la priorità (bassa, media o alta), la scadenza (attraverso un *date-picker*) e una descrizione.

- *Dialog Corso*: dialog che permette l'inserimento o la modifica di un elemento della categoria *Corso*(Figura 3.14). Nel dialog verranno richiesti all'utente il nome del corso, quello del professore, l'email del professore ed il numero di CFU.

- *Dialog Evento*: dialog che permette l'inserimento o la modifica di un elemento della sezione *Orario*(Figura 3.15). Nel dialog verranno richiesti all'utente il nome dell'evento, il luogo, l'ora di inizio e fine (attraverso un *time-picker*) ed il giorno della settimana.

- *Dialog Appello*: dialog che permette l'inserimento o la modifica di un elemento della categoria *Appello* (Figura 3.16). Nel dialog verranno richiesti all'utente la materia e la data dell'appello (attraverso un *date-picker*).

- *Dialog Appunto*: dialog che permette l'inserimento o la modifica di un elemento della categoria *Appunti* all'interno di un *Corso* (Figura 3.17). Nel dialog verranno richiesti all'utente il titolo ed il testo dell'appunto.

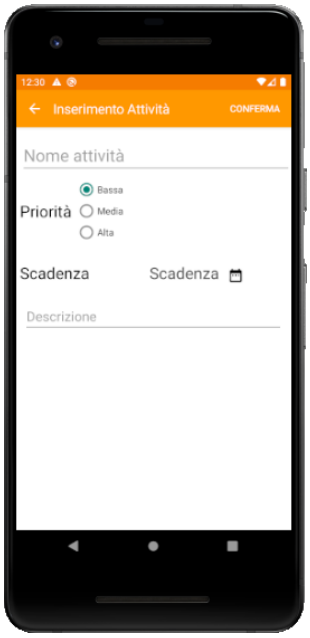

Figura 3.13: Dialog ToDo

Figura 3.14: Dialog Corso

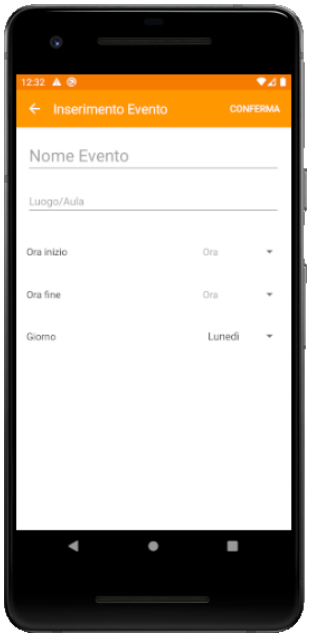

Figura 3.15: Dialog Evento

Figura 3.16: Dialog Appello

Figura 3.17: Dialog Appunto

3.3 Particolari soluzioni adottate

In questa sezione, descriviamo le soluzioni più interessanti adottate per la realizzazione dell'applicazione.

3.3.1 Libreria FirebaseUI

Un elemento fondamentale della nostra app sono le liste (*RecyclerView*), che permettono di realizzare tutte le funzionalità principali dell'applicazione stessa, quali la visualizzazione dei *Corsi*, delle *ToDo*, dell'*Orario* e degli *Appelli*.

Per quanto riguarda il popolamento delle viste, è stato utilizzato l'interfacciamento al database cloud Firebase Firestore, configurato secondo la struttura precedentemente illustrata.

In particolare, per facilitare la procedura di *Binding* tra le nostre liste e la base di dati, è stato utilizzato un plugin open source la cui documentazione è reperibile al sito `https://firebaseopensource.com/projects/firebase/firebaseui-android/`. Si tratta della libreria *FirebaseUI*, che consente una semplice connessione tra elementi della UI e le API di Firebase.

Tale libreria, in realtà, è composta da moduli separati per utilizzare le varie funzionalità di Firebase. Tali moduli sono *FirebaseUI Auth*, *FirebaseUI Firestore*, *FirebaseUI Database* e *FirebaseUI Storage*.

Nel nostro caso, è stata utilizzata la sola componente *FirebaseUI Firestore*. Per importare tale libreria è stata aggiunta al file `app/build.gradle` la dipendenza `implementation 'com.firebaseui:firebase-ui-firestore:6.2.1'`.

