UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche, Informatiche e Matematiche
Corso di Laurea Triennale in Scienze dell'Informazione

Progetto e Sviluppo di una Piattaforma Social per l'Incontro tra Domanda e Offerta di Ripetizioni Private

Cavazza Alessia

Tesi di Laurea

Relatore:
Prof. Riccardo Martoglia

Anno Accademico 2018/2019

PAROLE CHIAVE:
applicazioni mobile,
Firebase,
NoSQL database,
Android studio,
Geofire

Indice

Indice Figure

Introduzione

L'interesse per lo sviluppo di un'applicazione che vada a semplificare la ricerca di ripetizioni private, e l'interazione tra studente e professore di ripetizioni, nasce da un bisogno personale che da diversi anni mi appartiene sia come studente che come professore.

Infatti questa applicazione ha lo scopo di creare uno strumento unico di comunicazione per tutte quelle persone che ad oggi devono organizzarsi con siti, bacheche fisiche e online, o altro, per postare i propri annunci.

Durante il periodo di tirocinio interno all'Università quindi, sono state sviluppate diverse funzionalità dell'app, che ho ritenuto di priorità per lo scopo che mi ero prefissata, mentre altre sono state lasciate a sviluppi futuri, come spiegato nell'ultima sezione di questa tesi.

Per l'implementazione dell'app è stato utilizzato Android Studio come IDE e Firebase per gestire i dati legati all'applicazione.

Per quanto riguarda la struttura di questa tesi, nel primo capitolo verrà spiegato più in dettaglio il "caso di studio", che già parzialmente è stato descritto sopra.

In seguito si proporranno le tecnologie utilizzate per lo sviluppo vero e proprio dell'applicazione. In questo capitolo quindi verranno presentati Android Studio, Firebase, Geofire, Node.js e Picasso.

Dopodiché, nel penultimo capitolo verrà mostrata la progettazione a cui segue l'implementazione dell'applicazione. Nell'implementazione vengono mostrate tutte le schermate con cui l'utente può interagire e il codice relativo ad esse.

Infine, come già detto, nell'ultimo capitolo, vi è un'ultima riflessione su quanto è stato fatto delle funzionalità pensate/progettate per l'applicazione e quanto invece è stato lasciato a sviluppi futuri.

1 Il Caso di Studio

La scelta del caso di studio è dovuta essenzialmente ad esperienze personali che hanno portato all'idea dello sviluppo di un'applicazione per l'incrocio tra domanda e offerta di lezioni di ripetizioni private. Infatti, si è voluta creare un'applicazione che semplificasse la ricerca a studenti e professori, poiché, nel personale, ci si è trovati spesso, sia come insegnante di ripetizioni che come studente in cerca di ripetizioni, in difficoltà nel trovare qualcuno nelle vicinanze e in tempi ragionevoli.
Inoltre si è voluto chiedere un parere anche ad altri coetanei, i quali si sono trovati d'accordo sull'utilità di una simile proposta, in quanto ad oggi non risulta esserci ancora in commercio un'applicazione mobile che affronti questi temi. L'unico modo infatti per un insegnante di ripetizioni di postare annunci è di utilizzare le classiche bacheche scolastiche o i siti di annunci online, che però risultano ostici e meno trasparenti di un'applicazione mobile. Questo infatti è il riscontro che si è avuto da persone che sono state intervistate e che si prestano come insegnanti di ripetizioni, o come studenti. Per quanto riguarda le tecnologie già esistenti che possano sopperire a questo problema, sono stati rilevati, come già detto, i siti di annunci online, di cui alcuni presentano anche la versione mobile (tra questi ad esempio Kijiji, Subito.it). Lo scopo di tali siti però è quello di pubblicare annunci di qualsiasi tipo e, mentre in alcuni di questi è abbastanza facile trovare la sezione riguardante gli annunci di ripetizioni, in altri risulta molto più difficoltoso. Inoltre non sempre chi si trova ad inserire annunci è avvezzo all'uso del computer, mentre invece le applicazioni mobili, com'è noto, per semplicità e chiarezza

dell'interfaccia, risultano molto più adatte.
Oltre ai siti si è scoperta l'esistenza di applicazioni simili ma non disponibili in italiano (ad esempio Teach Me Now), quindi non utilizzabili sul nostro territorio.
Ricapitolando quindi, l'attenzione verso questo caso di studio parte da un'esperienza personale, da una mancanza di mezzi e strumenti per raggiungere un obiettivo, e ciò si è riscontrato in altre persone che si prestano come studenti o professori.
Per quanto riguarda invece l'applicazione vera e propria si è voluto dar modo all'utente di poter trovare o postare annunci per ripetizioni in luoghi da lui selezionati o dalla posizione in cui si trova quando effettua la ricerca. Inoltre, dopo che un utente ha trovato un professore o uno studente che meglio risponde ai suoi bisogni, gli è stata data la possibilità di contattare tale figura per mezzo di una funzionalità di messaggistica di cui è dotata l'app. Questo infatti era uno dei requisiti per lo sviluppo dell'app, perché ci si voleva discostare dalle piattaforme di annuncio attuali che non consentono questa funzionalità di social, che è sembrata invece estremamente importante perché rende molto più professionale il colloquio tra studente-professore, e permette a colui che posta l'annuncio di evitare di inserire direttamente nell'annuncio il proprio numero di telefono.

2 Tecnologie e Ambienti utilizzati

2.1 Android Studio

2.1.1 Introduzione ad Android Studio

Per lo sviluppo dell'applicazione si è scelto di utilizzare Android Studio per la vastità di materiale e documentazione disponibile in internet. Inoltre nelle ricerche effettuate, per la valutazione delle tecnologie da utilizzare, Android Studio è risultato quello più adatto per la semplicità e la chiarezza di utilizzo dell'interfaccia, nonché lo strumento più utilizzato a livello mondiale per lo sviluppo di applicazioni Android.

Android Studio è un ambiente di sviluppo integrato (IDE) basato sul software di JetBrains IntelliJ IDEA per lo sviluppo di applicazioni Android. Le versioni di Android Studio sono compatibili con Linux, Windows e Machintosh Completamento avanzato del codice, refactoring, e analisi rendono lo sviluppo dell'applicazione più facile e veloce per lo sviluppatore. Inoltre essendo scaturito dalla stessa Google ed essendo nato appositamente per Android, è possibile realizzare applicazioni per dispositivi mobili con sistema operativo Android che dialoghino facilmente con gli svariati servizi di cloud forniti da Google. [1]

Altre caratteristiche messe a disposizione da Android Studio sono:

- editor per layout visuale utilizzabile in modalità drag and drop;
- accesso a SDK Manager e AVD Manager;
- inline debugging;
- monitoraggio delle risorse di memoria e della CPU utilizzate dall'app.

Dalla versione 1.3, si è avuta una ricongiunzione tra SDK (per lo sviluppo in Java) ed NDK (il Native Developemente Kit, pensato per sviluppatori C/C++) per il quale è stato aggiunto l'editing e il debugging in Android Studio. Nella seconda major release, invece, si è puntato all'incremento della produttività e alla riduzione dei tempi di sviluppo con la nascita dell'Instant Run, che mira a vedere in azione con grande rapidità le modifiche apportate al codice senza aspettare lunghi rebuild del progetto, e al potenziamento degli emulatori.

2.1.2 Android SDK

L'SDK(Software Development Kit) è un insieme di strumenti utili allo sviluppo e alla documentazione del software, dotato di librerie, documentazioni, compilatore e licenze. Questo SDK è costituito da molti strumenti – programmi, emulatori, piattaforme per ogni versione di Android e molto altro – la cui composizione non è immutabile ma viene gestita tramite il programma Android SDK Manager. L'SDK specifico concepito per Android è compreso nell'installazione di Android Studio. Tra i tools più importanti messi a disposizione dall'SDK ci sono:

- Emulator.exe: lancia un emulatore Android. Può essere eseguito sia a riga di comando sia utilizzando l'IDE Androd Studio e per poter funzionare ha bisogno che sia stato creato un dispositivo virtuale tramite l'apposito AVD (Android Virtual Device) Manager. É utile in quanto è possibile utilizzarlo per testare lo sviluppo dell'applicazione senza ricorrere ad un dispositivo mobile reale;
- Android Debug Bridge: tool particolarmente utile quando si vuole programmare a riga di comando e quindi permette di manipolare lo stato di un'istanza dell'emulatore o, se connesso, un dispositivo reale.

2.1.3 Configurazione applicazione

Un progetto in Android Studio si compone di tre parti principali: la cartella con il codice Java, la cartella "res" e la cartella contenente il file di configurazione detto "AndroidManifest.xml", contenuti nel modulo principale di default "app".

Dopodiché vi sarà una sezione "Gradle Scripts" contenente i file Gradle che sono utili

per la build automation. Un file Gradle serve per la costruzione di tutto il progetto e l'altro solo per il modulo “app”, che è quello solitamente modificato dal programmatore (verrà mostrato nella sezione seguente).

Di seguito è riportata un'immagine (Figura 2.1) raffigurante l'impostazione sopra descritta di un'applicazione:

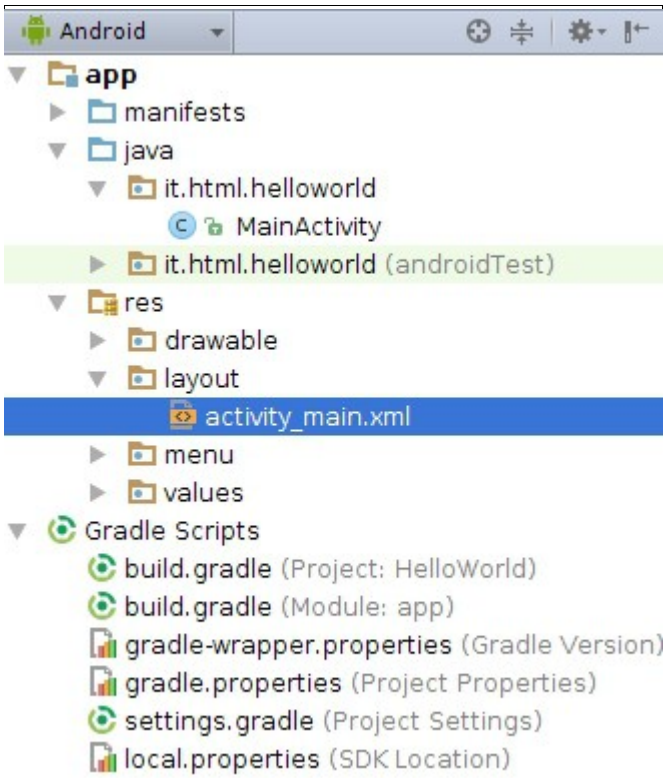

Figura 2.1: Esempio impostazione di un'applicazione Android

Gradle

Android Studio utilizza il toolkit Gradle per automatizzare e gestire il processo di costruzione dell'applicazione. Esso permette di definire configurazioni personalizzate per l'implementazione dell'app attraverso la modifica dei file .gradle, aggiungendo le varie librerie nella sezione “dependencies” .

File “build.gradle”:

```
apply plugin: 'com.android.application'
```

```
android {
compileSdkVersion 21
buildToolsVersion "22.0.1"
defaultConfig {
applicationId "it.html.helloworld"
minSdkVersion 14
targetSdkVersion 21
versionCode 1
versionName "1.0"
}
buildTypes {
release {
minifyEnabled false
proguardFiles getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.pro'
}
}
}
dependencies {
compile fileTree(dir: 'libs', include: ['*.jar'])
compile 'com.android.support:appcompat-v7:22.1.1'
}
```

L' `apply plugin` carica il plugin Gradle per Android, per poter riuscire ad accedere al contenuto della direttiva `android{}` del file. Qui vengono impostati: la versione SDK minima dell'applicazione, il `tasgetSdkVersion`, la versione di sviluppo e la versione pubblica. Dentro alla sezione `dependencies` invece vengono inserite le librerie per espandere le funzionalità del progetto.

Dopo aver settato i file .gradle è necessario sincronizzare il progetto con il nuovo setting.

Manifest File

Ogni applicazione Android dev'essere provvista del file Manifest (fornito automaticamente da Android Studio). Il file Manifest rivela informazioni essenziali sull'applicazione che vengono utilizzate dai tool di sviluppo Android, dal sistema operativo stesso Android e da Google Play. Inoltre in esso sono dichiarati:

- il nome del package dell'app, che solitamente corrisponde al namespace. I tool di sviluppo Android si servono di tale nome per determinare la locazione delle entità di codice nello sviluppo dell'applicazione;
- I componenti dell'app, che includono le activities, i services, i broadcast receivers e i content providers;
- I permessi di cui l'app ha bisogno per accedere a parti protette del sistema o di altre app.

Ad esempio una parte del Manifest file dell'applicazione del nostro caso di studio è riportato di seguito:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.pc.wave">
    <uses-permission
android:name="android.permission.STORAGE" />
    <uses-permission
android:name="android.permission.INTERNET" />
    <application
        android:allowBackup="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action
android:name="android.intent.action.MAIN" />
                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Res

La cartella “res” invece contiene le risorse, ovvero un'icona o un'immagine qualsiasi, un documento XML, oppure un file binario che viene utilizzato nell'applicazione, dell'applicazione. L'ambiente Android riprende il concetto di 'programmazione dichiarativa', dove alcune informazioni (risorse) vengono memorizzate in file di configurazioni esterni al codice. Ciò offre il vantaggio di non dover ricompilare il codice al fronte di semplici modifiche ad un file di configurazione.

Ad ogni tipo di risorsa corrisponde una cartella di “res”, infatti essa è composta da:

- “drawable”: contenente tre cartelle chiamate “hdpi”, “ldpi” e “mdpi” che contengono immagini di tipo GIF, PNG e JPEG. Le immagini verranno scelte dalle cartelle a seconda del dispositivo Android in utilizzo;
- “values”: qui vengono messe le risorse XML (stringhe, interi e altri tipi)
- “layout”: contiene documenti XML che definiscono i componenti GUI utilizzati dall'applicazione. Tali componenti possono essere Layout o controlli grafici.

Quindi in sintesi, in res sono contenute le risorse che verranno utilizzate nelle Activities dell'applicazione, attraverso la classe R, contenuta nella cartella “gen” che definisce come gestire questo processo.

2.1.4 Componenti Android

I componenti Android, come suggerito sopra, utilizzati nell'applicazione del nostro caso di studio sono:

- Activities: derivate dalla classe “android.app.Activity”, sono elementi indipendenti l'uno dall'altro e che implementano l'interazione tra utente e dispositivo. Ogni Activity presenta un ciclo di vita, rappresentato in Figura 2.2:

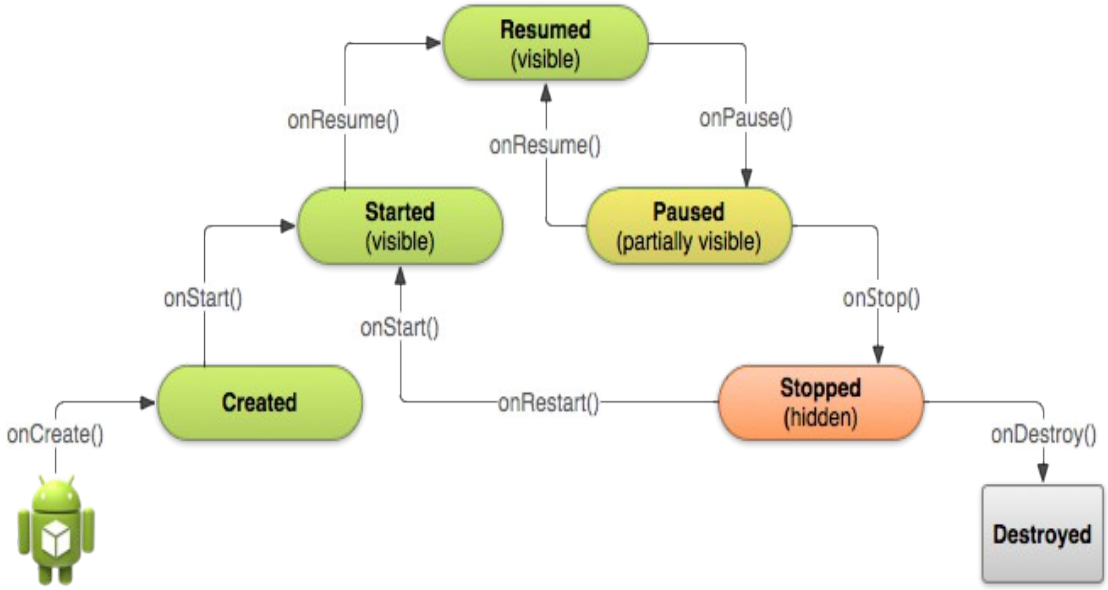

Figura 2.2: Ciclo di vita di un'Activity

Nel momento in cui un'Activity viene creata (metodo di callback `onCreate()`) attraverso la funzione startActivity viene avviata `onStart()`, dopodiché

l'Activity può essere messa in pausa `onPaused()` e `onResume()` o conclusa `onStop()` e `onDestroy()`;

- Services: derivati dalla classe “android.app.Service”, servono per mantenere l'esecuzione in background di processi e non presentano perciò un'interfaccia grafica con cui l'utente possa interagire;
- Intents: è una descrizione astratta di un’operazione da eseguire. L'uso più frequente di un Intent consiste nell'avvio di una Activity, come nell'esempio riportato in Figura 2.3:

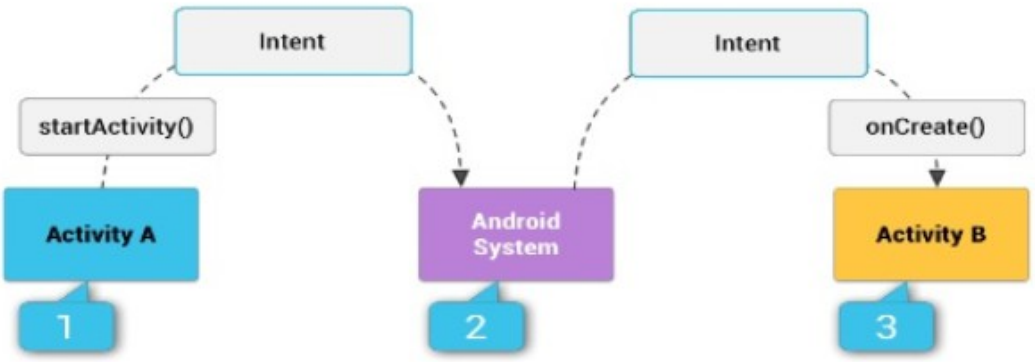

Figura 2.3: Avvio di un'Activity attraverso un Intent

Come in Figura 2.3: L'Intent descrive l'Activity da iniziare e contiene eventuali dati da passare all'Activity:

```
Intent intent = new Intent(CurrentActivity.this,
RequestedActivity.class);
startActivity(intent);
```

2.2 Tecnologia Firebase

Si è scelto di utilizzare Firebase per lo sviluppo dell'applicazione, poiché, esaminando le diverse tecnologie e gli strumenti per lo sviluppo dell'applicazione, è risultato che Firebase, negli ultimi tempi, è uno degli strumenti più utilizzati nello sviluppo di applicazioni Android.
Inoltre Firebase è già integrato in Android Studio, quindi non vi sono problemi di installazione o dipendenze.
Di seguito verrà introdotto più nel dettaglio Firebase e gli strumenti di Firebase utilizzati nel caso di studio.