Successivamente, sono state seguite le linee guida riportate nella documentazione uffciale della libreria per costruire i nostri *Adapter*.

Il Listato 3.1 mostra il codice dell'*Adapter* relativo alla lista degli *Appelli*. Tale *Adapter*, come si può vedere dal listato, estende la classe *FirestoreRecyclerOptions*, fornita proprio dal plugin FirestoreUI.

Listing 3.1: Esempio di Adapter realizzato con FirestoreUI

```
1 package com.progetto.progmobile.uiutilities;
2
```

```
3  import android.view.LayoutInflater;
4  import android.view.View;
5  import android.view.ViewGroup;
6  import android.widget.TextView;
7
8  import androidx.annotation.NonNull;
9  import androidx.recyclerview.widget.RecyclerView;
10
11 import com.firebase.ui.firestore.FirestoreRecyclerAdapter;
12 import com.firebase.ui.firestore.FirestoreRecyclerOptions;
13 import com.google.firebase.firestore.DocumentSnapshot;
14 import com.progetto.progmobile.R;
15 import com.progetto.progmobile.entities.Appello;
16
17 public class AdapterAppelli extends FirestoreRecyclerAdapter<Appello,
       AdapterAppelli.AppelloHolder> {
18
19     private OnItemClickListener listener;
20
21     public AdapterAppelli(@NonNull FirestoreRecyclerOptions<Appello>
           options) {
22         super(options);
23     }
24     @Override
25     protected void onBindViewHolder(@NonNull AppelloHolder holder,
           int position, @NonNull Appello model) {
26         holder.textMateria.setText(model.getMateria());
27         holder.textData.setText(new StringBuilder().append(model.
               getGiorno())
28                 .append("/").append(model.getMese()).append("/").
                       append(model.getAnno()).toString());
29
```

```
30      }
31
32      @NonNull
33      @Override
34      public AppelloHolder onCreateViewHolder(@NonNull ViewGroup parent
            , int viewType) {
35          View v = LayoutInflater.
36                  from(parent.getContext()).inflate(R.layout.
                        riga_appello, parent, false);
37          return new AppelloHolder(v);
38      }
39
40      public void deleteItem(int position) {
41
42          getSnapshots().getSnapshot(position).getReference().delete();
43      }
44      class AppelloHolder extends RecyclerView.ViewHolder {
45          TextView textMateria, textData;
46
47          public AppelloHolder(@NonNull View itemView) {
48              super(itemView);
49              textMateria = itemView.findViewById(R.id.textMateria);
50              textData = itemView.findViewById(R.id.textData);
51
52              itemView.setOnClickListener(new View.OnClickListener() {
53                  @Override
54                  public void onClick(View v) {
55                      int position = getAdapterPosition();
56                      if (position != RecyclerView.NO_POSITION &&
                            listener != null) {
57                          listener.onItemClick(getSnapshots().
                                getSnapshot(position), position);
```

```
58                          }
59
60                      }
61                  });
62              }
63      }
64
65      public interface OnItemClickListener {
66          void onItemClick(DocumentSnapshot documentSnapshot , int
                position);
67
68      }
69
70      public void setOnItemClickListener(OnItemClickListener listener)
            {
71          this.listener = listener;
72      }
73
74  }
```

Capitolo 4

Implementazione Xamarin

Nella seconda parte del progetto la stessa app è stata realizzata mediante il framework *Xamarin* utilizzando, per quanto riguarda la condivisione del codice, l'approccio *.NET* standard (basato su PLC).

L'obiettivo, quindi, è stato quello di sviluppare un'unica app in grado di essere eseguita su sistemi operativi diversi (*Android* e *iOS*), minimizzando il codice specifico per le singole piattaforme. In quest'ottica è stata, però, implementata soltanto la versione *Android* dell'app, non potendo testare il codice per *IOS*.

L'applicazione realizzata in Xamarin implementa le medesime funzionalità dell'applicazione nativa per Android realizzata con Android Studio.

4.0.1 Struttura del progetto in Visual Studio

Per la realizzazione del progetto è stato utilizzato l'IDE Visual Studio 2019, con l'utilizzo del framework Xamarin, integrato perfettamente all'interno dell'IDE stesso.

La cartella del progetto in Xamarin ha la struttura riportata in Figura 4.1.