2.2.1 Introduzione a Firebase

Firebase è un BaaS (BackEnd as a Service) di proprietà di Google: un Baas è un modello per fornire a sviluppatori di app web o mobili un modo per collegare le loro applicazioni a un backend cloud storage e API (interfacce di programmazione di un'applicazione) esposte da applicazioni backend, fornendo allo stesso tempo funzioni quali la gestione degli utenti, le notifiche push, e l'integrazione con servizi di social networking. Questi servizi sono forniti attraverso l'uso di kit di sviluppo software personalizzato (SDK) e interfacce di programmazione delle applicazioni (APIs).
Applicazioni web e mobile richiedono un analogo insieme di funzionalità sul backend, comprese le notifiche push, l'integrazione con i social network e il cloud storage. Ognuno di questi servizi ha la propria API che deve essere incorporata singolarmente nell'app, è ciò può richiedere molto tempo per gli sviluppatori di applicazioni. I provider di BaaS formano un ponte tra il frontend di un'applicazione e i vari cloud-based backends tramite una API unificata e le SDK(pacchetto di sviluppo per applicazioni). Fornendo questo tipo di gestione dei dati di backend gli sviluppatori non hanno bisogno di sviluppare il backend per ognuno dei servizi di cui l'applicazione ha bisogno, i questo

modo viene risparmiato tempo e denaro.

Firebase è una piattaforma per lo sviluppo di applicazioni web e mobili che non ha bisogno di un linguaggio di programmazione server side.

Caratteristiche non trascurabili di Firebase sono:

- capacità di sincronizzazione dati (e aggiornamento istantaneo), oltre che di storage;
- diverse librerie client per far integrare Firebase con ogni app Android, JavaScript, Java e con sistemi Apple: per tutte le più comuni tecnologie web e mobile esistono librerie già pronte per essere importate nei propri progetti;
- i dati in Firebase sono replicati e sottoposti a backup continuamente. La comunicazione con il client avviene sempre in modalità crittografata tramite SSL con certificati a 2048-bit;
- è possibile usufruire di piani con costi differenti. Il piano gratuito è quello utilizzato in questo caso di studio.

Inoltre Firebase si compone di diversi strumenti che semplificano il lavoro del programmatore, che di seguito sono elencati:

- Firebase Authentication
- Firebase Realtime Database
- Firebase Storage
- Firebase Functions
- Push Notification
- Firebase Analytics
- Firebase Cloud Messaging
- Firebase Hosting
- Firebase Test Lab for Android
- Firebase Crash Reporting
- Firebase Notification
- Firebase App Indexing
- Firebase Dynamic Link
- Firebase Invites

- Firebase Adwords

tra questi verranno trattati i primi quattro nell'elenco, in quanto sono quelli che sono stati utilizzati per sviluppare l'applicazione e che sono immediatamente disponibili con il piano di pagamento scelto, ovvero quello gratuito (come mostrato in Figura 2.4).[2]

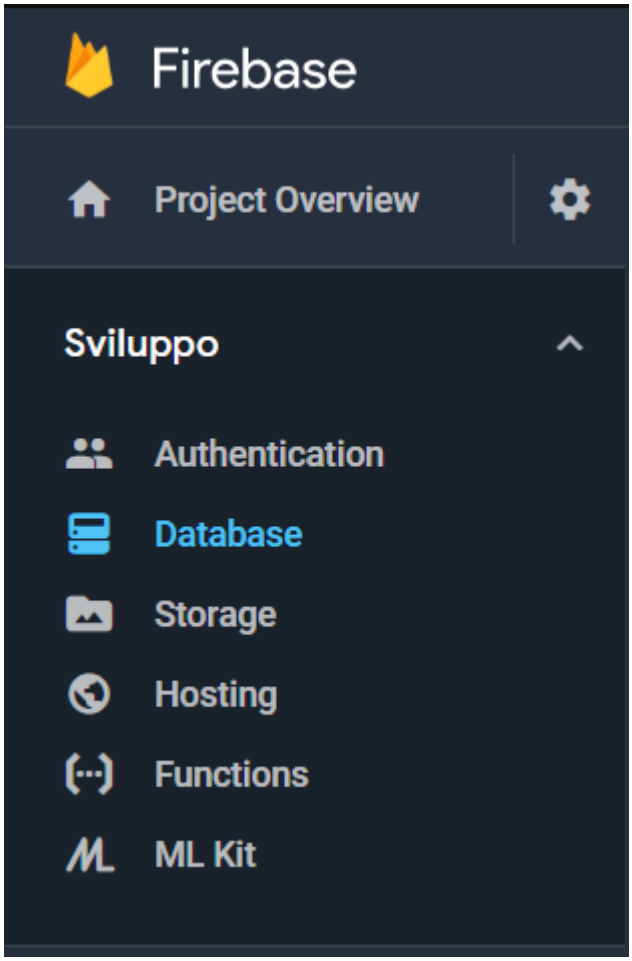

Figura 2.4: Strumenti disponibili gratuitamente in Firebase

2.2.2 Firebase Authentication

Firebase Authentication permette di implementare funzioni di autenticazione degli utenti nell'applicazione attraverso l'uso di passwords, numeri di telefono, o identity providers come Google, Facebook, Twitter e altri. Utilizzando la SDK di Firebase Authentication

è possibile integrare manualmente uno o più metodi di sign-in nella propria app, come si vede nella Figura 2.5 qui sotto:

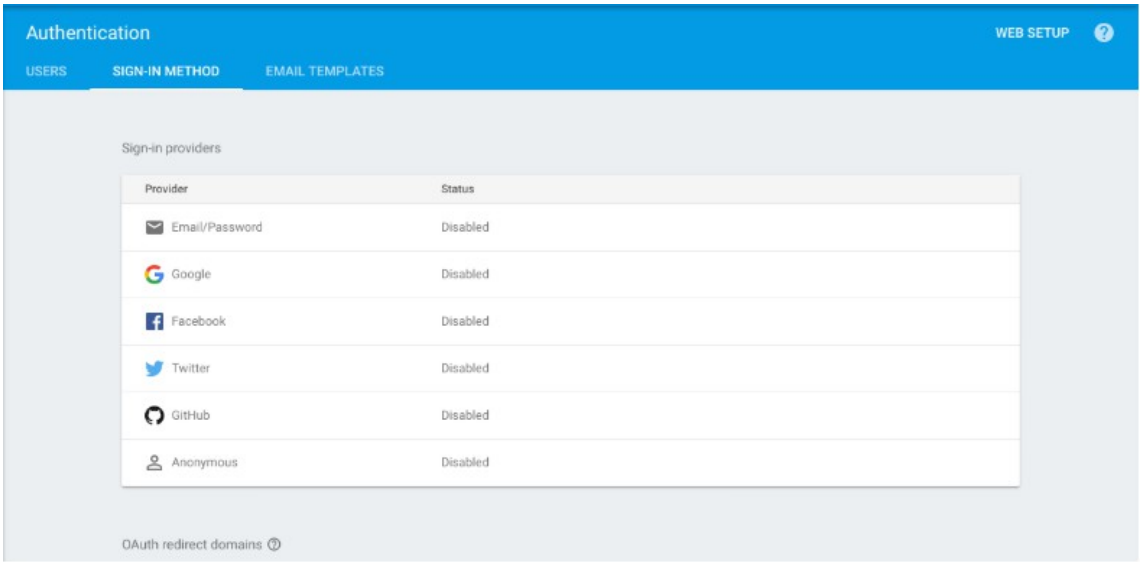

Figura 2.5: Tipologie di autenticazioni in Firebase Authentication

Nel nostro caso di studio si è scelto di basare l'autenticazione sull'email e sulla password dell'utente.

Per prima cosa quindi verranno inserite le credenziali (in questo caso email e password), che verranno passate alla SDK di Firebase Authentication. I servizi backend di Firebase verificheranno tali credenziali. Dopodiché sarà possibile accedere alle informazioni di base dello user e controllare il suo accesso ai dati immagazzinati in altri strumenti Firebase, come per esempio il Firebase Database. Inoltre attraverso la modifica delle regole di sicurezza di Firebase Realtime Databse e Cloud Storage sarà possibile limitare le autorizzazioni a tale utente, che di default ha i permessi di lettura e scrittura.

2.2.3 Lo user in Firebase

L'utente in Firebase rappresenta l'account di un utente registrato.

Ogni account presenta le seguenti caratteristiche: ID, un indirizzo email primario, un nome e un photo URL. Altre proprietà non possono essere aggiunte direttamente al set, è però possibile memorizzarle in altri servizi di storage di Firebase (come nel nostro caso il Firebase Realtime Database). Per esempio, la prima volta che l'utente si registra

all'applicazione, i dati del profilo dell'utente vengono popolati con le informazioni disponibili: nel nostro caso l'utente si è registrato con email e password, e solamente il campo indirizzo email verrà popolato. In ogni caso, ogni volta che verranno modificati i dati dell'utente, le sue informazioni verranno aggiornate automaticamente.
Nel momento in cui un utente si registra o effettua il login, diventa il “current user” dell'istanza Auth. L'istanza mantiene lo stato dell'utente in modo tale da non perdere le informazioni dell'utente ad ogni refresh di pagina o restart dell'applicazione.
Quando l'utente effettua il logout, l'istanza Auth cessa di mantenere un riferimento all'oggetto user, e non c'è nessun current user.

2.2.4 Firebase Realtime Database

Realtime Database è un database NoSQL cloud-hosted. In quanto NoSQL ha diverse ottimizzazioni e funzionalità rispetto ad un database relazionale, che verranno trattate più nel dettaglio in seguito. L'API di Realtime Database è disegnata per permettere solamente operazioni che possano essere eseguite rapidamente. Questo fa sì che pur avendo milioni di utenti attivi sull'applicazione, non ci siano problemi di responsività.
Realtime Database utilizza la sincronizzazione dei dati invece delle solite richieste HTTP. In questo modo ogni volta che i dati vengono modificati, il client connesso riceve tali cambiamenti istantaneamente.
I dati vengono sincronizzati attraverso i clients in realtime, e rimangono disponibili anche se l'applicazione va offline; nel momento in cui la connessione riprende, la funzione di sincronizzazione del Realtime Database provvede tutti i file mancanti al client, cosa molto importante per la stabilità del lavoro svolto dall'applicazione.
Il Realtime Database di quest'applicazione ad esempio è popolato con i dati di esempio utilizzati per testarla e si presenta così Figura 2.6:

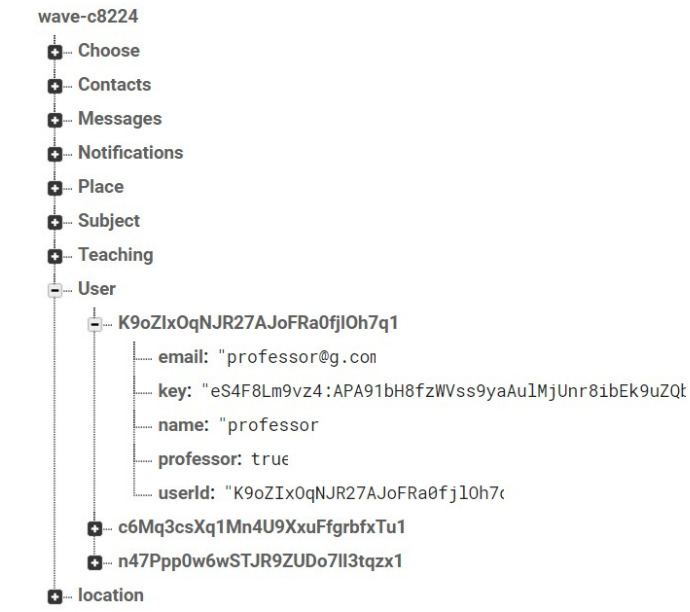

Figura 2.6: Esempio di database in Realtime Database di Firebase

2.2.5 Differenza tra database relazionale e NoSQL

La strutturazione rigida dei contenuti, tipica dei database relazionali, è un elemento assente nei database NoSQL, e proprio tale assenza è uno degli aspetti che maggiormente ne hanno permesso il successo.

Le informazioni non sono più memorizzate in tabelle, ma in oggetti completamente diversi e non necessariamente strutturati, come ad esempio documenti archiviati in collezioni.

Un oggetto JSON ad esempio, come vedremo in seguito, rappresenta un documento da inserire nel database.

La chiave primaria del documento è un campo denominato '_id' che, se non viene fornito in fase di inserimento, verrà aggiunto automaticamente dal DBMS.

Nei DBMS NoSQL a documenti, è significativa l'assenza delle relazioni. I meccanismi con cui vengono collegate le informazioni sono infatti due:

- Embedding: significa annidare un oggetto JSON all'interno di un altro. Questa tecnica sostituisce molto spesso le relazioni 1-a-1 e 1-a-molti. È tuttavia

sconsigliabile utilizzarla quando i documenti (quello annidato e quello che lo contiene) crescono di dimensione in maniera sproporzionata tra loro, oppure se la frequenza di accesso ad uno dei due è molto minore di quella dell'altro;

• Referencing: somiglia molto alle relazioni dei RDBMS, e consiste nel fare in modo che un documento contenga, tra i suoi dati, l'id di un altro documento. Molto utile per realizzare strutture complesse, relazioni molti-a-molti.

2.2.6 Cos'è JSON?

JSON (Javascript Object notation), è un tipo di formato molto utilizzato per lo scambio di dati in applicazioni client-server. È basato su JavaScript, ma il suo sviluppo è specifico per lo scambio di dati ed è indipendente dallo sviluppo del linguaggio di scripting dal quale nasce e con il quale è perfettamente integrato e semplice da utilizzare.

JSON è una valida alternativa al formato XML-XSLT e sempre più servizi di Web Services mettono a disposizione entrambe le possibilità di integrazione. Leggere ed interpretare uno stream in JSON è semplice in tutti i linguaggi, non solo in JavaScript, con cui è completamente integrato ma anche con PHP, Java ed altri linguaggi server-side, per mostrare i dati da fonti esterne ed impaginarli secondo le proprie soluzioni personalizzate.

2.2.7 Perché prediligere un database NoSQL

Limiti dei database relazionali:

Sono state effettuate ricerche per valutare i pro e i contro di un database SQL/NoSQL ed è emerso, come già detto sopra, che non sempre i DBMS tradizionali si dimostrano in

grado di gestire quantità di dati molto grandi, se non al prezzo di performance molto più limitate e costi piuttosto elevati. Qui di seguito sono stati raccolti alcuni punti di debolezza ravvisati nell'uso dei DBMS relazionali:

- i Join: nonostante la loro efficacia, queste operazioni coinvolgono spesso più righe del necessario (soprattutto se le query non sono ottimizzate), limitando le performance delle interrogazioni eseguite;
- struttura rigida delle tabelle, che si rivela utile finché si ha necessità di introdurre informazioni caratterizzate sempre dalle medesime proprietà, ma molto meno efficiente per informazioni di natura eterogenea;
- conflitto di impedenza, che consiste nella differenza strutturale tra i record delle tabelle e gli oggetti comunemente utilizzati per gestire i dati nei software che interagiscono con le basi di dati. Questo problema ha prodotto – tra l'altro - la nascita di strumenti come gli O/RM, librerie in grado di convertire oggetti in record (e viceversa);
- proprietà ACID: ogni transazione deve essere atomica, consistente, isolata e duratura. Implementare queste proprietà, però, comporta una serie di controlli e precauzioni che penalizzano le prestazioni del sistema. I database non relazionali sono meno stringenti, offrendo una maggiore elasticità.[3]

Quando il NoSQL conviene:

Sempre dopo aver effettuato svariate ricerche, sono stati raccolte informazioni sul perché sia conveniente utilizzare un database NoSQL piuttosto che relazionale. In generale, conviene scegliere l'approccio NoSQL quando:

- la struttura dei dati non è definibile a priori. Pensiamo ad esempio alla gestione dei contenuti che impongono oggi i social network o gli applicativi web per la gestione dei contenuti;
- i dati disposti nei vari oggetti sono molto collegati tra loro e, situazione aggravata da una loro gran quantità, il Join rischia di non essere lo strumento ottimale: sarebbe da prediligere la navigazione tra oggetti sfruttando i riferimenti tra i vari nodi di informazione;
- è necessario interagire molto frequentemente con il database, volendo evitare

conversioni onerose tra record e oggetti, né tanto meno ricorrere all'utilizzo di O/RM o altre librerie specifiche;

- servono prestazioni più elevate, potendo considerare troppo stringenti le proprietà ACID per certi campi di applicazione.

Qui di seguito Figura 2.7 uno schema riassuntivo:

	NoSQL	SQL
Model	Non-relational	Relational
	Stores data in JSON documents, key/value pairs, wide column stores, or graphs	Stores data in a table
Data	Offers flexibility as not every record needs to store the same properties	Great for solutions where every record has the same properties
	New properties can be added on the fly	Adding a new property may require altering schemas or backfilling data
	Relationships are often captured by denormalizing data and presenting it in a single record	Relationships are often captured in a using joins to resolve references across tables
	Good for semi-structured data	Good for structured data
Schema	Dynamic or flexible schemas	Strict schema
	Database is schema-agnostic and the schema is dictated by the application. This allows for agility and highly iterative development	Schema must be maintained and kept in sync between application and database
Transactions	ACID transaction support varies per solution	Supports ACID transactions
Consistency	Consistency varies per solution, some solutions have tunable consistency	Strong consistency supported
Scale	Scales well horizontally	Scales well vertically

Figura 2.7: Database SQL e NoSQL a confronto

Questi sono solo alcuni dei motivi che hanno portato a prediligere la scelta di utilizzare Firebase Realtime Database, invece che, ad esempio, SQLite, che si trova già installato di default su Android Studio. Oltre a ciò c'è stata anche la curiosità e la volontà di imparare ad utilizzare uno strumento nuovo e diverso da ciò che già era conosciuto e quindi meno stimolante.

2.2.8 Firebase Cloud Storage

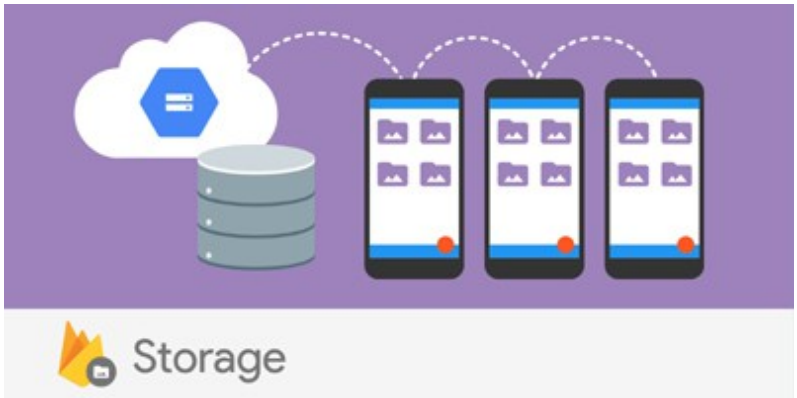

Figura 2.8: Concetto alla base di Firebase Cloud Storage

Cloud Storage è un servizio di storage. Le SDK di Firebase per Cloud Storage fanno sì che le operazioni di upload e download dei file vengano eseguite secondo i criteri di sicurezza di Google. Grazie ad esse è possibile immagazzinare immagini, audio, video e altri contenuti user-generated direttamente da client.

2.2.9 Come aggiungere Firebase alla propria applicazione

Firebase è facilmente integrabile nelle applicazioni Android. In Android Studio basta semplicemente clickare su Tools->Firebase e seguire le istruzioni per aggiungere le dipendenze necessarie all'integrazione dello strumento di Firebase che si vuole utilizzare.

2.3 Tecnologia Geofire

L'adozione di Geofire nasce dalla necessità di uno strumento per la localizzazione dei dispositivi/utenti che utilizzano l'applicazione. Infatti una delle funzioni principali dell'applicazione in esame è la ricerca per città o per “posizione corrente” del professore che dia ripetizioni per una data materia.
É stato scelto proprio Geofire, invece di altri strumenti di geolocalizzazione, perché si integra perfettamente con Firebase, essendo un suo addon.