Facendo riferimento alla cartella *ProgettoEsame*, che contiene la parte di codice condivisa del progetto, abbiamo deciso di organizzare i file al suo interno contenuti mediante l'utilizzo delle seguenti cartelle:

- *Model*: questa cartella contiene tutti i file delle classi in cui sono definiti i modelli dei dati della nostra applicazione.

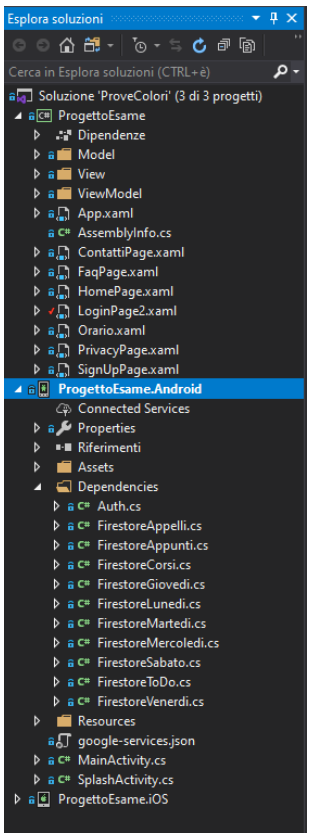

Figura 4.1: Struttura Progetto Xamarin

- *View*: questa cartella contene i file delle viste della nostra applicazione, i cui layout sono specificati nei file `.xaml`. È importante sottolineare come il comportamento associato ai file di tali viste non sia implementato nel file `.cs`.

- *ViewModel*: questa cartella contiene la parte di codice che si occupa del popolamento degli elementi delle viste definite nella cartella *View*. Inoltre si occupa anche della gestione del comportamento delle viste stesse.

Consideriamo, ora, la cartella *ProgettoEsame.Android*, contenente le parti di codice specifiche per il sistema operativo Android.

Vogliamo far notare come, all'interno di tale cartella, sia stata creata una nuova cartella di nome *Dependencies*, la quale contiene i file delle classi specifiche di Android che consentono l'interfacciamento a Firebase. Infatti, in Xamarin.Forms, non è previsto un pieno supporto al servizio fornito da Firebase. Per realizzare le funzionalità di quest'ultimo, infatti, è necessario implementare i metodi di accesso specificatamente sia per Android che per IOS. Nelle sezioni

successive illustreremo il meccanismo con cui è stato possibile utilizzare Firebase all'interno della nostra applicazione Xamarin.

Nella cartella *ProgettoEsame.Android*, come si può facilmente notare dalla Figura 4.1, sono presenti, inoltre, due file `.cs` relativi a due activity, ovvero *MainActivity* e *SplashActivity*. In *SplashActivity* è definita la schermata di caricamento dell'applicazione. Tale activity è stata definita appositamente per la soluzione specifica di Android ed è particolarmente utile in quanto permette di coprire il tempo di caricamento della libreria Xamarin.Forms durante l'avvio dell'app. Infatti tale schermata rimane attiva fino a quando l'applicazione è pronta per essere eseguita; in tale momento il controllo passa alla *MainActivity* che si occupa del caricamento della classe *App* delle soluzione comune del progetto.

4.1 Le pagine principali dell'applicazione

Le pagine realizzate nell'applicazione Xamarin sono le stesse di quella per Android.

Le pagine relative alle liste, sono state definite all'interno della cartella *View* della soluzione comune, mentre le pagine statiche non sono state inserite in nessuna cartella, infatti sono contenute direttamente all'interno della soluzione cross-platform *ProgettoEsame*.

In particolare, sono state realizzate quattro pagine principali, raggiungibili attraverso la barra di navigazione definita nella pagina *HomePage*. Tali pagine sono le stesse definite anche nel progetto Android, cioè *Corsi*, *ToDo*, *Orario* e *Appelli*, le quali mostrano, ciascuna, le liste corrispondenti.

Inoltre, sono state realizzate anche le pagine che consentono l'inserimento e la modifica dei singoli elementi delle liste, esattamente come effettuato in Android Studio.

Riportiamo, ora, le pagine principali dell'applicazione realizzata in Xamarin.

Tali pagine sono:

- Pagina di registrazione (Figura 4.2);
- Pagina di login (Figura 4.3);
- Pagina relativa alla lista delle attività da svolgere (Figura 4.4);

- Pagina relativa alla lista dei corsi (Figura 4.5);
- Pagina che visualizza l'orario dei singoli giorni della settimana (Figura 4.6);
- Pagina relativa alla lista degli appelli (Figura 4.7);
- Pagina relativa alla lista degli appunti (Figura 4.8);
- Pagina che consente l'inserimento di una nuova attività (Figura 4.9);
- Pagina relativa alla modifica di un'attività (Figura 4.10);
- Pagina relativa all'inserimento di un nuovo corso (Figura 4.11);
- Pagina relativa alla visualizzazione e alla modifica di un appunto (Figura 4.12);
- Pagina relativa all'inserimento di un nuovo evento nell'orario (Figura 4.13);
- Pagina relativa all'inserimento di un nuovo appello (Figura 4.14);

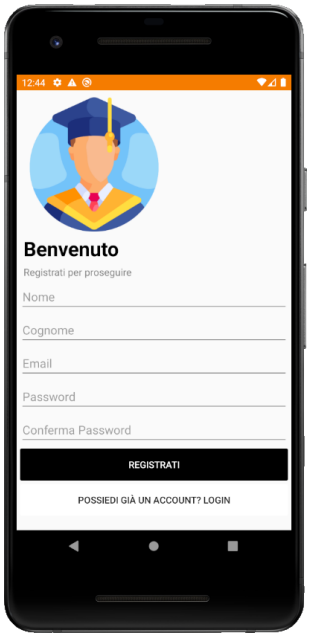

Figura 4.2: Pagina di SignUp

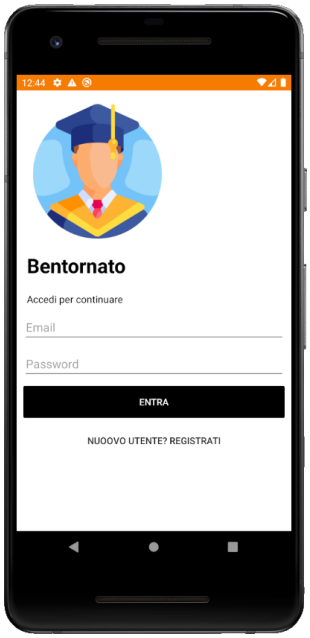

Figura 4.3: Login

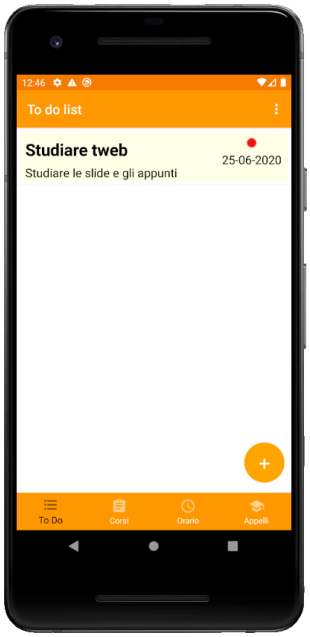

Figura 4.4: Pagina con la lista delle To Do

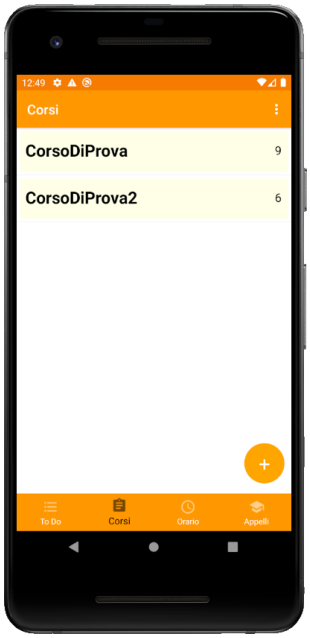

Figura 4.5: Pagina con la lista dei Corsi