2.3.1 Introduzione a Geofire

Geofire è una libreria open source che permette di immagazzinare nel Realtime Database di Firebase la location di un dispositivo e di aggiornarla in realtime. In questo modo è possibile creare query per l'interlocuzione del database in realtime, specificando un raggio, espresso in metri, entro il luogo richiesto, che può essere la posizione attuale o una città/paese/indirizzo le cui coordinate sono salvate nel database.[4]

Un esempio viene riportato in Figura 2.9 con i dati del database dell'applicazione:

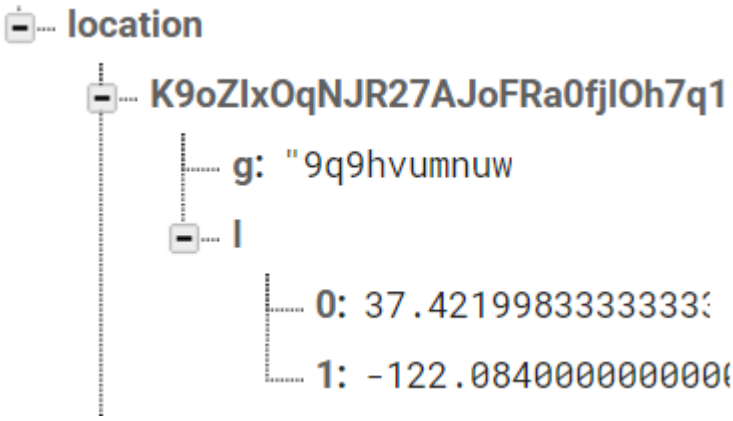

Figura 2.9: Chiave Geofire salvata in Firebase Realtime Database

2.4 Node.js

Node.js è stato utilizzato per implementare le notifiche push che vengono ricevute da un'utente al quale è stata inviata una richiesta di contatto all'interno dell'applicazione.
É stato quindi creato uno script apposito per la creazione della notifica.[5]

2.4.1 Introduzione a Node.js

Node.js è un framework per realizzare applicazioni in JavaScript. La caratteristica che ne ha determinato il successo è quella di poter scrivere applicazioni non solo “client-side”, ma anche “server-side”.
La piattaforma è basata sul JavaScript Engine V8, che è il runtime di Google utilizzato anche da Chrome e disponibile sulle principali piattaforme, anche se maggiormente performante su sistemi operativi UNIX-like.
Altra caratteristica che ne ha garantito il successo è la gestione asincrona delle azioni. Ciò comporta un miglioramento in termini di efficienza rispetto agli altri web server, poiché le azioni non vengono messe in coda ed eseguite solo nel momento in cui un'altra è stata completata.
Node.js è dotato di un package manager “npm” attraverso cui è possibile scaricare e installare numerosi plugin e applicazioni. Nel nostro caso è stato installato “firebase-tools”, per poter realizzare lo script per le notifiche dell'applicazione, mentre lo script, come spiegato nella prossima sezione, è stato scritto in “index.js”.

2.5 Picasso

Picasso è una potente libreria per il downloading e il caching di immagini per Android.
Semplifica il processo di gestione delle immagini da uno storage esterno.
Molti problemi comuni nel loading delle immagini in Android sono gestiti automaticamente da Picasso:

- gestire un ImageView in un Adapter;
- trasformazioni complesse dell'immagine con un utilizzo minimo della memoria;
- caching automatico di memoria e disco.

Grazie a Picasso quindi risulta molto semplice gestire un'immagine, che come nel nostro caso, è stato presa dal Firebase Storage.[6]
Per poter utilizzare Picasso basta aggiungerlo tra le dipendenze nel file "build.gradle" dell'applicazione:

```
implementation 'com.squareup.picasso:picasso:(insert latest version)'
```

Dopodiché si può subito iniziare ad utilizzare Picasso, basta avere un ImageView pronto per inserirci l'immagine:

```
ImageView imageView = (ImageView) findViewById(R.id.imageView);

Picasso.with(this).load(<percorso_immagine>).into(imageView);
```

Inoltre è possibile aggiungere altre regole per la visualizzazione dell'immagine come:

```
.resize(100, 100)

.rotate(180)

.placeholder(R.drawable.image_name)

.error(R.drawable.image_name)
```

3 Progettazione e Implementazione

3.1 Progettazione dell'applicazione

3.1.1 Progettazione Logica

Avendo a che fare con un database NoSQL, la progettazione logica non è avvenuta con l'ausilio dello schema ER, essendo questo uno strumento di mappatura delle relazioni tra le varie entità che popolano il database. Nel nostro caso la progettazione è stata fatta seguendo la logica dei database NoSQL, dove ogni dato è immagazzinato nel file .json (che nel caso di Firebase è possibile esportare direttamente dalla pagina del database del progetto).
Lo schema seguente (Figura 3.1) quindi rappresenta graficamente tale file, con i dati di esempio che sono stati utilizzati per testare in fase di ultimazione l'applicazione.

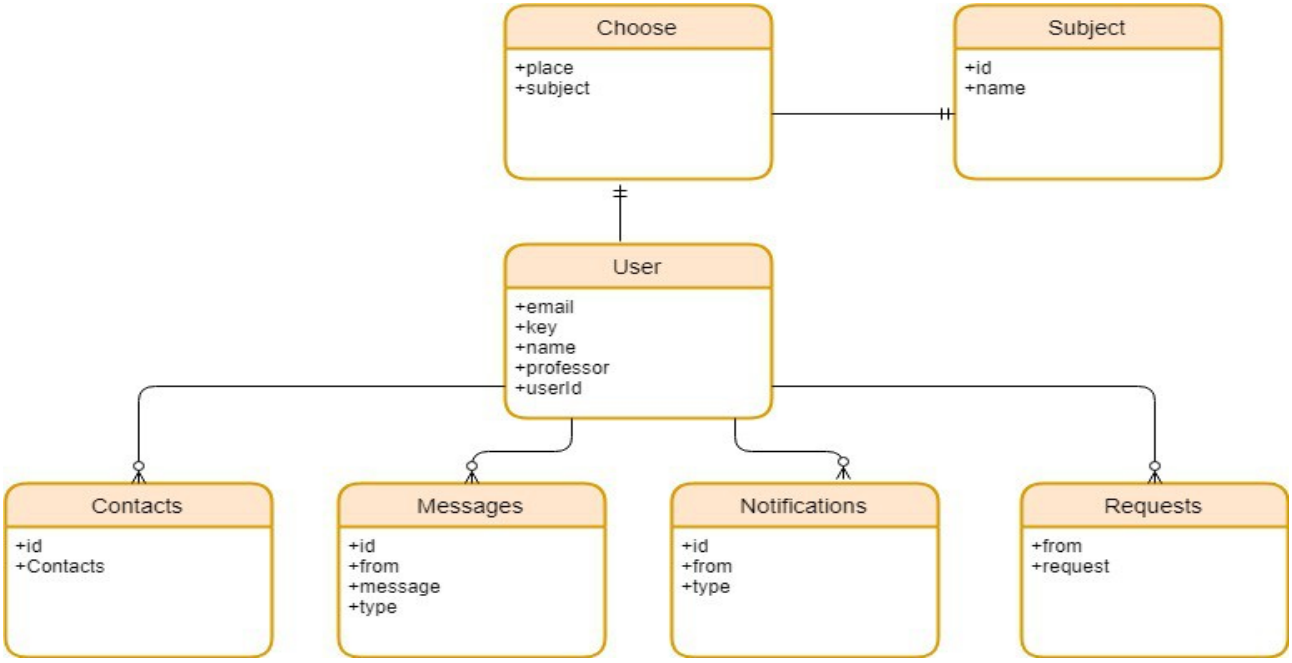

Figura 3.1: Diagramma completo delle Entità che compongono il progetto

Di seguito invece si riporta il file .json sopracitato, con i dati di prova suddetti:

```
{
  "Choose" : {
    "c6Mq3csXq1Mn4U9XxuFfgrbfxTu1" : {
      "place" : "Mountain View",
      "subject" : "Fisica"
    },
    "n47Ppp0w6wSTJR9ZUDo7lI3tqzx1" : {
      "place" : "Mountain View",
      "subject" : "Fisica"
    }
  },
  "Contacts" : {
    "K9oZIxOqNJR27AJoFRa0fjlOh7q1" : {
      "n47Ppp0w6wSTJR9ZUDo7lI3tqzx1" : {
        "Contacts" : "Saved"
      }
    },
    "n47Ppp0w6wSTJR9ZUDo7lI3tqzx1" : {
      "K9oZIxOqNJR27AJoFRa0fjlOh7q1" : {
        "Contacts" : "Saved"
      }
    }
  },
"Messages" : {
    "K9oZIxOqNJR27AJoFRa0fjlOh7q1" : {
```

```
      "n47Ppp0w6wSTJR9ZUDo7lI3tqzx1" : {
        "-LgnDIv3cIQO1-_LVnt5" : {
          "from" : "K9oZIxOqNJR27AJoFRa0fjlOh7q1",
          "message" : "hello",
          "type" : "text"
        }
      }
    },
    "n47Ppp0w6wSTJR9ZUDo7lI3tqzx1" : {
      "K9oZIxOqNJR27AJoFRa0fjlOh7q1" : {
        "-LgnDIv3cIQO1-_LVnt5" : {
          "from" : "K9oZIxOqNJR27AJoFRa0fjlOh7q1",
          "message" : "hello",
          "type" : "text"
        }
      }
    }
  },
  "Notifications" : {
    "K9oZIxOqNJR27AJoFRa0fjlOh7q1" : {
      "-LerlpLTU3vzdooOFLbq" : {
        "from" : "n47Ppp0w6wSTJR9ZUDo7lI3tqzx1",
        "type" : "request"
      }
    }
  },
  "Place" : {
    "Mountain View" : {
      "latitude" : 37.421998333333335,
      "longitude" : -122.08400000000002,
      "name" : "Mountain View"
    }
  },
  "Subject" : {
    "Subject1" : {
      "id" : "1",
      "name" : "Fisica"
    },
    "Subject2" : {
      "id" : "2",
      "name" : "Matematica"
    },
    "Subject3" : {
      "id" : "3",
      "name" : "Italiano"
    }
  },
  "Teaching" : {
    "K9oZIxOqNJR27AJoFRa0fjlOh7q1" : [ null, "Fisica",
"Matematica", "Italiano" ]
  },
```

```
  "User" : {
    "K9oZIxOqNJR27AJoFRa0fjlOh7q1" : {
      "email" : "professor@g.com",
      "key" : "fuVpqfpqKn0:APA91bHyOFyg4M_wHpOXgHawL-B-
chZaCO23lF9nCo_J9xChIkUgtUoXhOPU6QvidIrKFi-
cCsd4GZ7n4kkEdkPcgaFlA0Y9dzLnNuXB09yqdZyaTscEhIN0sUF8uKvGUwjcEx_
kWt-U",
      "name" : "professor",
      "professor" : true,
      "userId" : "K9oZIxOqNJR27AJoFRa0fjlOh7q1"
    },
    "c6Mq3csXq1Mn4U9XxuFfgrbfxTu1" : {
      "email" : "stud2@g.com",
      "key" :
"eS4F8Lm9vz4:APA91bH8fzWVss9yaAulMjUnr8ibEk9uZQbhdDBE2sZfQDc_GLa
qWSy_Zfo8ZGLpfOh5VVU7sRZqn-oUZ8mVFy0S-
dVU9r6dYx27H36MNUKEaC_AkF_B1AvVffZ7Z3Cbl3MHjgJqSFt2",
      "name" : "stud2",
      "professor" : false,
      "userId" : "c6Mq3csXq1Mn4U9XxuFfgrbfxTu1"
    },
    "n47Ppp0w6wSTJR9ZUDo7lI3tqzx1" : {
      "email" : "student@g.com",
      "key" :
"eS4F8Lm9vz4:APA91bH8fzWVss9yaAulMjUnr8ibEk9uZQbhdDBE2sZfQDc_GLa
qWSy_Zfo8ZGLpfOh5VVU7sRZqn-oUZ8mVFy0S-
dVU9r6dYx27H36MNUKEaC_AkF_B1AvVffZ7Z3Cbl3MHjgJqSFt2",
      "name" : "student",
      "professor" : false,
      "userId" : "n47Ppp0w6wSTJR9ZUDo7lI3tqzx1"
    }
  },
  "location" : {
    "K9oZIxOqNJR27AJoFRa0fjlOh7q1" : {
      ".priority" : "9q9hvumnuw",
      "g" : "9q9hvumnuw",
      "l" : [ 37.421998333333335, -122.08400000000002 ]
    },
    "Mountain View" : {
      ".priority" : "9q9hvumnuw",
      "g" : "9q9hvumnuw",
      "l" : [ 37.421998333333335, -122.08400000000002 ]
    },
    "c6Mq3csXq1Mn4U9XxuFfgrbfxTu1" : {
      ".priority" : "9q9hvumnuw",
      "g" : "9q9hvumnuw",
      "l" : [ 37.421998333333335, -122.08400000000002 ]
    },
    "n47Ppp0w6wSTJR9ZUDo7lI3tqzx1" : {
      ".priority" : "9q9hvumnuw",
      "g" : "9q9hvumnuw",
```

```
      "l" : [ 37.421998333333335, -122.08400000000002 ]
    }
  }}
```

3.1.2 Costruzione delle Entità

Entità User:

L'entità User rappresenta l'utente registrato nell'applicazione. Può essere registrato come professore o come studente a seconda che l'attributo “professor” sia settato a True o a False. Ogni User ha uno userID che viene utilizzato per gestire le azioni dell'utente o in relazione ad altre entità:

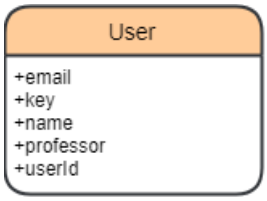

Figura 3.2: Entità User


```
"User" : {
    "K9oZIxOqNJR27AJoFRa0fjlOh7q1" : {
      "email" : "professor@g.com",
      "key" : "fuVpqfpqKn0:APA91bHyOFyg4M_wHpOXgHawL-B-
chZaCO23lF9nCo_J9xChIkUgtUoXhOPU6QvidIrKFi-
cCsd4GZ7n4kkEdkPcgaFlA0Y9dzLnNuXB09yqdZyaTscEhIN0sUF8uKvGUwjcEx_
kWt-U",
      "name" : "professor",
      "professor" : true,
      "userId" : "K9oZIxOqNJR27AJoFRa0fjlOh7q1"
    }
```


Entità Place:

L'entità Place rappresenta la località che viene memorizzata nel momento in cui un utente professore effettua la registrazione. Differentemente da “location” questa entità è indipendente dallo User con la quale è stata registrata, infatti della località vengono memorizzati solo gli attributi come riportati in Figura 3.3 :

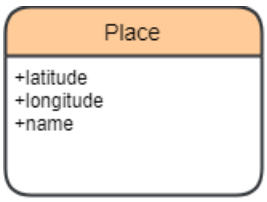

Figura 3.3: Entità Place

```
"Place" : {
    "Mountain View" : {
      "latitude" : 37.421998333333335,
      "longitude" : -122.08400000000002,
      "name" : "Mountain View"
    }
  }
```

Entità Subject:

L'entità Subject è stata aggiunta manualmente in Firebase Realtime Database con materie di esempio per testare l'applicazione. Gli oggetti immagazzinati nel database vengono mostrati nella checkbox di scelta dello user professore, nel momento in cui indica quali materie insegna. Inoltre le materie appaiono anche come oggetto di scelta nello Spinner della pagina dove lo studente può filtrare la ricerca di ripetizioni per luogo e, appunto, per materia.

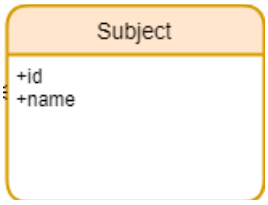

Figura 3.4: Entità Subject

```
"Subject" : {
    "Subject1" : {
      "id" : "1",
      "name" : "Fisica"
    },
```

Entità Teaching:

Nel momento in cui un professore spunta nel Checkbox delle materie quelle che

insegna, queste ultime vengono inserite in Teaching, in modo tale da poter avere subito la lista dei professori che insegnano una certa materia, quando lo studente effettua la ricerca.

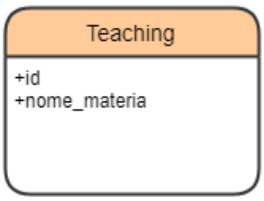

Figura 3.5: Entità Teaching

```
"Teaching" : {
    "K9oZIxOqNJR27AJoFRa0fjlOh7q1" : [ "Fisica", "Matematica",
"Italiano" ]
  }
```

Entità Choose:

Quando l'utente studente effettua le scelte dagli spinner, l'entità Choose contenente lo userID di quell'utente viene aggiornato (o creato) con i nuovi filtri e si può trovare la lista di professori che corrispondono a quei criteri.

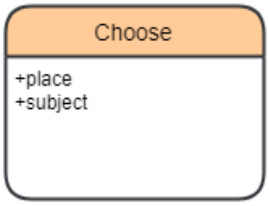

Figura 3.6: Entità Choosing

```
"Choose" : {
    "c6Mq3csXq1Mn4U9XxuFfgrbfxTu1" : {
      "place" : "Mountain View",
      "subject" : "Fisica"
    }
```

Entità location:

Tale entità viene popolata automaticamente da Geofire, poiché, ogni qualvolta viene rivelata una nuova location del device corrente, Geofire aggiorna/crea la posizione, creando una chiave “g” contenente le coordinate della posizione, senza la quale non

potrebbero essere accedute le funzionalità di Geofire.

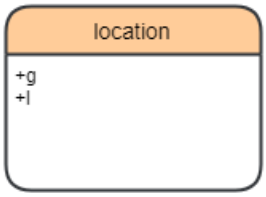

Figura 3.7: Entità location

```
"location" : {
    "K9oZIxOqNJR27AJoFRa0fjlOh7q1" : {
      ".priority" : "9q9hvumnuw",
      "g" : "9q9hvumnuw",
      "l" : [ 37.421998333333335, -122.08400000000002 ]
    }
```

Entità Notifications:

Qui le notifiche ricevute dai destinatari di richieste di contatto (per poter contattare un professore, lo studente deve prima inviargli una richiesta e, solo dopo che il professore ha accettato, si può iniziare una conversazione), vengono memorizzate, finché l'utente non apre l'applicazione e controlla la richiesta.

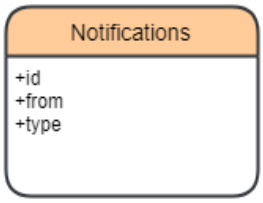

Figura 3.8: Entità Notifications

```
"K9oZIxOqNJR27AJoFRa0fjlOh7q1" : {
      "-LerlpLTU3vzdooOFLbq" : {
        "from" : "n47Ppp0w6wSTJR9ZUDo7lI3tqzx1",
        "type" : "request"
      }
    }
```

Entità Contacts:

Ogni utente può accedere ai propri contatti dall'applicazione, perciò è necessario mantenere nel database un'entità come quella in Figura 3.9 :

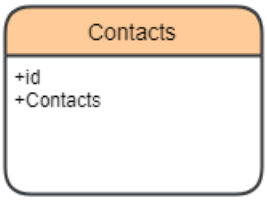

Figura 3.9: Entità Contacts

```
"K9oZIxOqNJR27AJoFRa0fjlOh7q1" : {
      "n47Ppp0w6wSTJR9ZUDo7lI3tqzx1" : {
        "Contacts" : "Saved"
      }
    }
```

Entità Messages:

Per ogni messaggio inviato viene mantenuto nel database un oggetto dell'entità Messages, contenente informazioni riguardanti il destinatario, il mittente e il corpo del messaggio:

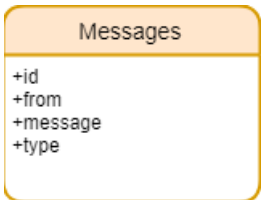

Figura 3.10: Entità Messages

```
"Messages" : {
    "K9oZIxOqNJR27AJoFRa0fjlOh7q1" : {
      "n47Ppp0w6wSTJR9ZUDo7lI3tqzx1" : {
        "-LgnDIv3cIQO1-_LVnt5" : {
          "from" : "K9oZIxOqNJR27AJoFRa0fjlOh7q1",
          "message" : "hello",
          "type" : "text"
        }
      }
    }
```

La prima chiave d'esempio `"K9oZIxOqNJR27AJoFRa0fjlOh7q1"` identifica il mittente del messaggio, mentre la seconda annidata è l'id del ricevente. Le chiavi sono quelle definite durante la creazione dell'utente e sono quindi chiavi primarie. La terza chiave annidata è invece l'id del messaggio.