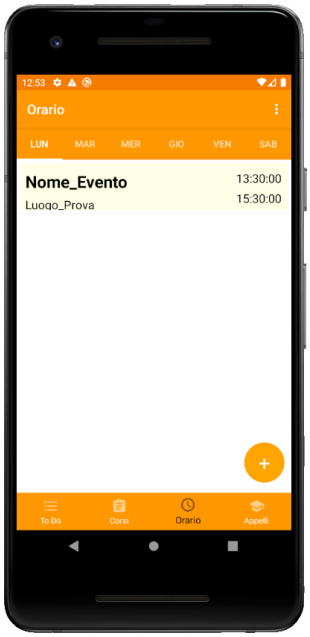

Figura 4.6: Pagina relativa all'Orario

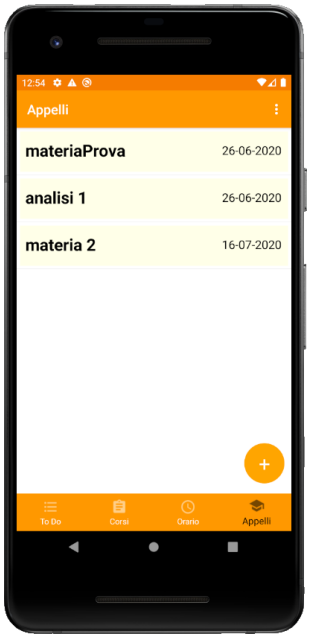

Figura 4.7: Pagina con la lista degli Appelli

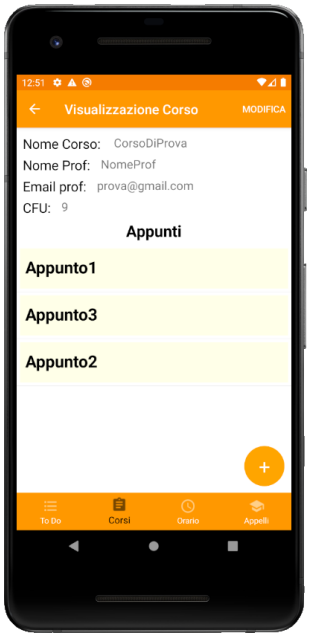

Figura 4.8: Pagina con la lista degli Appunti

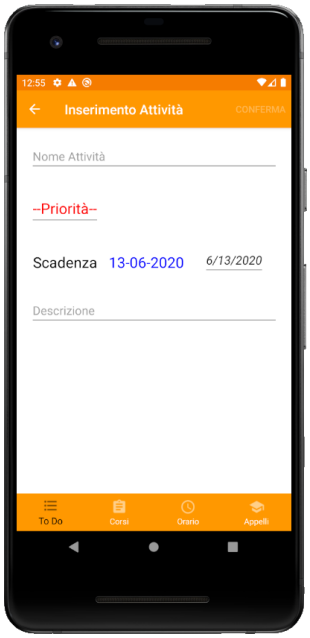

Figura 4.9: Pagina di inserimento attività

Figura 4.10: Pagina di modifica attività

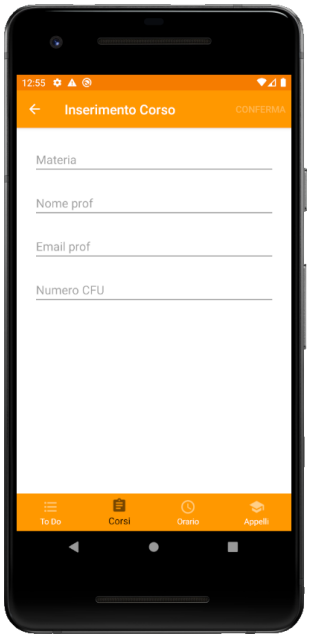

Figura 4.11: Pagina di inserimento corso

Figura 4.12: Pagina di visualizzazione appunto

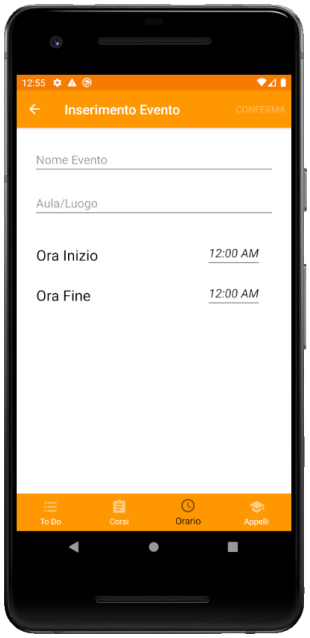

Figura 4.13: Pagina di inserimento evento

Figura 4.14: Pagina di inserimento appello

4.2 Implementazione del pattern MVVM

Xamarin utilizza una propria versione del pattern *MVC* (Model, View, Controller) chiamata *MVVM* (Model, View, View-Model). All'interno dell'applicazione, è stato fatto un uso intenso di tale pratica di programmazione, in particolare all'interno delle pagine che gestiscono le liste. Invece, negli altri casi abbiamo preferito adottare un approccio standard come quello seguito nel caso dell'applicazione nativa per Android.