3.1.3 Costruzione del diagramma

L'entità User presenta il campo booleano “professor” che determina se l'utente registrato è un professore (professor=True) o uno studente.
Nel caso lo User sia un professore, può insegnare una o più materie tra quelle presenti (Subjects), come rappresentato nella Figura 3.11 .

Figura 3.11: Relazione User-Subject

L'entità Teaching subentra per poter memorizzare la scelta di materie effettuata al momento della registrazione (e in caso di modifica) da parte del professore. In questo modo si segue la logica NoSQL del Realtime Database di Firebase.

Ogni materia può essere scelta da più utenti (studenti o professori), come mostrato in Figura 3.12 .

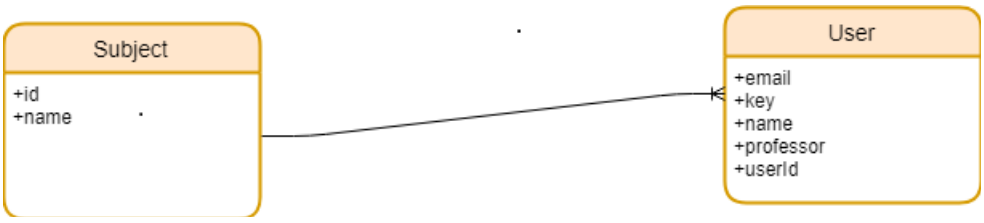

Figura 3.12: Relazione Subject-User

La differenza tra l'entità location e l'entità Place consiste nel fatto che la prima viene mantenuta per poter realizzare le funzionalità di Geofire, mentre gli oggetti della seconda vengono memorizzati una sola volta, quando un utente professore si registra.
Ad ogni utente quindi è associata una location, vedi Figura 3.13, che si aggiorna ad ogni spostamento del device online, mentre Place non è associata ad uno user specifico, se non nell'atto di creazione dell'entità. Tale scelta è stata fatta per poter creare automaticamente (e non quindi manualmente da Firebase) i luoghi tra cui lo user studente può scegliere di cercare ripetizioni.

Figura 3.13: Relazione User-location

L'utente studente, nel momento in cui effettua il login, deve scegliere in quale città effettuare la ricerca e per quale materia. Tale azione è mantenuta nel database dall'entità Choose, come in Figura 3.14 .

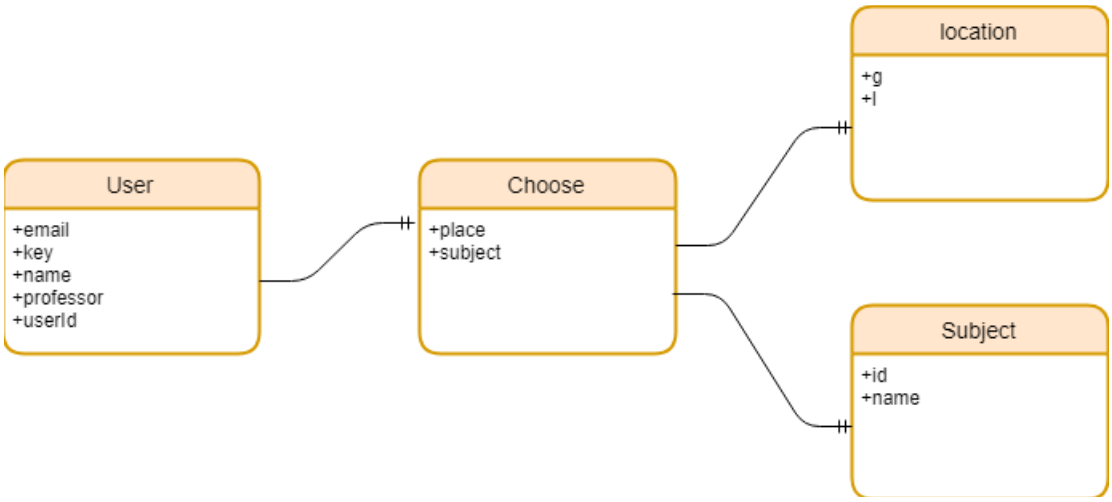

Figura 3.14: Relazione User-Choose-Subject-location

Un utente studente può contattare un utente professore, e viceversa. Ogni utente può avere da 0 a N contatti. Così come può avere da 0 a N richieste di contatto, da 0 a N notifiche e da 0 a N messaggi.

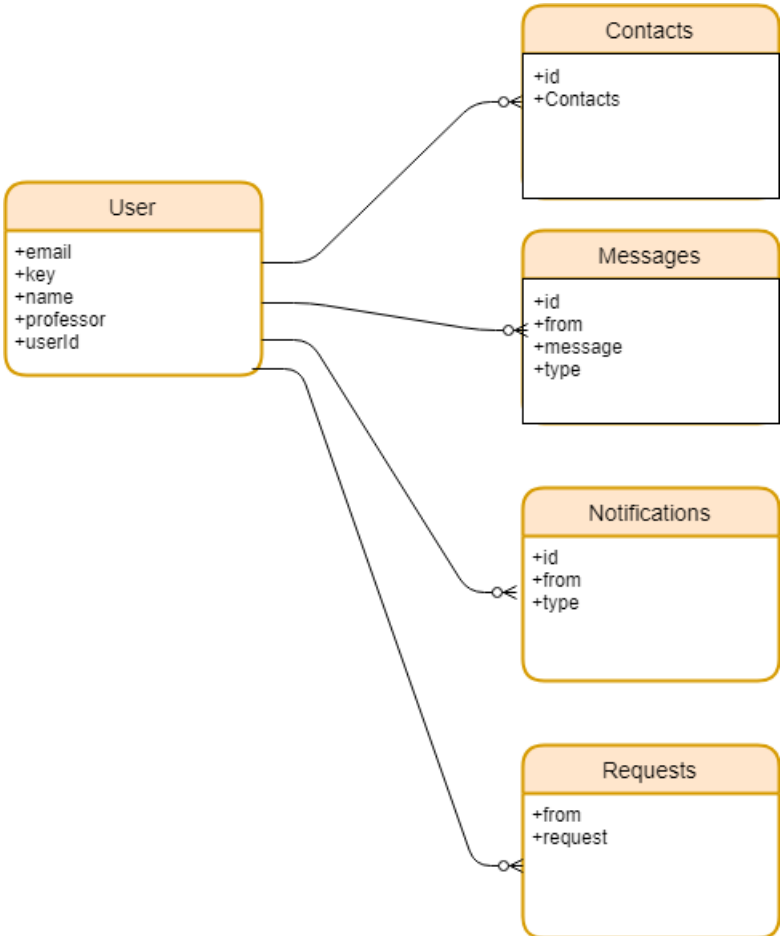

Figura 3.15: Relazione User-Contacts-Messages-Notifications-Requests

3.2 Implementazione dell'applicazione

Per implementare l'applicazione è stato utilizzato Android Studio e per gestire i dati Firebase. Per testare le funzionalità ad ogni ulteriore sviluppo sono stati utilizzati gli emulatori virtuali di Android Studio, con sistema operativo Android 9 Pie, API level 28.

3.2.1 Manifest File

Qui di seguito viene riportato il Manifest File "AndroidManifest.xml", dentro al quale si trovano le istruzioni per far partire ed eseguire l'applicazione:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.pc.wave">
    <uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission
android:name="android.permission.READ_INTERNAL_STORAGE" />
    <uses-permission
android:name="android.permission.READ_EXTERNAL_STORAGE" />
    <uses-permission
android:name="android.permission.STORAGE" />
    <uses-permission
android:name="android.permission.INTERNET" />
    <application
        android:allowBackup="true"
        android:icon="@drawable/logo"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action
android:name="android.intent.action.MAIN" />
                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".RegistrationActivity" />
        <activity android:name=".LoginActivity" />
        <activity android:name=".StartProfessor"/>
        <activity android:name=".StartStudent" />
        <activity android:name=".HomeProfessor" />
```

```
        <activity android:name=".HomeStudent" />
        <activity android:name=".ProfileActivity" />
        <activity android:name=".DetailActivity" />
        <activity android:name=".ContactsActivity" />
        <activity android:name=".ChatActivity" />
        <activity android:name=".RequestActivity" />
    </application>
</manifest>
```

Nelle prime righe, dopo la definizione del package name, troviamo i permessi che l'applicazione ha sul dispositivo: i primi, riguardanti i permessi GPS, sono stati inseriti per sfruttare le funzionalità di Geofire, mentre gli altri per poter salvare o ricavare dati e immagini dallo storage del dispositivo, per ad esempio, salvare le immagini di profilo su Firebase Cloud Storage.

- ACCESS_COARSE_LOCATION permette di accedere ad una location approssimata;
- ACCESS_FINE_LOCATION permette di accedere ad una location precisa.

Dopodiché vengono definite alcune componenti grafiche (come il logo dell'applicazione), e poi si indica quale classe verrà lanciata per prima in `<category android:name="android.intent.category.LAUNCHER" />` che nel nostro caso è MainActivity.java .

3.2.2 RegistrationActivity

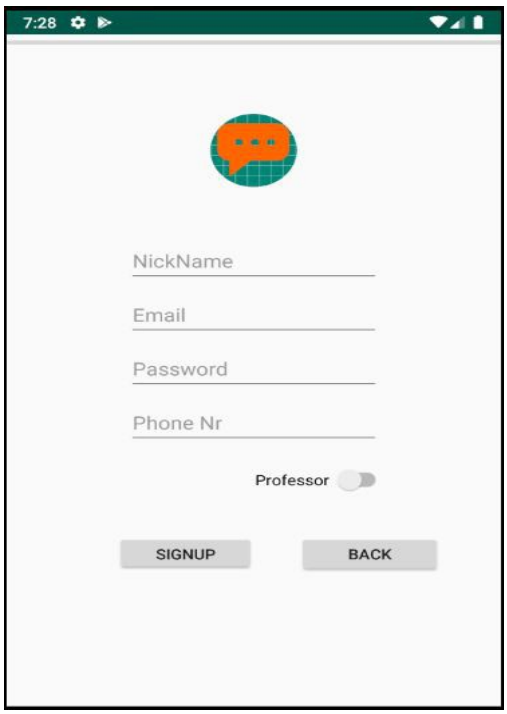

Figura 3.16: Screenshot schermata di registrazione

I campi richiesti sono: nickname, email, password, phone nr. Lo switch presente in basso a destra con di fianco scritto "Professor" se viene impostato setterà il campo "professor" di User a True, altrimenti sarà False. Il cerchio in alto con il placeholder, se selezionato, apre la schermata con lo storage interno del dispositivo per poter scegliere la propria immagine da registrare nel profilo.

Se anche una sola di queste informazioni non viene inserita, o viene inserita in modo non corretto (ad esempio email già esistente, password troppo corta), viene visualizzato un Toast con messaggio di errore finché non vengono inseriti i dati correttamente.

Per prima cosa viene creato un riferimento al database Realtime Database di Firebase con `DatabaseReference` e si crea una sua istanza:

```
private DatabaseReference reference;
FirebaseDatabase firebaseDatabase;
firebaseDatabase = FirebaseDatabase.getInstance();
reference = firebaseDatabase.getReference("User");
```

Poi si crea un'istanza di FirebaseAuth, attraverso il quale sarà possibile accedere all'id dell'utente e settare l'utente che verrà registrato come CurrentUser. Inoltre dopo aver

creato l'istanza, è possibile utilizzarne i metodi di FirebaseAuth per implementare la procedura di registrazione :

```
auth.createUserWithEmailAndPassword(email, password)
        .addOnCompleteListener(RegistrationActivity.this, new
OnCompleteListener<AuthResult>() {
            @Override
            public void onComplete(@NonNull Task<AuthResult> task) {
                Toast.makeText(RegistrationActivity.this,
                  "createUserWithEmail:onComplete:" +
                  task.isSuccessful(), Toast.LENGTH_SHORT).show();

if (!task.isSuccessful()) {

            Toast.makeText(RegistrationActivity.this, "Authentication
            failed." + task.getException(),
            Toast.LENGTH_SHORT).show();
        }
    }
});
```

Nel momento in cui un utente effettua la registrazione, viene salvata la location attuale del dispositivo, per facilitare il popolamento del database in “Place”. Ciò è stato fatto infatti, come già spiegato nel capitolo precedente, per evitare di inserire i luoghi manualmente nel database.

Come per le altre interazioni con Realtime Database, inizialmente viene creata un'istanza e un riferimento al database, che poi verrà popolato in questo caso con il nome del luogo e le coordinate:

```
private DatabaseReference placeRef;

private Location location;

public Criteria criteria;
public String bestProvider;

placeRef = firebaseDatabase.getReference("Place");

//SAVING LOCATION ADDRESS

DatabaseReference georef =
FirebaseDatabase.getInstance().getReference("location");
```

In place vengono salvate le location al momento della registrazione, mentre poi viene creata un'istanza di Geofire che crea le chiavi che andrà ad aggiornare e ad utilizzare per recuperare le coordinate in “Location”

```
final GeoFire geoFire = new GeoFire(georef);
Geocoder geocoder = new Geocoder(this, Locale.getDefault());
LocationManager locationManager =
(LocationManager)this.getSystemService(Context.LOCATION_SERVICE);
LocationListener locationListener = new LocationListener() {
    @Override
    public void onLocationChanged(Location location) {
    }
    @Override
    public void onStatusChanged(String provider, int status, Bundle
extras) {
    }
    @Override
    public void onProviderEnabled(String provider) {
    }
    @Override
    public void onProviderDisabled(String provider) {
    }
};
criteria = new Criteria();
```

Prima di procedere al salvataggio delle coordinate e della location, l'applicazione verifica (codice qui di seguito) che siano stati verificati tutti i permessi per potere accedere alle informazioni del dispositivo:

```
bestProvider =
String.valueOf(locationManager.getBestProvider(criteria, true));
if (ActivityCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION) !=
PackageManager.PERMISSION_GRANTED &&
ActivityCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_COARSE_LOCATION) !=
PackageManager.PERMISSION_GRANTED) {
    // TODO: Consider calling
    //    ActivityCompat#requestPermissions
    // here to request the missing permissions, and then overriding
    //   public void onRequestPermissionsResult(int requestCode,
String[] permissions,int[] grantResults)
    // to handle the case where the user grants the permission. See
the documentation
    // for ActivityCompat#requestPermissions for more details.
    return;
}
```

Infine la location viene finalmente trovata e memorizzata nel database, creando una nuova entità Place e salvando la chiave in “location”:

```
location = locationManager.getLastKnownLocation(bestProvider);
if (location != null) {
    final Double latitude = location.getLatitude();
```

```
    final Double longitude = location.getLongitude();
    try {
      List<Address> address =
      geocoder.getFromLocation(location.getLatitude(),
      location.getLongitude(), 1);
      final String address_city = address.get(0).getLocality();
      placeRef.addValueEventListener(new ValueEventListener() {
            @Override
            public void onDataChange(@NonNull DataSnapshot
            dataSnapshot) {
                // The method onDataChange is called once with the
                  initial value and again whenever data at this
                  location is updated.

                if(!dataSnapshot.hasChild(address_city)){
                    Place place = new Place(address_city, latitude,
                     longitude);
                    placeRef.child(address_city).setValue(place);
                    geoFire.setLocation(address_city, new
                   GeoLocation(location.getLatitude(),
                   location.getLongitude()),new
                   GeoFire.CompletionListener(){
                               @Override
                               public void onComplete(String key,
                                DatabaseError error) {
                                   if (error != null) {
                                       System.err.println("There was
                                        an error saving the location
                                        to GeoFire: " + error);
                                   } else {
                                       System.out.println("Location
                                        saved successfully!");
                                   }
                               }
                    });
                }else{
                    System.out.println("PLACE ALREADY EXISTS");
                }
            }
            @Override
            public void onCancelled(@NonNull DatabaseError
            databaseError) {
            }
        });
    } catch (IOException e) {
        e.printStackTrace();
    }
}
else{
    locationManager.requestLocationUpdates(bestProvider, 1000, 0,
      locationListener);
}
```

La classe Android `Address` viene utilizzata per creare un oggetto Address in cui salvare le informazioni sulla location per poterle poi memorizzare in “Place” (mentre in

location non serve, perché vengono memorizzate solo le coordinate, latitudine e longitudine, e viene tutto svolto in automatico da Geofire).

`address.get(0).getLocality()` viene utilizzato per memorizzare il nome della località, grazie ad Address sarebbe possibile salvare anche lo Stato o l'indirizzo vero e proprio.

Per quanto riguarda il salvataggio dell'immagine di profilo scelta dall'utente che sta effettuando la registrazione si deve invece creare un riferimento al Firebase Cloud Storage e salvare l'immagine nella cartella "ProfilePics":

```
private StorageReference mStorageRef;
private static final int PICK_IMAGE_REQUEST = 1;
private Uri mImageUri;

mStorageRef=FirebaseStorage.getInstance().getReference().child("Profil
ePics");
imageView.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        openFileChooser();
    }
});
```

openFileChooser() viene chiamato nel momento in cui l'utente vuole scegliere l'immagine da assegnare al proprio profilo. Viene aperto lo storage interno (solitamente la cartella "Downloads" del dispositivo e da lì l'utente può scegliere l'immagine e salvarla come immagine di profilo:

```
private void openFileChooser(){
    Intent intent = new Intent(Intent.ACTION_OPEN_DOCUMENT);
    intent.addCategory(Intent.CATEGORY_OPENABLE);
    intent.setType("image/*");
    startActivityForResult(intent, PICK_IMAGE_REQUEST);
}
@Override
protected void onActivityResult(int requestCode, int resultCode,
Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if(requestCode == PICK_IMAGE_REQUEST && resultCode == RESULT_OK &&
      data.getData() != null){
        mImageUri = data.getData();
        try {
            Bitmap bitmap =
            MediaStore.Images.Media.getBitmap(getContentResolver(),
            mImageUri);
```

```
            imageView.setImageBitmap(bitmap);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
reference.addValueEventListener(new ValueEventListener() {
    @Override
    public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
        getValues();
        if (mImageUri != null) {
            final StorageReference fileReference =
            mStorageRef.child(auth.getUid());
            fileReference.putFile(mImageUri)
                    .addOnSuccessListener(new
                        OnSuccessListener<UploadTask.TaskSnapshot>() {
                        @Override
                        public void onSuccess(final
                        UploadTask.TaskSnapshot taskSnapshot) {
                            getApplicationContext();
                            Toast.makeText(RegistrationActivity.this,
                              "Upload successful",
                              Toast.LENGTH_LONG).show();
                        }
                    })
                    .addOnFailureListener(new OnFailureListener() {
                        @Override
                        public void onFailure(@NonNull Exception e) {
                            Toast.makeText(RegistrationActivity.this,
                              e.getMessage(),
                              Toast.LENGTH_SHORT).show();
                        }
                    });
        } else {
            Toast.makeText(RegistrationActivity.this, "No file
            selected", Toast.LENGTH_SHORT).show();
        }
      reference.child(auth.getUid()).setValue(user);
```

Qui viene chiamato il currentUser (infatti l'id dell'utente che si sta registrando a questo punto è già stato creato) e si va a modificare il profilo dello user FirebaseUser, che com'era già stato definito nel capitolo precedente, presenta come campi, email, password e photoUrl:

```
        FirebaseUser ur = auth.getCurrentUser();
        UserProfileChangeRequest profileUpdates = new
UserProfileChangeRequest.Builder().setDisplayName(user.getName()).buil
d();
        UserProfileChangeRequest profileUpdates1 = new
UserProfileChangeRequest.Builder().setPhotoUri(mImageUri).build();
```

```
        ur.updateProfile(profileUpdates);
        ur.updateProfile(profileUpdates1);
```

Un nuovo oggetto della classe User viene definito e tutte le informazioni inserite dall'utente vengono salvate nell'oggetto che viene salvato nel database, poi si verifica se l'utente è professore o studente e si inizia una nuova Activity a seconda del valore di ritorno di "professor":

```
User user = new User();

private void getValues(){

    user.setName(etNickname.getText().toString());
    user.setEmail(etEmail.getText().toString());
    user.setUserId(auth.getUid());
    String deviceToken = FirebaseInstanceId.getInstance().getToken();
    user.setKey(deviceToken);
    if(professor.isChecked()){
        user.setProfessor(true);
    }
    else{
        user.setProfessor(false);
    }
}

...

if(user.isProfessor()){

    startActivity(new Intent(RegistrationActivity.this,
      StartProfessor.class));
    finish();
}
if(!user.isProfessor()){
    startActivity(new Intent(RegistrationActivity.this,
      StartStudent.class));
    finish();
}

...
```

In generale, come nel modello di codice che segue, la sintassi per scrivere e leggere dal Realtime Database di Firebase è:

```
// Write to the database

FirebaseDatabase database = FirebaseDatabase.getInstance();
DatabaseReference myRef = database.getReference("message");
myRef.setValue("Hello, World!");
```

```
// Read from the database
myRef.addValueEventListener(new ValueEventListener() {
 @Override
 public void onDataChange(DataSnapshot dataSnapshot) {
 // This method is called once with the initial value and again
 // whenever data at this location is updated.
 String value = dataSnapshot.getValue(String.class);
 Log.d(TAG, "Value is: " + value);
 }
 @Override
 public void onCancelled(DatabaseError error) {
 // Failed to read value
 Log.w(TAG, "Failed to read value.", error.toException());
 }
});
```

3.2.3 LoginActivity

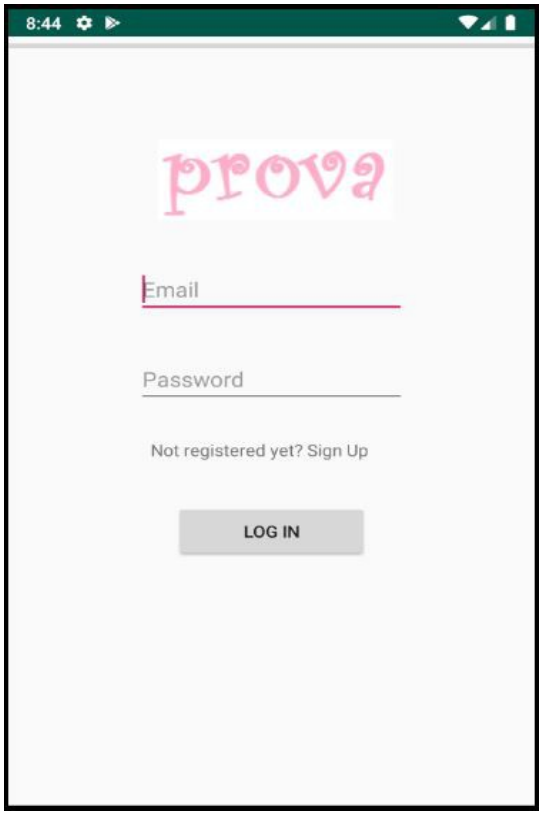

Figura 3.17: Schermata di Login

Nella schermata di Login l'utente può effettuare il login con le proprie credenziali o accedere alla pagina di registrazione nel caso non si sia ancora registrato. La MainActivity, che inizia l'esecuzione dell'applicazione, crea un Intent che fa partire proprio questa LoginActivity.
Innanzitutto bisogna recuperare l'id del currentUser (l'utente che effettua il login), poi si può procedere con l'invocazione della funzione `signInWithEmailAndPassword()` di FirebaseAuth:

```
private FirebaseAuth mAuth;
mAuth = FirebaseAuth.getInstance();


mAuth.signInWithEmailAndPassword(email,
password).addOnCompleteListener(new OnCompleteListener<AuthResult>() {
    @Override
    public void onComplete(@NonNull Task<AuthResult> task) {
        if (task.isSuccessful()) {
            String currentUserID = mAuth.getCurrentUser().getUid();
            String deviceToken =
FirebaseInstanceId.getInstance().getToken();
```

Dopodiché si procede ad ordinare gli oggetti di User per professore e a prendere l'oggetto le cui informazioni equivalgono a quelle del currentUser. In caso contrario, le informazioni inserite sono scorrette, e viene visualizzato un Toast di errore:

```
FirebaseUser currentUser = FirebaseAuth.getInstance().getCurrentUser();
final String RegisteredUserID = currentUser.getUid();
databaseReferenceToken = FirebaseDatabase.getInstance()
                    .getReference().child("User")
                    .child(currentUserID).child("key")
                    .setValue(deviceToken)
                    .addOnCompleteListener(new
OnCompleteListener<Void>() {
@Override
 public void onComplete(@NonNull Task<Void> task) {
      if(task.isSuccessful()){
            mDatabase = FirebaseDatabase.getInstance().
            getReference("User").child(RegisteredUserID);
            mDatabase.orderByChild("professor");
            mDatabase.addValueEventListener(new ValueEventListener() {
            @Override
            public void onDataChange(@NonNull DataSnapshot
dataSnapshot) {
```

Infine si verifica se l'utente è professore o studente e come nella registrazione si attiva l'Activity corrispondente.

```
String userType;
userType=dataSnapshot.child("professor").getValue().toString();
if(userType.equals("true")){
   startActivity(new Intent(LoginActivity.this, StartProfessor.class));
}
else if(userType.equals("false")){
   startActivity(new Intent(LoginActivity.this, StartStudent.class));
}
else{
   Toast.makeText(LoginActivity.this, "Failed Login. Please Try
Again", Toast.LENGTH_SHORT).show();
   return;
}
```

3.2.4 StartStudent

In questa classe viene implementata la prima schermata che si prospetta allo studente nel momento in cui effettua il login. Lo studente può scegliere di effettuare la propria ricerca filtrando per materia e per luogo (vedi Figura 3.18 e 3.19):

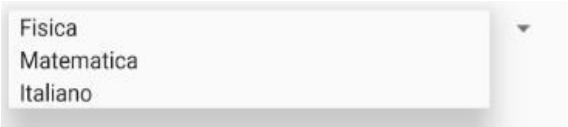

Figura 3.19: Spinner con materie

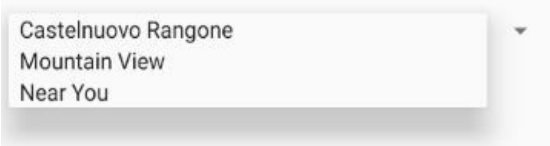

Figura 3.18: Spinner con luoghi

I due spinner vengono popolati dai dati d'esempio del database, che sono stati utilizzati durante la fase di sviluppo dell'applicazione. Nel caso in cui si scelga come luogo “Near You” verrà ricercata la posizione attuale dello studente, che viene aggiornata nel database (in “location”) ogni volta che un utente va online.

Nel momento in cui uno studente seleziona la materia e il luogo, nel database viene aggiornata Choose con i nuovi dati. Per esempio se lo studente “studente3@g.com” cercasse ripetizioni di Fisica vicino a lui, il database risulterebbe così aggiornato:

Figura 3.20: Esempio di ricerca studente

Nel database questo studente risulta alle coordinate riportate in Figura 3.21, che sono le stesse, come si può vedere dell'utente con ID “K9oZIxOqNJR27AJoFRa0fjlOh7q1”:

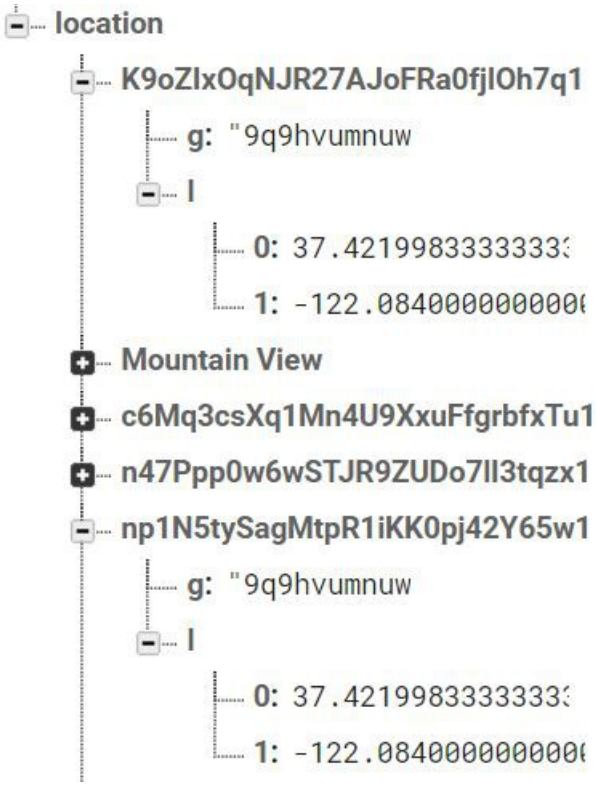

Figura 3.21: Location dello studente

L'altro utente con tali coordinate risulta essere un professore e risulta insegnare Fisica:

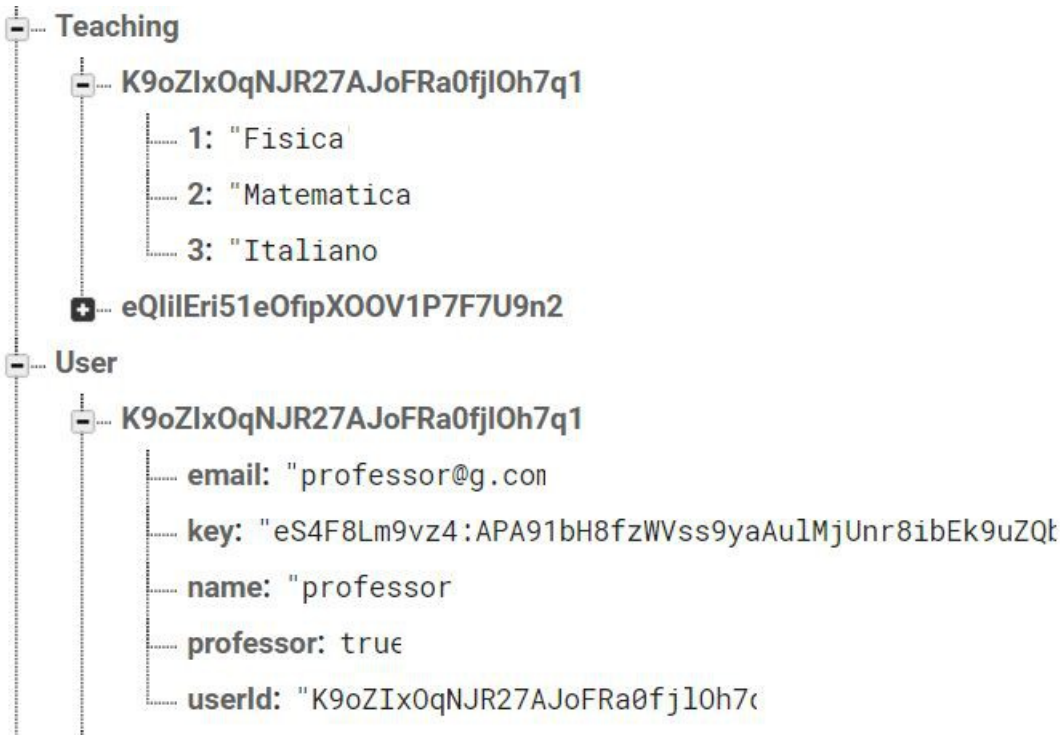

Figura 3.22: Teaching e User

Quindi, essendoci almeno un professore che corrisponde alle caratteristiche richieste dallo studente, quest'ultimo potrà passare alla nuova schermata, implementata in HomeStudent.java, con la lista dei professori trovati, che per finire il nostro esempio risulterà come in Figura 3.23:

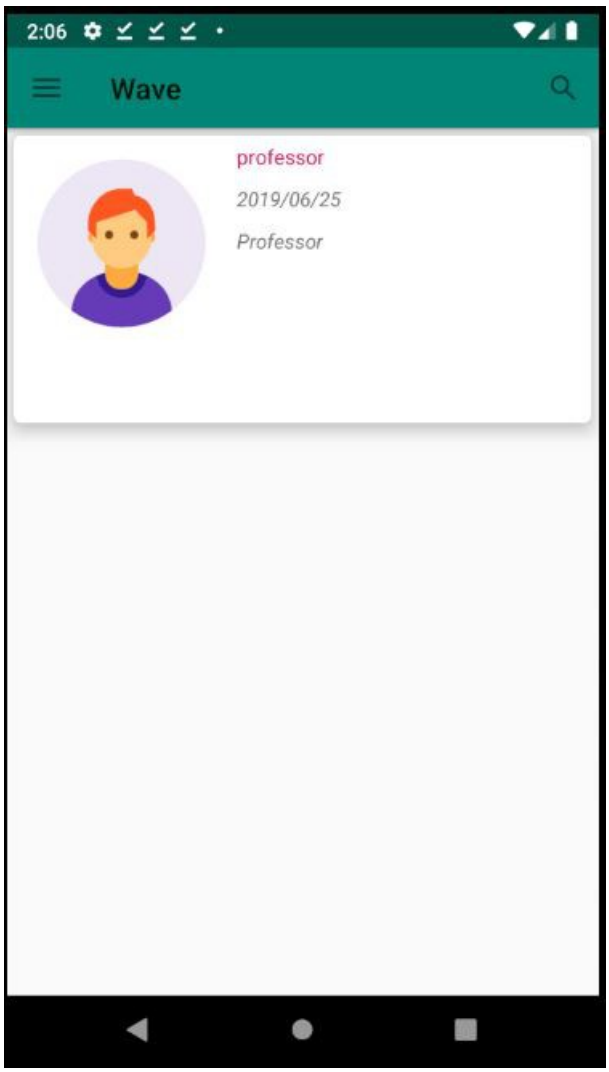

Figura 3.23: Lista professori risultante dalla ricerca dello studente

Di seguito viene riportato il codice scritto per realizzare le funzionalità spiegate sopra. Innanzitutto per quanto riguarda lo spinner delle materie, è stata utilizzata la classe Spinner e un Adapter per poterne gestire la visualizzazione. Dopo aver interrogato il database, lo spinner è stato popolato creando una lista con gli oggetti del database:

```
Spinner spnSubject;
DatabaseReference ref, choose;
List<String> subjects;

spnSubject = findViewById(R.id.spnSubject);
choose = FirebaseDatabase.getInstance().getReference("Choose");
```

```
subjects = new ArrayList<>();


ref = FirebaseDatabase.getInstance().getReference("Subject");
ref.addValueEventListener(new ValueEventListener() {
    @Override
    public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
        subjects.clear();
        for(DataSnapshot subjectSnapshot : dataSnapshot.getChildren())
            {
            String upload =
            subjectSnapshot.getValue(Subject.class).getName();
            subjects.add(upload);
        }
        spinnerAdapter = new ArrayAdapter(StartStudent.this,
            android.R.layout.simple_spinner_item, subjects);
        spnSubject.setAdapter(spinnerAdapter);
        spnSubject.setOnItemSelectedListener(new
            AdapterView.OnItemSelectedListener() {
            @Override
            public void onItemSelected(AdapterView<?> parent, View
            view, final int position, long id) {
                spnChoose =
                  spnSubject.getItemAtPosition(position).toString();
                choose.addListenerForSingleValueEvent(new
                  ValueEventListener() {
                    @Override
                    public void onDataChange(@NonNull DataSnapshot
                        snap) {
                        try {
                              choose.child(mAuth.getUid()).
                              child("subject").setValue(spnChoose);
                        }catch (DatabaseException e){
                            e.printStackTrace();
                        }
                    }
                    @Override
                    public void onCancelled(@NonNull DatabaseError
                        databaseError) {
                    }
                });
            }
            @Override
            public void onNothingSelected(AdapterView<?> parent) {
            }
        });
    }
    @Override
    public void onCancelled(@NonNull DatabaseError databaseError) {
    }
});
```

Poi è stato creato alla stessa maniera lo Spinner per i luoghi. Come spiegato sopra è stato interrogato il database per determinare i luoghi da inserire tra le scelte nello

Spinner, poi è stata aggiornata Choose con la scelta effettuata dallo studente:

```
placeRef.addValueEventListener(new ValueEventListener() {
    @Override
    public void onDataChange(@NonNull DataSnapshot plSnapshot) {
        places.clear();
        for (DataSnapshot placesSnapshot : plSnapshot.getChildren()) {
            String upload1 =
            placesSnapshot.getValue(Place.class).getName();
            places.add(upload1);
        }
        places.add("Near You");
        spinnerAdapter2 = new ArrayAdapter(StartStudent.this,
            android.R.layout.simple_spinner_item, places);
        spnPlace.setAdapter(spinnerAdapter2);
        spnPlace.setOnItemSelectedListener(new
            AdapterView.OnItemSelectedListener() {
            @Override
            public void onItemSelected(AdapterView<?> parent, View
            view, final int position, long id) {
                spnChoose2 =
                  spnPlace.getItemAtPosition(position).toString();
                choose.addListenerForSingleValueEvent(new
                  ValueEventListener() {
                    @Override
                    public void onDataChange(@NonNull DataSnapshot
                        snap) {
                        try {
                            choose.child(mAuth.getUid()).
                              child("place").setValue(spnChoose2);
                        }catch (DatabaseException e){
                            e.printStackTrace();
                        }
                    }
                    @Override
                    public void onCancelled(@NonNull DatabaseError
                        databaseError) {
                    }
                });
            }
            @Override
            public void onNothingSelected(AdapterView<?> parent) {
            }
        });
    }
    @Override
    public void onCancelled(@NonNull DatabaseError databaseError) {
    }
});
btnGO.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        startActivity(new Intent(StartStudent.this,
HomeStudent.class));
    }
});
```

3.2.5 StartProfessor

In questa pagina i professori, dopo aver effettuato il login, mettono una spunta alle materie che insegnano. Tali scelte vengono inserite nel database in “Teaching” . In questo caso la lista delle materie, lette da “Subject”, viene inserita in un Checkbox (ad ogni item del checkbox corrisponde una materia), invece che in uno Spinner, risultando come in Figura 3.24:

Figura 3.24: Materie del professore

In codice tutto ciò risulta:

```
public class StartProfessor extends AppCompatActivity {
    public class Item {
        boolean checked;
        String ItemString;
        String id;
        Item (){}
        Item(String t, boolean b, String id){
            ItemString = t;
            checked = b;
            id = id;
        }
        public boolean isChecked(){
            return checked;
        }
    }
    static class ViewHolder {
        CheckBox checkBox;
        TextView text;
    }
    public class ItemsListAdapter extends BaseAdapter {
        private Context context;
        private List<Item> list;
        ItemsListAdapter(Context c, List<Item> l) {
            context = c;
            list = l;
        }
        @Override
        public int getCount() {
            return list.size();
        }
        @Override
```

```
        public Object getItem(int position) {
            return list.get(position);
        }
        @Override
        public long getItemId(int position) {
            return position;
        }
        public boolean isChecked(int position) {
            return list.get(position).checked;
        }
        @Override
        public View getView(final int position, View convertView,
            ViewGroup parent) {
            View rowView = convertView;
            // reuse views
            ViewHolder viewHolder = new ViewHolder();
            if (rowView == null) {
                LayoutInflater inflater = ((Activity)
                  context).getLayoutInflater();
                rowView = inflater.inflate(R.layout.row_checkbox,
                  null);
                viewHolder.checkBox =
                  rowView.findViewById(R.id.rowCheckBox);
                viewHolder.text =
                  rowView.findViewById(R.id.rowTextView);
                rowView.setTag(viewHolder);
            } else {
                viewHolder = (ViewHolder) rowView.getTag();
            }
            viewHolder.checkBox.setChecked(list.get(position).checked);
            final String itemStr = list.get(position).ItemString;
            viewHolder.text.setText(itemStr);
            viewHolder.checkBox.setTag(position);
            viewHolder.checkBox.setOnClickListener(new
            View.OnClickListener() {
                @Override
                public void onClick(View view) {
                    boolean newState = !list.get(position).isChecked();
                    list.get(position).checked = newState;
                    Toast.makeText(getApplicationContext(),
                            itemStr + "setOnClickListener\nchecked: "+
                              newState,
                            Toast.LENGTH_LONG).show();
                }
            });
            viewHolder.checkBox.setChecked(isChecked(position));
            return rowView;
        }
    }
    Button btnLookup;
    List<Item> items;
    ListView listView;
    ItemsListAdapter myItemsListAdapter;
    DatabaseReference reference;
    Item item;
    FirebaseDatabase database = FirebaseDatabase.getInstance();
```

```
    DatabaseReference refTeach;
    FirebaseAuth auth;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_start_professor);
        listView = (ListView)findViewById(R.id.listSubject);
        btnLookup = (Button)findViewById(R.id.lookup);
        items = new ArrayList<>();
        auth = FirebaseAuth.getInstance();
        refTeach =
            database.getReference("Teaching").child(auth.getUid());
        reference =
            FirebaseDatabase.getInstance().getReference("Subject");
        reference.addValueEventListener(new ValueEventListener() {
            @Override
            public void onDataChange(@NonNull DataSnapshot
            dataSnapshot) {
                items.clear();
                for (DataSnapshot subjectSnapshot :
                  dataSnapshot.getChildren()) {
                    final Subject upload =
                        subjectSnapshot.getValue(Subject.class);
                    item = new Item();
                    item.ItemString = upload.getName();
                    item.checked = false;
                    item.id = upload.getId();
                    items.add(item);
                    item = null;
                }
                myItemsListAdapter = new
                  ItemsListAdapter(StartProfessor.this, items);
                listView.setAdapter(myItemsListAdapter);
                myItemsListAdapter.notifyDataSetChanged();
            }
            @Override
            public void onCancelled(@NonNull DatabaseError
            databaseError) {
            }
        });
        listView.setOnItemClickListener(new
            AdapterView.OnItemClickListener(){
            @Override
            public void onItemClick(AdapterView<?> parent, View view,
                                    int position, long id) {
                Toast.makeText(StartProfessor.this,
                        ((Subject)
                        (parent.getItemAtPosition(position))).
                        getName(),Toast.LENGTH_LONG).show();
            }});
        btnLookup.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                for (int i=0; i<items.size(); i++){
                    if (items.get(i).isChecked()){
```

```
refTeach.child(items.get(i).id).setValue(items.get(i).ItemString);
                        //refTeach.setValue(items.get(i).ItemString);
                    }
                }
                startActivity(new Intent(StartProfessor.this,
                  HomeProfessor.class));
            }
        });
    }
}
```

Come vedremo in seguito un professore può cambiare le materie insegnate nella sezione apposita del drawer dell'applicazione (così come uno studente può cambiare la ricerca).

3.2.6 HomeStudent

Dopo aver effettuato il login lo studente può accedere alla lista dei professori che corrispondono alla ricerca effettuata. In HomeStudent quindi viene elaborata la query dello studente, partendo dalla scelta dei professori per località. Vengono effettuati due controlli a riguardo, a seconda che lo studente abbia scelto “Near You” o il nome di una località:

```
if(dataSnapshot.child(mAuth.getUid()).child("place").getValue().equals(
"Near You"))
else ...
```

all'inizio di questa classe infatti viene aggiornata la posizione dell'utente, in modo da poter effettuare la query di ricerca dei professori nel caso sia stato scelto “Near You”:

```
locationManager = this.getSystemService(Context.LOCATION_SERVICE);
locationListener = new LocationListener() {
    @Override
    public void onLocationChanged(Location location) {
    }
    @Override
    public void onStatusChanged(String provider, int status, Bundle
extras) {
    }
    @Override
    public void onProviderEnabled(String provider) {
    }
    @Override
    public void onProviderDisabled(String provider) {
    }
};
criteria = new Criteria();
bestProvider =
String.valueOf(locationManager.getBestProvider(criteria, true));
if (ActivityCompat.checkSelfPermission(HomeStudent.this,
Manifest.permission.ACCESS_FINE_LOCATION) !=
PackageManager.PERMISSION_GRANTED &&
ActivityCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_COARSE_LOCATION) !=
PackageManager.PERMISSION_GRANTED) {
    // TODO: Consider calling
    //    ActivityCompat#requestPermissions
    // here to request the missing permissions, and then overriding
    //   public void onRequestPermissionsResult(int requestCode,
String[] permissions,
    //                                          int[] grantResults)
    // to handle the case where the user grants the permission. See
the documentation
    // for ActivityCompat#requestPermissions for more details.
    return;
}
location = locationManager.getLastKnownLocation(bestProvider);
```

```
if (location != null) {
    final Double latitude = location.getLatitude();
    final Double longitude = location.getLongitude();
    DatabaseReference current_user_db =
      FirebaseDatabase.getInstance().getReference().child("location");
    final GeoFire geoFire2 = new GeoFire(current_user_db);
    geoFire2.removeLocation(mAuth.getUid(), new
      GeoFire.CompletionListener() {
        @Override
        public void onComplete(String key, DatabaseError error) {
            geoFire2.setLocation(mAuth.getUid(), new
            GeoLocation(latitude, longitude),new
                    GeoFire.CompletionListener(){
                        @Override
                        public void onComplete(String key,
                        DatabaseError error) {
                            //Do some stuff if you want to
                        }
                    });
        }
    });
}else{
    locationManager.requestLocationUpdates(bestProvider, 1000, 0,
      locationListener);
}
```

Dopo aver controllato la scelta dello studente, nel corpo dell'if e dell'else si va a cercare nel database, in “location”, il luogo e si controlla quali utenti, professori, abbiano coordinate corrispondenti a tale luogo:

```
DatabaseReference current_user_db =
FirebaseDatabase.getInstance().getReference().child("location");
final GeoFire geoFire = new GeoFire(current_user_db);
```

Viene creata una geoQuery per trovare quegli utenti che si trovano in coordinate a x metri (in questo caso 10) dalla location con location.latitude e location.longitude :

```
geoFire.getLocation(input, new LocationCallback() {

//input è mAuth.getUid() nel caso dell'if e String name_place nel caso
dell'else

    @Override
    public void onLocationResult(String key, GeoLocation location) {
        if (location != null) {
            GeoQuery geoQuery = geoFire.queryAtLocation(new
            GeoLocation(location.latitude, location.longitude), 10);
            geoQuery.removeAllListeners();
            geoQuery.addGeoQueryDataEventListener(new
            GeoQueryDataEventListener() {
```

```
                @Override
                public void onDataEntered(final DataSnapshot
                  locSnapshot, GeoLocation location) {
                    if (locSnapshot.exists()) {
                        final ArrayList<String> arrayListUserIds = new
                        ArrayList<>();
                        arrayListUserIds.add(locSnapshot.getKey());
```

Dopo aver creato un ArrayList contenente gli utenti di quella location si va a controllare in “User” quali di questi siano professori e si aggiungono alla lista mUsers:

```
mDatabaseRef.addValueEventListener(new ValueEventListener() {
      @Override
      public void onDataChange(@NonNull
      DataSnapshot dataSnapshot) {
            for(DataSnapshot teacherSnapshot :
            dataSnapshot.getChildren()) {
                  for(int i=0; i<arrayListUserIds.size(); i++) {
                        if (arrayListUserIds.get(i).
                        equals(teacherSnapshot.getKey())) {
                              final User upload =
                              teacherSnapshot.getValue(User.class);
                                    if (upload.isProfessor()) {
                              upload.setKey(teacherSnapshot.getKey());
                                          mUsers.add(upload);
                                     }
                        }
                  }
             }
      }
}
```

In seguito vado a controllare in “Teaching” quali professori insegnano le materie scelte dallo studente e li inserisco nell'ArrayList temp:

```
if(mUsers.size()>0){
      chooseRef.addValueEventListener(new ValueEventListener() {
      @Override
      public void onDataChange(@NonNull final DataSnapshot
      chooseSnapshot) {
            refProf.addValueEventListener(new ValueEventListener() {
                  @Override
                  public void onDataChange(@NonNull DataSnapshot
                  refSnapshot) {
                        final ArrayList temp = new ArrayList();
                        for(DataSnapshot ref:
                        refSnapshot.getChildren()){
                              for(DataSnapshot ref2:ref.getChildren()){
                                    if(ref2.getValue().
                                    equals(chooseSnapshot.
                                    child(mAuth.getUid()).
                                    child("subject").getValue())){
```

```
                                        temp.add(ref.getKey());
                                    }
                                }
                    ...}
```

[7]

Ora che ho temp e mUsers posso finalmente trovare quali professori insegnano nell'area richiesta dallo studente (le materie che ha scelto), mettendo nella lista profOk gli utenti che si trovano sia in temp che in mUsers:

```
profOk.clear();
for (int i = 0; i < mUsers.size(); i++) {
      for(int j=0; j<temp.size(); j++) {
            if (mUsers.get(i).getKey().equals(temp.get(j))) {
                  final User upload3 = mUsers.get(i);
                  profOk.add(upload3);
            } else {
                  System.out.println("NO match");
            }
      }
}
```

Infine lo studente può visualizzare gli utenti trovati come descritto sopra grazie alla RecyclerView:

```
mAdapter = new RecyclerAdapter(HomeStudent.this, profOk);

mRecyclerView.setAdapter(mAdapter);

mAdapter.setOnItemClickListener(HomeStudent.this);
 mAdapter.notifyDataSetChanged();
```

Si è optato per il RecyclerView, al posto del ListView, in quanto risulta essere molto più efficiente e vantaggioso rispetto al suo predecessore. Infatti il RecyclerView utilizza un meccanismo di riciclo del ViewHolder: Anziché creare tutti gli elementi della lista durante lo scroll, esso mantiene in una coda, chiamata *recycler bin*, alcuni degli elementi precedentemente visualizzati per riutilizzarli in un secondo momento. Infatti, durante lo scorrimento della lista, il RecyclerView recupererà dalla coda un elemento e lo popolerà con le nuove informazioni da mostrare all'utente.

Inoltre con il ListView non è supportato lo scroll orizzontale e verticale, e vi è una peggiore occupazione della memoria perché appunto non è presente il riciclo del ViewHolder.[8]

I componenti del RecyclerView sono:

Componente	Tipologia	Descrizione
Adapter	RecyclerView.Adapter	È responsabile di estrarre i dati dal Data Source e di usare questi dati per creare e popolare i ViewHolder. Quest'ultimi saranno poi inviati al Layout Manager del RecyclerView.Adapter.
ViewHolder	RecyclerView.ViewHolder	È la chiave di volta tra il RecyclerView e l'Adapter e permette la riduzione nel numero di view da creare. Questo oggetto infatti fornisce il layout da popolare con i dati presenti nel DataSource e viene riutilizzato dal RecyclerView per ridurre il numero di layout da creare per popolare la lista.
Layout Manager	RecyclerView.LayoutManager	È responsabile della creazione e del posizionamento delle view all'interno del RecyclerView. Esistono diverse tipologie di LayoutManager come il LinearLayoutManager utilizzato per creare liste orizzontali o verticali.
DataSource	List	È l'insieme di dati utilizzato per popolare la lista tramite l'Adapter

Di seguito invece è stata riportata la classe utilizzata per personalizzare l'Adapter del RecyclerView:

```
public  class RecyclerAdapter extends
RecyclerView.Adapter<RecyclerAdapter.RecyclerViewHolder> implements
Filterable {
    private Context mContext;
    private List<User> users;
    private List<User> filteredUsers;
//lista contenente gli utenti filtrati
    private OnItemClickListener mListener;
    public RecyclerAdapter(Context context, List<User> uploads) {
        mContext = context;
        users = uploads;
        filteredUsers = uploads;
    }
    @Override
    public RecyclerViewHolder onCreateViewHolder(ViewGroup parent, int
      viewType) {
        View v =
      LayoutInflater.from(mContext).inflate(R.layout.row_model, parent,
            false);
        return new RecyclerViewHolder(v);
    }
    @Override
    public void onBindViewHolder(final RecyclerViewHolder holder,
      final int position) {
```

```
//qui vengono specificate quali informazioni degli user nella lista
verranno mostrate nel ViewHolder
        User currentUser = filteredUsers.get(position);
        holder.nameTextView.setText(currentUser.getName());
        if(currentUser.isProfessor()){
            holder.descriptionTextView.setText("Professor");
        }
        if(!currentUser.isProfessor()){
            holder.descriptionTextView.setText("Student");
        }
        holder.dateTextView.setText(getDateToday());
        final StorageReference ref =
            FirebaseStorage.getInstance().getReference().
            child("ProfilePics").child(currentUser.getUserId());
        ref.getDownloadUrl().addOnSuccessListener(new
            OnSuccessListener<Uri>() {
            @Override
            public void onSuccess(Uri uri) {
                Picasso.get()
                        .load(uri)
                        .placeholder(R.drawable.placeholder)
                        .fit()
                        .centerCrop()
                        .into(holder.teacherImageView);
            }
        });
    }
```

Questo di seguito è il codice scritto per implementare il filtro utilizzato per aggiornare con gli utenti filtrati il RecyclerView quando l'utente effettua la ricerca (come in Figura 3.26):

```
//OVERRIDE ADAPTER METHODS
@Override
public int getItemCount() {
      return filteredUsers.size();
}
@Override
public long getItemId(int position){
      return position;
}
@Override
public Filter getFilter(){
      return new Filter() {
      @Override
      protected FilterResults performFiltering(CharSequence
      charSequence) {
            FilterResults results = new FilterResults();
            if(charSequence == null || charSequence.length() == 0){
                  results.values = users;
                  results.count = users.size();
                  }else{
```

```
                    filteredUsers = new ArrayList<User>();
                    for(int i=0; i<users.size();i++){
                       if(users.get(i).getName().
toUpperCase().contains( charSequence.toString().toUpperCase() ))
                            filteredUsers.add(users.get(i));
                    }
                    results.values = filteredUsers;
                    results.count = filteredUsers.size();
                }
                return results;
            }
            @Override
            protected void publishResults(CharSequence constraint,
            FilterResults results) {
                /**We just reset the data set to the filtered list.
                  The mList variable is used
                 * to get data for each view in getView() method of
                  the adapter.
                 * As soon as the data set is changed, we call
                  notifyDataSetChanged() to refresh the Views
                 */
                //filteredUsers = (List<User>) results.values;
                notifyDataSetChanged();
            }
        };
    }
```

Mentre il ViewHolder viene creato così:

```
public class RecyclerViewHolder extends RecyclerView.ViewHolder
implements View.OnClickListener {
        public TextView nameTextView,descriptionTextView,dateTextView;
        public ImageView teacherImageView;
        public RecyclerViewHolder(View itemView) {
            super(itemView);
            nameTextView =itemView.findViewById ( R.id.nameTextView );
            descriptionTextView =
            itemView.findViewById(R.id.descriptionTextView);
            dateTextView = itemView.findViewById(R.id.dateTextView);
            teacherImageView =
            itemView.findViewById(R.id.teacherImageView);
            itemView.setOnClickListener(this);
        }
        @Override
        public void onClick(View v) {
            if (mListener != null) {
                int position = getAdapterPosition();
                if (position != RecyclerView.NO_POSITION) {
                    mListener.onItemClick(position);
                }
            }
        }
    }
```

La lista si presenta quindi in questa maniera agli utenti-studenti (Figura 3.25):

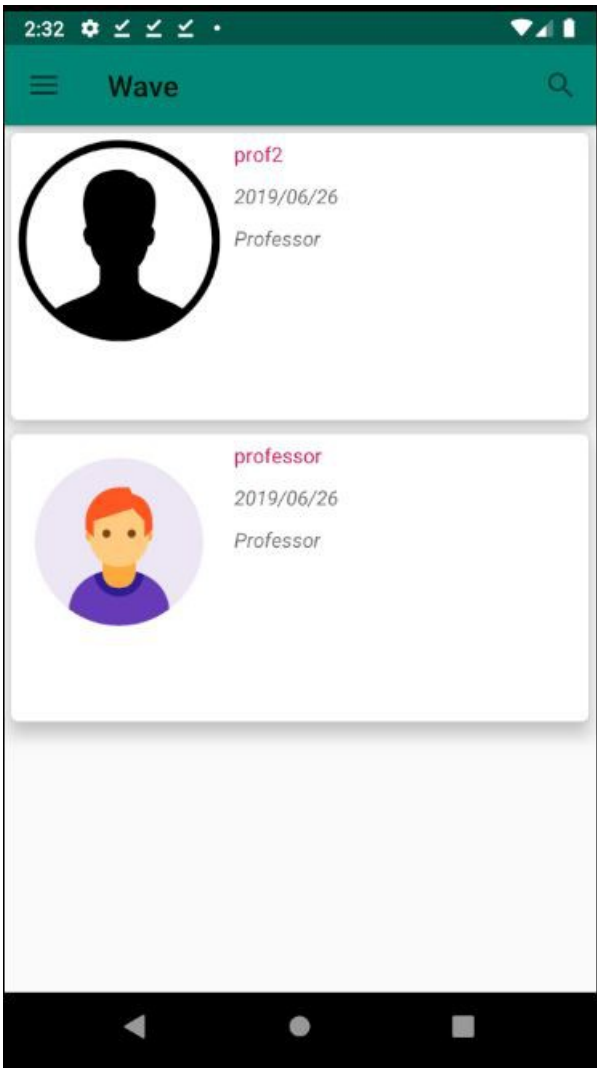

Figura 3.25: Lista professori

Come si può notare dalla Figura 3.25 è possibile effettuare una ricerca sulla lista dei professori, interagendo con l'icona in alto a destra:

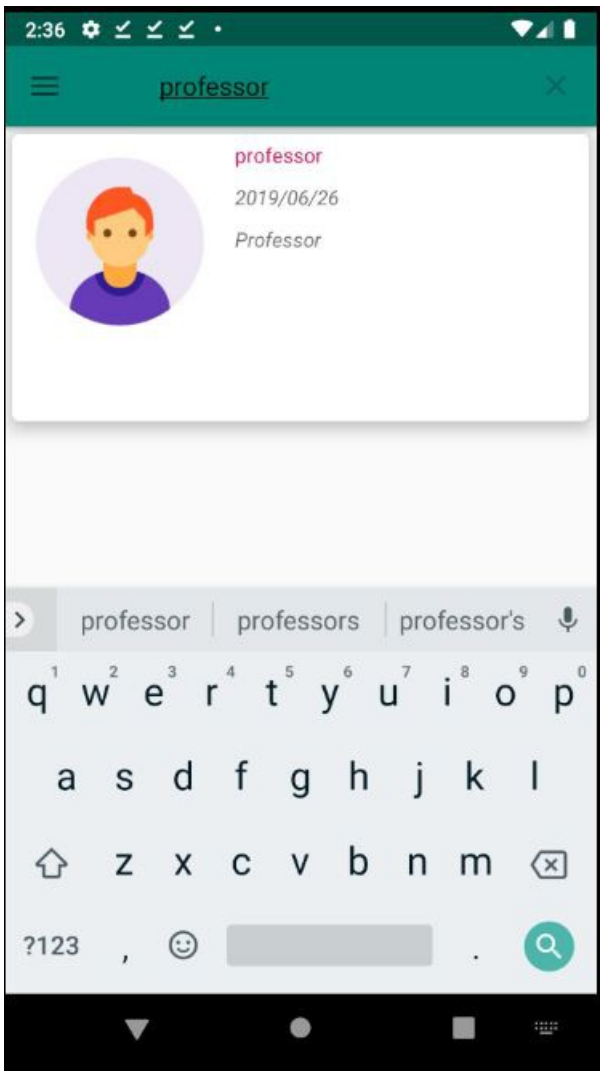

Figura 3.26: Lista professori con ricerca

Interagendo con l'icona in alto a sinistra invece viene aperto il Drawer (Figura 3.27), da cui è possibile accedere a:

- Profile: sezione contenente le informazioni del proprio profilo, e dalla quale è possibile modificare i propri dati;
- Contacts: lista dei contatti;
- Choose: riporta alla schermata implementata in StartStudent, nel caso si voglia effettuare una nuova ricerca;
- Requests: sezione contenente la lista delle richieste di contatto.

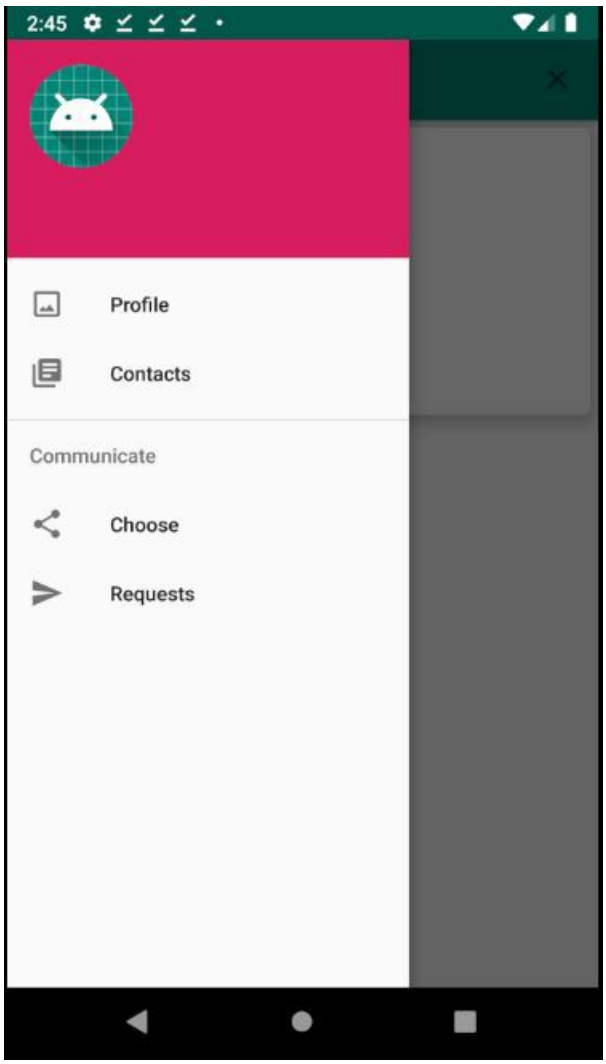

Figura 3.27: Drawer

3.2.7 DetailActivity

Data la lista dei professori, l'utente può visualizzare tutte le informazioni relative al professore con cui ha scelto di mettersi in contatto, scegliendolo dalla lista.
Ad esempio in Figura 3.28:

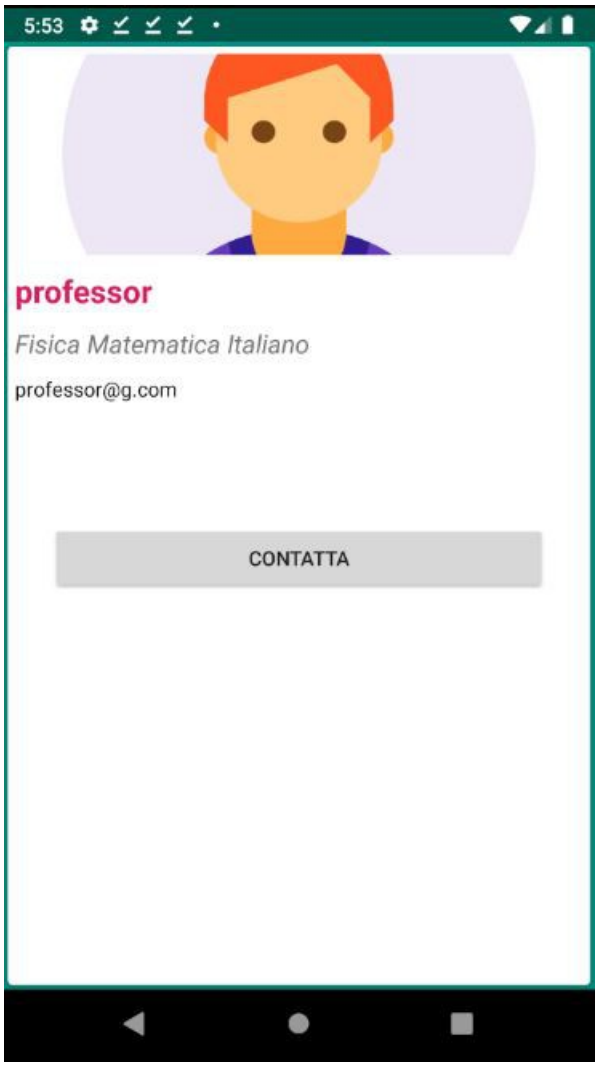

Figura 3.28: Dettagli professore

Attraverso l'interazione con il bottone "contatta" è possibile inviare una richiesta di contatto al professore in questione. Il professore riceverà una notifica (Figura 3.29) e potrà controllare le proprie richieste di contatto nella sezione "Requests" nel Drawer. A questo punto potrà decidere se accettare o meno la richiesta:

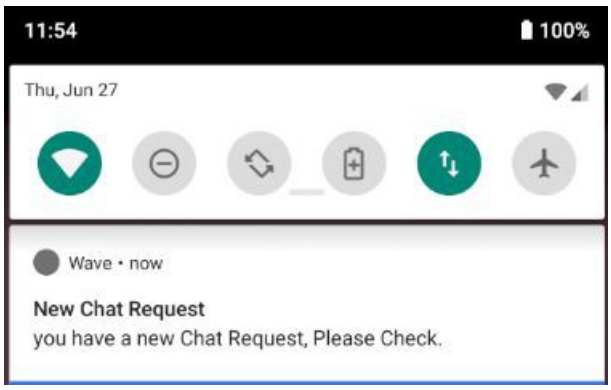

Figura 3.29: Notifica

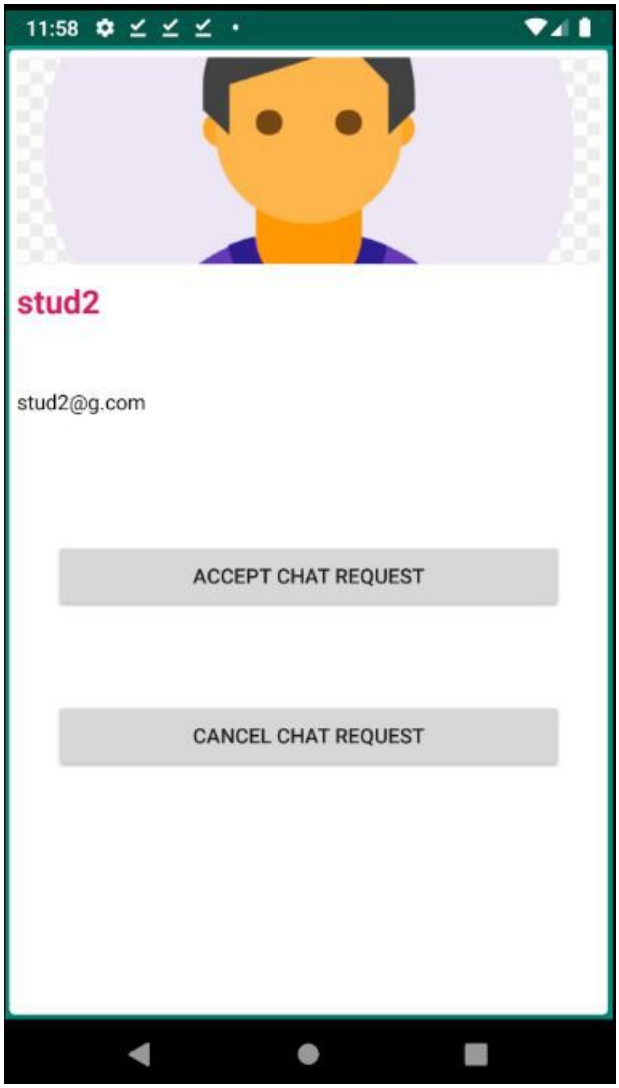

Figura 3.30: Accetta/Rifiuta Chat Request

Per inviare una notifica all'utente che riceve la richiesta di contatto, è stato creato un

campo “Notifications” nel database, che viene aggiornato ogni volta che viene creata una notifica:

```
HashMap<String,String> chatNotificationMap = new HashMap<>();

chatNotificationMap.put("from",senderUserID);
chatNotificationMap.put("type","request");
NotificationRef.child(receiverUserID).push()
.setValue(chatNotificationMap)
```

Inoltre per ogni utente che si registra viene salvato il device-token identificativo del dispositivo da cui si sta effettuando la registrazione all'app. Ogni volta che un utente effettua il login viene aggiornato tale token:

```
databaseReferenceToken = FirebaseDatabase.getInstance()

        .getReference().child("User")
        .child(currentUserID).child("key")
        .setValue(deviceToken)
```

Il device token viene poi utilizzato nello script perché la notifica viene inviata da device a device, quindi lo script deve conoscere l'ID del device a cui andrà inviata la notifica.

Lo script in node.js è in “index.js”:

```
'use strict'

const functions = require('firebase-functions');
const admin = require('firebase-admin');
admin.initializeApp(functions.config().firebase);

exports.sendNotification    =    functions.database.ref('/Notifications/
{receiver_user_id}/{notification_id}')
.onWrite((data, context) =>
{
      const receiver_user_id = context.params.receiver_user_id;
      const notification_id = context.params.notification_id;

      const   sender_user_id   =   admin.database().ref(`/Notifications/$
      {receiver_user_id}/${notification_id}`).once('value');
      return sender_user_id.then(fromUserResult =>
            {
                  const from_sender_user_id=fromUserResult.val().from;
                  console.log('You have a notification from :' ,
                  sender_user_id);
                  const userQuery = admin..database().ref(`/Users/$
                  {receiver_user_id}/key`).once('value');
                  return userQuery.then(userResult =>
                        {
                              const senderUserName = userResult.val();
                        });
            });
      const DeviceToken = admin.database().ref(`/User/$
      {receiver_user_id}/key`).once('value');
```

```
        return DeviceToken.then(result =>
        {
                const token_id = result.val();
                const payload =
                {
                        notification:
                        {
                                title: "New Chat Request",
                                body: `you have a new Chat Request, Please
                                Check.`,
                                icon: "default"
                        }
                };
                return admin.messaging().sendToDevice(token_id, payload)
                .then(response =>
                        {
                                console.log('This was a notification
                                feature.');
                        });
        });
});
```

Nel database quindi risulterà un oggetto del tipo nella Figura 3.31, con l'ID dello user che ha ricevuto la notifica, l'ID della notifica e i campi “from” e “type”:

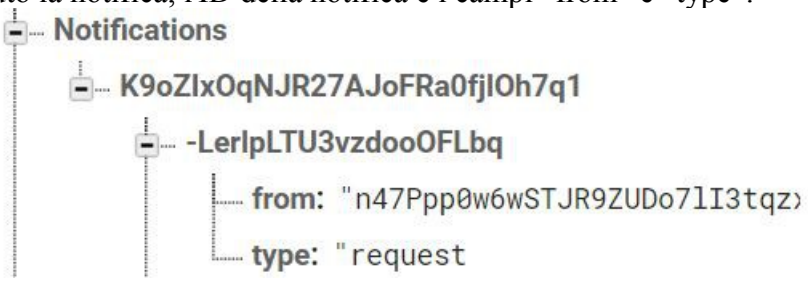

Figura 3.31: Notifica nel database

Infine andrà aggiunta la dipendenza nel file “build.gradle” dell'app

```
implementation 'com.google.firebase:firebase-messaging:18.0.0'
```

e grazie allo strumento “Functions” di Firebase sarà possibile controllare lo stato della funzione appena scritta, accedendo alle sezioni “Log”, “Salute” e “Utilizzo” (in Figura 3.32 è stata riportata la sezione di Log):

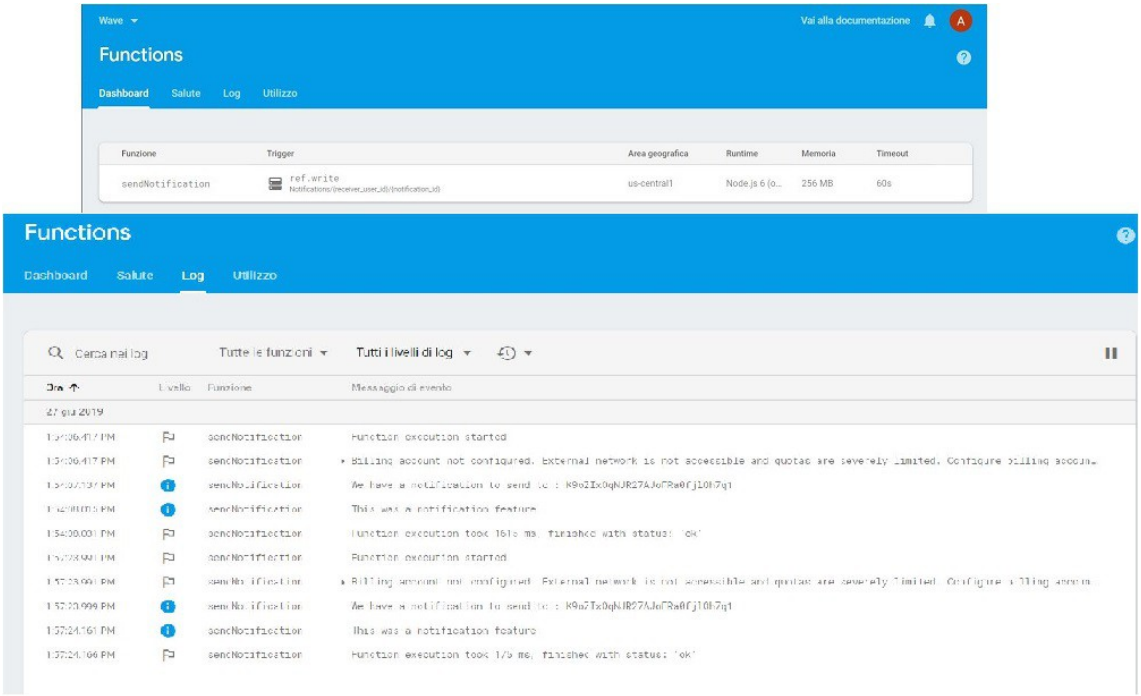

Figura 3.32: Sezione Log

Tornando alla DetailActivity, dopo aver accettato una richiesta di contatto, è sempre possibile rimuovere il contatto (come in Figura 3.33) :

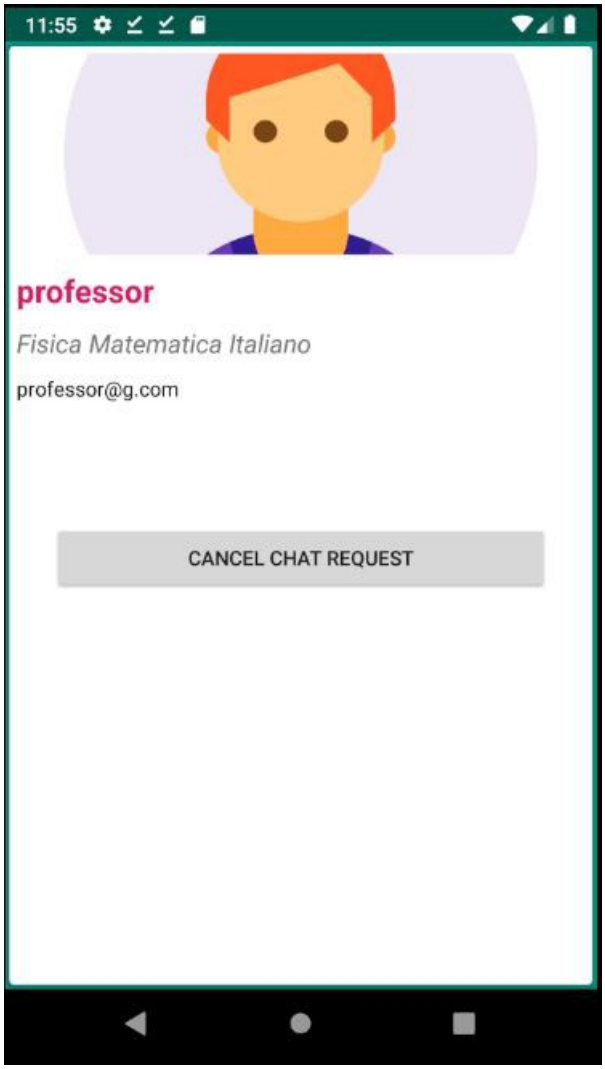

Figura 3.33: Rimuovi contatto

Tutto ciò viene implementato definendo due utenti:

- l'utente ricevente, ovvero l'utente di cui si stanno guardando i dettagli di profilo,
- l'utente mittente, ovvero l'utente attuale, che sta inviando la richiesta di contatto al ricevente.

Questi due utenti vengono quindi inseriti nel database in “ChatRequests”. Per quanto riguarda la gestione della richiesta è stata creata una funzione ManageChatRequest:

```
public void ManageChatRequest(){
    ChatRequestRef.child(senderUserID)
```

```
            .addValueEventListener(new ValueEventListener() {
                @Override
                public void onDataChange(@NonNull DataSnapshot
                  dataSnapshot) {
                    if(dataSnapshot.hasChild(receiverUserID)){
                        String request_type = dataSnapshot
                                .child(receiverUserID)
                                .child("request_type").getValue()
                                .toString();
```

A seconda del request_type presente in “ChatRequests” vengono mostrati diversi bottoni: se la richiesta è stata inviata, e l'interfaccia è quella del mittente apparirà il bottone “Cancel Chat Request”, altrimenti se l'interfaccia è quella del destinatario della chat request, appariranno i bottoni “Accept Chat Request” e “Cancel Chat Request”.