Questo design pattern permette di sudddividere la struttura dell'intera applicaizone in tre moduli funzionali separati, ma comunicanti tra di loro. Tali moduli sono, appunto, il *Model*, il *ViewModel* e il *View*.

Come anticipato nella descrizione della struttura del progetto, i diversi elementi del pattern sono stati sviluppati nelle apposite cartelle.

Inoltre, Xamarin, a differenza di Android, permette di poter utilizzare il concetto di *Binding*, una funzionalità molto potente che permette di aggiornare la vista automaticamente al variare del model di riferimento o in risposta a determinate eventi. Il *Binding* è perfettamente integrato nel pattern *MVVM* e costituisce uno degli elementi cardine di quest'ultimo.

Questo design architetturale è stato particolarmente utile per il nostro caso, infatti, esso permette di semplificare significativamente la gestione delle viste, rispetto a quanto visto in Android.

Le differenze tra le due applicazioni da noi realizzate, infatti, riguardano, sostanzialmente, il modo con cui sono state realizzate le liste. In particolare:

- Per Android sono state utilizzate le *RecyclerView*, le quali utilizzano un layout per ogni oggetto, un *LayoutManager* per visualizzare gli oggetti nel modo corretto e un *RecyclerViewAdapter*, per collegare il model alla vista e gestire l'interazione con l'utente.

- Per Xamarin sono state utilizzate, invece, le *ListView*, che necessitano di una vista template per generare ogni item e un binding con un *ObservableCollection* per collegare i dati con la vista stessa.

Di conseguenza l'utilizzo delle liste in Xamarin è risultato più semplice rispetto ad Android, per via della minor quantità di codice da implementare, grazie proprio all'impiego del pattern

MVVM a al concetto di *binding*.

4.3 Interfacciamento a Firebase

Per poter usufruire delle funzionalità di Firebase, sono state utilizzate delle librerie apposite, importate mediante i paccetti *NuGet* nella soluzione specifica per Android. Tali librerie sono:

- *Xamarin.Firebase.Auth*: libreria per effettuare le procedure di login e registrazione messe a disposizione da Firebase.
- *Xamarin.Firebase.Firestore*: libreria per effettuare query sul database Firestore.
- *Xamarin.Firebase.Common*: librerie di supporto alle precedenti, necessaria per poter utilizzare il servizio Google di Firebase.

In particolare, sono state realizzate delle interfacce nella soluzione comune del progetto, le cui funzioni sono state implementate da classi definite nella soluzione specifica per Android. È stato, cioè, utilizzato il concetto di *DependencyService*, che consente di recuperare le implementazioni delle piattaforme specifiche per iniettarle nella soluzione comune.

Il Listato 4.1 mostra l'interfaccia che definisce le funzioni di interfacciamento a Firestore per quanto riguarda le operazioni CRUD relative agli *Appelli*.

Interfacce analoghe sono state realizzate anche per le altre liste della nostra applicazione e sono state inserite all'interno della cartella *Helpers* contenuta in *ViewModel*.

Listing 4.1: Esempio di interfaccia verso Firestore

```
77 using ProgettoEsame.Model;
78 using System;
79 using System.Collections.Generic;
80 using System.Text;
81 using System.Threading.Tasks;
82 using Xamarin.Forms;
83
84 namespace ProgettoEsame.ViewModel.Helpers
```

```
85  {
86      public interface AppelliFirestore
87      {
88          bool InsertAppello(Appello appello);
89          Task<bool> DeleteAppello(Appello appello);
90          Task<bool> UpdateAppello(Appello appello);
91          Task<IList<Appello>> ReadAppelli();
92      }
93
94      public class DatabaseAppelliHelper
95      {
96          private static AppelliFirestore firestore = DependencyService
                .Get<AppelliFirestore>();
97
98          public static Task<bool> DeleteAppello(Appello appello)
99          {
100             return firestore.DeleteAppello(appello);
101         }
102
103         public static bool InsertAppello(Appello appello)
104         {
105             return firestore.InsertAppello(appello);
106         }
107
108         public static Task<IList<Appello>> ReadAppelli()
109         {
110             return firestore.ReadAppelli();
111         }
112
113         public static Task<bool> UpdateAppello(Appello appello)
114         {
115             return firestore.UpdateAppello(appello);
```

```
116             }
117
118         }
119 }
```