```
if(request_type.equals("sent")){
      Current_State = "request_sent";
      btnContatta.setText("Cancel Chat Request");
}
else if (request_type.equals("received")) {
      btnContatta.setText("Accept Chat Request");
      btnCancel.setVisibility(View.VISIBLE);
      btnCancel.setEnabled(true);
      btnCancel.setOnClickListener(new View.OnClickListener() {
      @Override
            public void onClick(View v) {
                  CancelChatRequest();
            }
     });
}
```

Se il contatto non è presente in “ChatRequests” ma in “Contacts”, vuol dire che la richiesta di contatto è già stata accettata ed è possibile messaggiare con l'utente destinatario della richiesta. Quindi il bottone che apparirà sarà quello di “Remove Contact”:

```
else {
      ContactsRef.child(senderUserID)
      .addListenerForSingleValueEvent(new ValueEventListener() {
            @Override
            public void onDataChange(@NonNull DataSnapshot
            dataSnapshot) {
                  if(dataSnapshot.hasChild(receiverUserID)) {
                        btnContatta.setText("Remove Contact");
                  }
            }
           @Override
           public void onCancelled(@NonNull DatabaseError
            databaseError) {
            }
```

```
                                    });
                        }
                    }
            @Override
            public void onCancelled(@NonNull DatabaseError
databaseError) {
            }
      });
```

Se viene premuto “Remove Contact” i dati relativi a quella coppia sender-receiver vengono cancellati:

```
private void RemoveSpecificContact() {
    ContactsRef.child(senderUserID).child(receiverUserID)
            .removeValue()
            .addOnCompleteListener(new OnCompleteListener<Void>() {
                @Override
                public void onComplete(@NonNull Task<Void> task) {
                    if(task.isSuccessful()){

                        ContactsRef.child(receiverUserID).
                        child(senderUserID)
                        .removeValue().addOnCompleteListener(new
                        OnCompleteListener<Void>() {
                         @Override
                         public void onComplete(@NonNull Task<Void>
                        task) {
                              if(task.isSuccessful()){
                                    btnContatta.setEnabled(true);
                                    btnContatta.setText("Send
                                    Message");
                                    btnCancel.setVisibility(View.
                                    INVISIBLE);
                                    btnCancel.setEnabled(false);
                              }
                         }
                         });
                    }
                }
            });
}
```

Mentre se la richiesta viene accettata, vengono eliminati i dati relativi alla coppia sender-receiver in “Chat Requests” e vengono creati per “Contacts”:

```
private void AcceptChatRequest() {
    ContactsRef.child(senderUserID).child(receiverUserID)
            .child("Contacts").setValue("Saved")
            .addOnCompleteListener(new OnCompleteListener<Void>() {
                @Override
                public void onComplete(@NonNull Task<Void> task) {
                    if(task.isSuccessful()){
```

```
            ContactsRef.child(receiverUserID).child(senderUserID)
                                .child("Contacts").setValue("Saved")
                                .addOnCompleteListener(new
OnCompleteListener<Void>() {
                                    @Override
                                    public void onComplete(@NonNull
Task<Void> task) {
                                        if(task.isSuccessful()){

ChatRequestRef.child(senderUserID)

.child(receiverUserID)
                                                        .removeValue()

.addOnCompleteListener(new OnCompleteListener<Void>() {
                                                            @Override
                                                            public void
onComplete(@NonNull Task<Void> task) {

if(task.isSuccessful()){

ChatRequestRef.child(receiverUserID)

.child(senderUserID)

.removeValue()

.addOnCompleteListener(new OnCompleteListener<Void>() {

@Override

public void onComplete(@NonNull Task<Void> task) {

btnContatta.setEnabled(true);

btnContatta.setText("Remove Contact");

btnCancel.setVisibility(View.INVISIBLE);

btnCancel.setEnabled(false);

}}); ...
```

Per poter iniziare a messaggiare con un contatto basta andare nella lista dei contatti, “Contacts” del Drawer e schiacciare sul contatto. Da qui si aprirà un'interfaccia come quella in Figura 3.34:

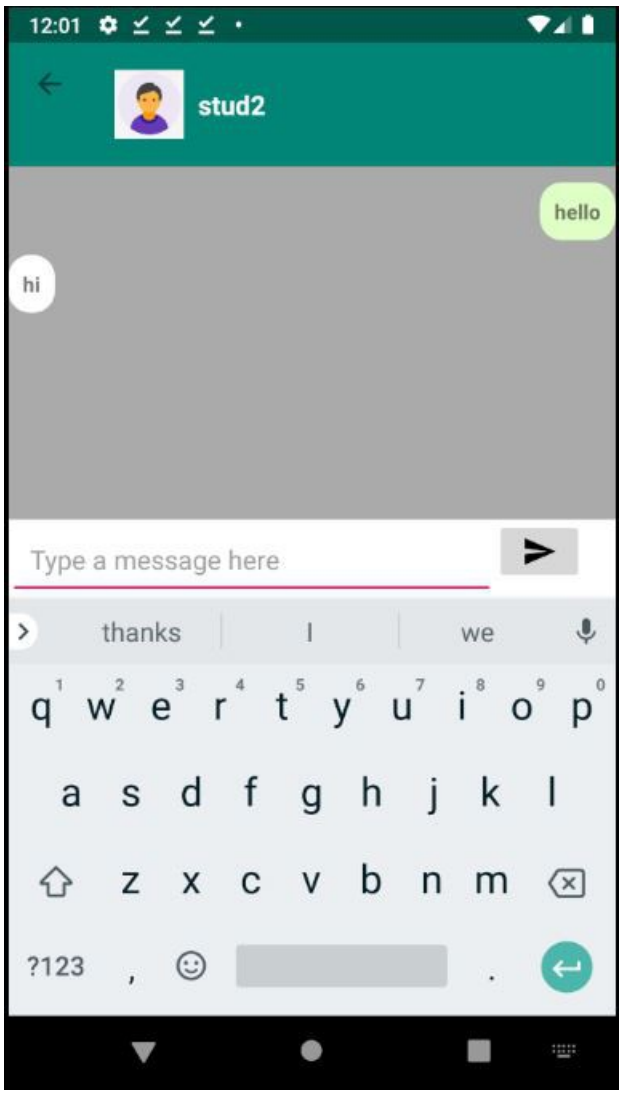

Figura 3.34: Chat

Ogni oggetto messaggio ha tre attributi:

```
public Messages(String from, String message, String type) {
    this.from = from;
    this.message = message;
    this.type = type;
}
```

Nel database infatti l'oggetto messaggio viene così memorizzato:

Figura 3.35: Come viene rappresentato il messaggio nel database

In Figura 3.35 ad esempio l'utente con ID *K9oZIxOqNJR27AJoFRa0fjlOh7q1* ha inviato un messaggio a *c6Mq3csXq1Mn4U9XxuFfgrbfxTu1*. L'ID del messaggio è invece *LiNnsFrunqt20ce66AQ*.

Per poter creare un'interfaccia come quella in Figura 3.34 si è giocato con i tre attributi, facendo apparire o scomparire, in MessageAdapter.java, le componenti grafiche create apposta per il messaggio del receiver e quello del sender:

```
@Override
public void onBindViewHolder(@NonNull final MessageViewHolder
messageViewHolder, int i) {
    String messageSenderID = mAuth.getCurrentUser().getUid();
    Messages messages = userMessagesList.get(i);
    String fromUserID = messages.getFrom();
    String fromMessageType = messages.getType();
    if(fromMessageType.equals("text")){

messageViewHolder.receiverMessageText.setVisibility(View.INVISIBLE);
 messageViewHolder.senderMessageText.setVisibility(View.INVISIBLE);
```

Se il messaggio viene inviato verrà utilizzato il sender_messages_layout.xml (senderMessageText), di colore verde, altrimenti verrà utilizzato il receiver_message_layout.xml:

```
if(fromUserID.equals(messageSenderID)){

messageViewHolder.senderMessageText.setVisibility(View.VISIBLE);

messageViewHolder.senderMessageText.setBackgroundResource(R.drawable.s
ender_messages_layout);
```

```
messageViewHolder.senderMessageText.setText(messages.getMessage());
        }else{

messageViewHolder.receiverMessageText.setVisibility(View.VISIBLE);

messageViewHolder.receiverMessageText.setBackgroundResource(R.drawable.
receiver_messages_layout);

messageViewHolder.receiverMessageText.setText(messages.getMessage());
        }
    }
}
```

sender_messages_layout.xml e receiver_messages_layout.xml si trovano nei drawables di “res”, perché sono le risorse che vengono utilizzate nel custom_messages_layout.xml.

sender_messages_layout.xml:

```
<?xml version="1.0" encoding="utf-8"?>

<shape xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:andrdoid="http://schemas.android.com/aapt">
    <solid android:color="#E0FFC7"></solid>
    <corners android:radius="15dp"></corners>
</shape>
```

receiver_messages_layout.xml:

```
<?xml version="1.0" encoding="utf-8"?>

<shape xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:andrdoid="http://schemas.android.com/aapt">
    <solid android:color="#FFFFFF"></solid>
    <corners android:radius="15dp"></corners>
</shape>
```

custom_messages_layout.xml:

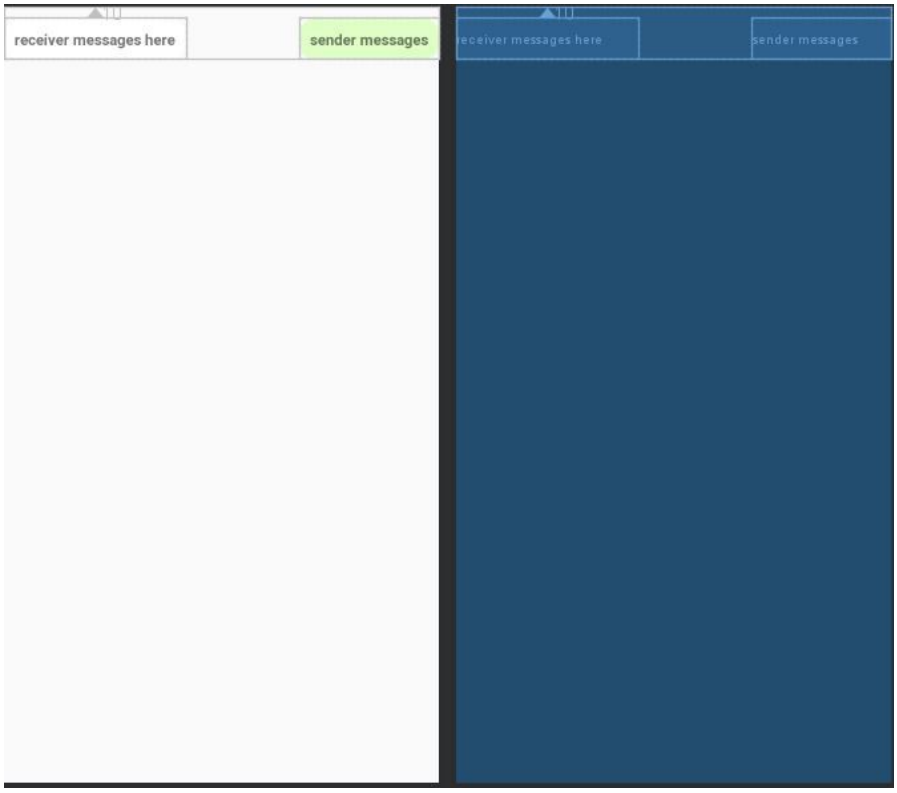

Figura 3.36: custom_messages_layout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <TextView
        android:id="@+id/receiver_message_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_marginTop="10dp"
        android:background="@drawable/receiver_messages_layout"
        android:text="receiver messages here"
        android:padding="10dp"
        android:textSize="14sp"
        android:textStyle="bold"/>

    <TextView
        android:id="@+id/sender_message_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentEnd="true"
        android:layout_marginTop="10dp"
        android:layout_alignParentRight="true"
        android:background="@drawable/sender_messages_layout"
```

```
        android:text="sender messages"
        android:textSize="14sp"
        android:textStyle="bold"
        android:padding="10dp"
        />
</RelativeLayout>
```

3.2.8 ProfileActivity

Infine l'ultima funzionalità realizzata che si vuole mostrare è quella del profilo utente. Infatti andando su “Profile” nel Drawer, è possibile accedere alle informazioni personali e modificarle. Per poter testare una tale funzionalità, si è deciso di limitarsi alla modifica del nome dell'utente, dell'email e dell'immagine, ma è possibile aggiungere anche altri attributi d'entità. Inoltre da qui è possibile, scegliendo l'opzione dal menu a tendina in alto a destra, effettuare il logout dall'applicazione.

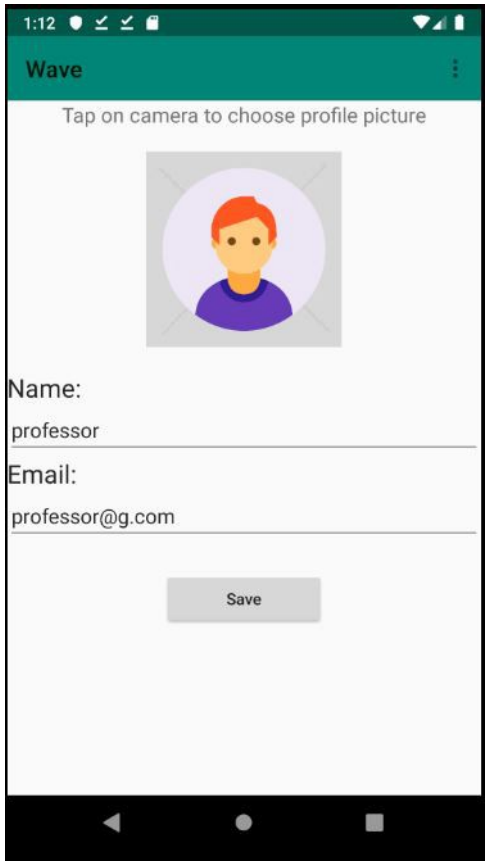

Figura 3.37: Profilo

La classe ProfileActivity.java è composta da due funzioni:

loadUserInformation():

```
private void loadUserInformation() {
    if (user != null) {
        String displayName = user.getDisplayName();
        Uri profileUri = user.getPhotoUrl();
        if(profileUri != null){
            final StorageReference ref =
            FirebaseStorage.getInstance().getReference().
            child("ProfilePics").child(mAuth.getUid());
            ref.getDownloadUrl().addOnSuccessListener(new
            OnSuccessListener<Uri>() {
                @Override
                public void onSuccess(Uri uri) {
                    Picasso.get()
                            .load(uri)
                            .placeholder(R.drawable.placeholder)
                            .fit()
                            .centerCrop()
                            .into(imageView);
                }
            });
        }
        for (UserInfo userInfo : user.getProviderData()) {
            if (displayName == null && userInfo.getDisplayName() !=
            null) {
                displayName = userInfo.getDisplayName();
            }
            if (profileUri == null && userInfo.getPhotoUrl() != null) {
                profileUri = userInfo.getPhotoUrl();
            }
        }
        textViewName.setText(displayName);
        textView.setText(user.getEmail());
    }
}
```

In questa funzione, volendo visualizzare solo i tre attributi base del currentUser, le informazioni sono stata caricate da Firebase Authentication, invece che dal database, come anche nella seconda funzione di seguito.

saveUserInformation:

```
private void saveUserInformation() {
    String displayName = textViewName.getText().toString();
    if (displayName.isEmpty()) {
        textViewName.setError("Name is required");
        textViewName.requestFocus();
        return;
```

```
    }
    if (user != null && profileImageUrl != null) {
        UserProfileChangeRequest profile = new
            UserProfileChangeRequest.Builder().
            setDisplayName(displayName).build();
        user.updateProfile(profile).addOnCompleteListener(new
            OnCompleteListener<Void>() {
            @Override
            public void onComplete(@NonNull Task<Void> task) {
                if (task.isSuccessful()) {
                    Toast.makeText(ProfileActivity.this, "Profile name
                    Updated", Toast.LENGTH_SHORT).show();
                }
            }
        });
        UserProfileChangeRequest profileUpdates1 = new
            UserProfileChangeRequest.Builder().
            setPhotoUri(Uri.parse(profileImageUrl)).build();
        user.updateProfile(profileUpdates1).addOnCompleteListener(new
            OnCompleteListener<Void>() {
            @Override
            public void onComplete(@NonNull Task<Void> task) {
                if (task.isSuccessful()) {
                    Toast.makeText(ProfileActivity.this, "Profile
                        image Updated", Toast.LENGTH_SHORT).show();
                }
            }
        });
        user.reload();
    }
}
```

4 Conclusioni e sviluppi futuri

I motivi che hanno portato ad analizzare questo caso di studio sono l'assenza di uno strumento simile, perlomeno in territorio italiano, che possa facilitare studenti e professori, e le esperienze personali e di altre persone che hanno portato alla luce un problema che quest'applicazione ha lo scopo di risolvere.
Nello specifico ci si è focalizzati soprattutto sulla costruzione di un'interfaccia semplice, che potesse essere chiara anche ai meno esperti e in futuro si potrebbero apportare molte migliorie, specie all'interfaccia dell'utente registrato come professore, che è stata volutamente trascurata per concentrarsi su altri aspetti. Per quanto riguarda questa parte infatti si è pensato alla possibilità di personalizzare il profilo di ogni utente professore con un calendario in cui sia possibile visualizzare date e orari. Grazie ad un calendario infatti gli studenti potrebbero verificare la disponibilità del professore ed eventualmente si potrebbe dar modo ad essi di prenotarsi. La prenotazione di una lezione renderebbe quest'ultima non accessibile agli altri studenti, mentre il professore avrebbe un prospetto della settimana dei propri impegni. In questo modo verrebbero semplificate alcune dinamiche, poiché adesso l'unico modo per uno studente di prenotarsi per una lezione è contattando il professore per messaggio, attraverso la chat sviluppata per l'applicazione.
Mentre per quanto riguarda la risoluzione del caso di studio, si è riusciti a soddisfare la necessità di mettere in contatto studenti e professori attraverso l'implementazione della parte di messaggistica.
Si è implementato un filtro dei dati della lista di professori presenti e dei contatti del proprio profilo, in modo da ottimizzare la ricerca della persona nel caso si voglia

comunicare con essa. Inoltre lo studente può effettuare la ricerca attraverso i filtri sul luogo (o un luogo della lista o la propria posizione) e la materia.
Si è scelto di gestire la memorizzazione dei luoghi inserendoli nel database man mano che un utente professore si registrava, in questo modo si è evitato di inserirli manualmente, facendo perdere tempo all'amministratore. Ovviamente questa parte potrebbe essere sviluppata diversamente, perché al momento questa logica non è totalmente efficiente. Inoltre sempre per quanto riguarda l'interfaccia del professore si è scelto di far postare gli annunci all'utente registrato come professore, semplicemente scegliendo tra un elenco di materie precedentemente inserite a mano nel database, quindi anche questa parte potrebbe essere migliorata, automatizzando l'inserimento delle materie tra cui scegliere nel database.
Come già detto, si è scelto di focalizzarsi su alcuni aspetti, per la maggior parte relativi all'interfaccia dello studio e all'aspetto social dell'applicazione, mentre si è volutamente trascurata l'interfaccia professore di cui è stata espressa la volontà di sviluppare in futuro altre funzionalità.

5 Bibliografia

1: Android Studio Documentation, https://developer.android.com/studio
2: Firebase Documentation, https://firebase.google.com/docs/
3: Articolo Sql e NoSql a confronto, https://www.html.it/articoli/sql-e-nosql-a-documenti-il-confronto/
4: Articolo su Geofire, https://firebase.googleblog.com/2013/09/geofire-location-queries-for-fun-and.html
5: Node.js Documentation, https://nodejs.org/en/
6: Picasso Documentation, https://github.com/square/picasso
7: Come ottenere il join in Firebase, https://www.youtube.com/watch?v=HjlQH3RsGcU&feature=youtu.be&t=164s
8: HTML.it: RecyclerView vs ListView, https://www.html.it/articoli/recyclerview-dietro-le-quinte/