L'implementazione di questa interfaccia è riportata nel Listato 4.2, che riporta la classe *FirestoreAppelli* contenuta all'interno della cartella *Dependencies* della soluzione del progetto specifica per la piattaforma Android. Questa cartella, ovviamente, contiene anche le altre classi che implementano le interfacce a Firestore per le altre liste della nostra applicazione.

Listing 4.2: Esempio di implementazione dell'interfaccia verso Firestore

```
123 using System;
124 using System.Collections.Generic;
125 using System.Linq;
126 using System.Text;
127 using System.Threading.Tasks;
128 using Android.App;
129 using Android.Content;
130 using Android.OS;
131 using Android.Runtime;
132 using Android.Views;
133 using Android.Widget;
134 using Firebase.Auth;
135 using Google.Type;
136 using Java.Util;
137 using ProgettoEsame.Model;
138 using ProgettoEsame.ViewModel.Helpers;
139 using Xamarin.Forms;
140 using Firebase.Firestore;
141 using Android.Gms.Tasks;
142 using Android.Service.VR;
143 using Org.W3c.Dom;
```

```
144 [assembly: Dependency(typeof(ProgettoEsame.Droid.Dependencies.
        FirestoreAppelli))]
145 namespace ProgettoEsame.Droid.Dependencies
146 {
147     class FirestoreAppelli : Java.Lang.Object, ViewModel.Helpers.
            AppelliFirestore, IOnCompleteListener
148     {
149
150         List<Appello> appelliList;
151         bool hasReadAppelli = false;
152
153         public FirestoreAppelli()
154         {
155             appelliList = new List<Appello>();
156         }
157
158
159
160         public async Task<bool> DeleteAppello(Appello appello)
161         {
162             try
163             {
164                 var collection = Firebase.Firestore.FirebaseFirestore
                        .Instance.Collection("users")
165                         .Document(Firebase.Auth.FirebaseAuth.Instance.
                            CurrentUser.Uid).Collection("Appelli");
166                 collection.Document(appello.Id).Delete();
167                 return true;
168             }
169             catch (Exception ex)
170             {
171                 return false;
```

```
172             }
173         }
174
175         public bool InsertAppello(Appello appello)
176         {
177
178             try
179             {
180                 var collection = Firebase.Firestore.FirebaseFirestore
                        .Instance.Collection("users")
181                         .Document(Firebase.Auth.FirebaseAuth.Instance.
                            CurrentUser.Uid).Collection("Appelli");
182                 var appelloDocument = new Dictionary<string, Java.
                    Lang.Object>
183                 {
184                     {"name", appello.Name },
185                     {"date", appello.Date }
186                 };
187                 collection.Add(appelloDocument);
188
189                 return true;
190             }
191             catch (Exception ex)
192             {
193                 return false;
194             }
195
196         }
197
198         public async Task<IList<Appello>> ReadAppelli()
199         {
200             hasReadAppelli = false;
```

```
201             var collection = Firebase.Firestore.FirebaseFirestore.
                    Instance.Collection("users")
202                 .Document(Firebase.Auth.FirebaseAuth.Instance.
                        CurrentUser.Uid).Collection("Appelli");
203
204             collection.Get().AddOnCompleteListener(this);
205             for (int i = 0; i < 25; i++)
206             {
207                 await System.Threading.Tasks.Task.Delay(100);
208                 if (hasReadAppelli)
209                     break;
210             }
211
212             return appelliList;
213
214         }
215
216         public async Task<bool> UpdateAppello(Appello appello)
217         {
218             try
219             {
220                 var collection = Firebase.Firestore.FirebaseFirestore
                        .Instance.Collection("users")
221                     .Document(Firebase.Auth.FirebaseAuth.Instance.
                            CurrentUser.Uid).Collection("Appelli");
222                 collection.Document(appello.Id).Update("name",
                        appello.Name, "date", appello.Date);
223                 return true;
224             }
225             catch (Exception ex)
226             {
227                 return false;
```

```
228             }
229         }
230
231
232         public void OnComplete(Android.Gms.Tasks.Task task)
233         {
234             if (task.IsSuccessful)
235             {
236                 var documents = (QuerySnapshot)task.Result;
237
238
239                 appelliList.Clear();
240                 foreach (var doc in documents.Documents)
241                 {
242                     string date;
243                     if (doc.Get("date") == null)
244                     {
245                         date = "";
246                     }
247                     else
248                     {
249                         date = doc.Get("date").ToString();
250                     }
251                     Appello appello = new Appello
252                     {
253                         Name = doc.Get("name").ToString(),
254                         Date = date,
255                         Id = doc.Id
256                     };
257
258                     appelliList.Add(appello);
259
```

```
260                     }
261                 }
262                 else
263                 {
264                     appelliList.Clear();
265 
266                 }
267                 hasReadAppelli = true;
268             }
269         }
270 }
```

Capitolo 5

Considerazioni finali

Si può concludere, al termine del progetto, che la produzione di un'applicazione è, come molti altri lavori nell'ambito dell'informatica, un processo nel quale la progettazione riveste un ruolo fondamentale, forse più importante della realizzazione effettiva. I diversi software utilizzati offrono funzionalità ed approcci differenti e, quindi, la scelta di quale utilizzare, dovrebbe esser fatta in relazione all'obiettivo che si vuole raggiungere. Nel complesso il progetto è stato per noi un'attività interessante oltre che formativa e ci ha permesso di entrare nel mondo della programmazione mobile.

5.1 Comparazione dei due approcci

La realizzazione del progetto in due versioni differenti, una nativa per Android ed una ibrida, ci ha permesso di confrontarci con i diversi approcci di sviluppo di applicazioni mobile.

Grazie a questo progetto, abbiamo avuto la possibilità di rafforzare le nostre conoscenze riaguardo al mondo Java, e di imparare un nuovo linguaggio, molto noto ed utilizzato come il C#. Abbiamo notato come il passaggio a C# sia stato piuttosto semplice, essendo questo un linguaggio sintatticamente e concettualmente molto simile al C e a Java stesso.

Un aspetto che ci interessa sottolineare è la differenza tra i due ambienti di sviluppo utilizzati, cioè Android Studio e Visual Studio. Dal nostro punto di vista, Android Studio risulta molto più semplice e di immediata comprensione rispetto al secondo. Inoltre, abbiamo notato anche una diversa fluidità nell'utilizzo dei due IDE, con Android Studio decisamente più

reattivo. Nonostante ciò, Visual Studio rimane, comunque, un IDE decisamente valido per lo sviluppo di applicazioni ibride; a tal proposito, ci siamo trovati bene con la realizzazione dei file di layout mediante il linguaggio di marckup *.xaml* e anche con l'utilizzo degli items di layout messi a disposizione da Xamarin.Forms.

Infine, vogliamo aggiungere che, sebbene le due app siano graficamente e funzionalmente molto simili, l'app Android risulta, come previsto, sicuramente più fluida e veloce rispetto a quella ottenuta utilizzando Xamarin. Queste differenze le abbiamo notate soprattutto nella fase di avvio dell'app, nel salvataggio delle *Properties*, ma anche nel passaggio tra le diverse pagine dell'applicazione.

5.2 Conclusioni e risultato finale

Possiamo dirci soddisfatti con entrambe le applicazioni che abbiamo prodotto, poiché siamo riusciti, sebbene con lievissime differenze, a raggiungere un ottimo livello di fedeltà rispetto ai mockup realizzati in fase di progettazione, ma siamo anche riusciti a soddisfare i requisiti funzionali dell'app che ci eravamo prefissati di raggiungere.

5.3 Repository GitHub

5.3.1 Repository dei progetti

Riportiamo in seguito i link ai due repository, contententi la realizzaizone del progetto, rispettivamente di Android e Xamarin:

- `https://github.com/denaldo98/progettoXamarinUltimo`
- `https://github.com/denaldo98/ProgMobile3`

5.3.2 Repository dei singoli membri del gruppo

- `https://github.com/denaldo98`
- `https://github.com/S1082351`

- https://github.com/MarcoMonini