UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Dipartimento di Scienze Matematiche, Fisiche e Informatiche
Corso di Laurea in Informatica

Progettazione e Sviluppo di un'applicazione Android per l'unione tra domanda e offerta di attività lavorative svolte in casa

Candidato:
Giovanni Gagliano

Relatore:
Prof. Riccardo Martoglia

Anno Accademico 2018/2019

PAROLE CHIAVE

Applicazione mobile
Android Studio
Kotlin
Firebase
Geofire

Sommario

INTRODUZIONE **1**

PARTE I
Il caso di studio

CAPITOLO 1: IL CASO DI STUDIO **3**
CAPITOLO 2: TECNOLOGIE UTILIZZATE **5**
2.1 ANDROID STUDIO: Introduzione 5
2.2 ANDROID SDK 6
2.3 CREAZIONE PROGETTO 6
2.4 STRUTTURA PROGETTO 7
2.4.1 Manifest File 8
2.4.2 Cartella res 8
2.4.3 Gradle 9
2.5 FONDAMENTA ANDROID 10
2.5.1 Activity 11
2.5.2 Service 12
2.5.3 Intent 12
2.6 FIREBASE 14
2.6.1 Introduzione 14
2.6.2 Perché firebase? 15
2.6.3 Autenticazione 16
2.6.4 Realtime Database 17
2.6.5 Firebase Cloud Storage 19
2.7 DATABASE NoSQL / RELAZIONALE 20
2.8 GEOFIRE 22
2.8.1 Geofire e Firebase 22
2.9 PICASSO 23
2.10 KOTLIN 24

2.10.1 Introduzione a Kotlin 24
2.10.2 Principali caratteristiche 24

PARTE II
Progettazione e implementazione

CAPITOLO 3: PROGETTAZIONE 29
3.1 ANALISI DEI REQUISITI 29
3.1.1 Funzionalità di base 30
3.1.2 Caso d'uso 30
3.1.3 Diagramma delle attività 33
3.2 PROGETTAZIONE DATABASE 35
3.2.1 Costruzione delle entità 37
CAPITOLO 4: IMPLEMENTAZIONE 40
4.1 INTRODUZIONE ALL'IMPLEMENTAZIONE 40
4.2 STRUTTURA PROGETTO 41
4.3 IMPLEMENTAZIONE DELL'APPLICAZIONE 45
4.3.1 Manifest File 45
4.3.2 Register Activity 47
4.3.3 RegisterAsExpert Activity 55
4.3.4 Login Activity 57
4.3.5 UserHome Activity 60
4.3.6 ProfileView Activity 68
4.3.7 ExpertProfile Activity 70
4.3.8 Edit Activity 71
4.3.9 Chatlog Activity 74
4.3.10 LatestMessages Activity 78

CONCLUSIONI E SVILUPPI FUTURI .....

Indice Figure

Figura 2.1: Struttura progetto su Android Studio 7
Figura 2.2: Gradle file di un'applicazione 10
Figura 2.3: Ciclo di vita di un'Activity in Android 11
Figura 2.4: Intent per l'avvio di un'activity 13
Figura 2.5: Strumenti offerti gratuitamente da Firebase 15
Figura 2.6: Elenco di tutti i metodi di accesso supportati da Firebase 17
Figura 2.7: Struttura del Realtime Database relativo alla mia applicazione, popolato con dati d'esempio. 18
Figura 2.8: Elenco di immagini relative alla mia applicazione memorizzate nello storage di Firebase 19
Figura 2.9: Schema che mostra le principali differenze tra un database relazionale ed un database NoSQL 21
Figura 2.10: Come Geofire memorizza i dati all'interno del Realtime Database 23
Figura 2.11: Caratteristiche fondamentali di Kotlin 28
Figura 3.1: Diagramma di caso d'uso dell'applicazione 32
Figura 3.2: Legenda del diagramma attività 33
Figura 3.3: Diagramma delle attività completo 34
Figura 3.4: Schema entità database 35
Figura 3.5: Entità Utenti 37
Figura 3.6: Entità Locations 37
Figura 3.7: Entità User Messages 38
Figura 3.8: Entità Latest Messages 38
Figura 4.1: Esempio struttura di un qualsiasi progetto Android in Android Studio 41
Figura 4.2: cartella classes 42
Figura 4.3: cartella core 42
Figura 4.4: cartella messages 42
Figura 4.5: cartella registerlogin 43
Figura 4.6: Cartella res con relative sottocartelle (drawable, layout e values) 44
Figura 4.8: Register Activity con campi completati con dati d'esempio 47
Figura 4.7: Schermata iniziale dell'applicazione che mostra il format di registrazione 47
Figura 4.10: Activity RegisterAsExpert con campi compilati 55
Figura 4.9: Spinner per la selezione della professione 55
Figura 4.11: Schermata di login 57
Figura 4.12: Activity UserHome 60
Figura 4.13: spinner dell'activity UserHome per la ricerca in base alla professione 65
Figura 4.16: L'utente sposta la barra che indica il raggio in km entro il quale si vogliono cercare gli utenti 67

Figura 4.15: Vengono visualizzati solo gli utenti che hanno come professione quella selezionata nello spinner..... 67
Figura 4.14: Nella barra di ricerca compare la stringa “Gab”. L’applicazione filtra gli utenti per Far visualizzare solo coloro che nel proprio username contengono quella stringa. 67
Figura 4.17: Visualizzazione dell’utente Gabriele con le relative informazioni. 68
Figura 4.18: ExpertProfile Activity di un utente d’esemipo 70
Figura 4.19: Edit Activity appena viene avviata 71
Figura 4.20: Chatlog Activity in cui l’utente sta scrivendo un messaggio ad un altro utente di nome Gabriele. 74
Figura 4.21: Chatlog Activity dopo che è stato inviato il primo messaggio 74
Figura 4.22: Chatlog dal lato dell’esperto 77
Figura 4.23: LatestMessages activity 78

INTRODUZIONE

Durante il mio corso di studi in informatica è maturato in me il desiderio di dare vita a qualcosa di originale che le persone possano utilizzare quotidianamente. Ho pertanto sfruttato l'opportunità del tirocinio per sviluppare un'applicazione mobile che potesse soddisfare questo desiderio. Proprio quando ero alla ricerca dell'oggetto della mia applicazione mi è stato chiesto dalla mia parrucchiera se esistesse un'applicazione che le potesse far raggiungere clienti vicini a lei. È nata così l'idea di creare un mezzo unico di ricerca e comunicazione per quei professionisti, di vari settori, che vogliono acquisire visibilità nella zona in cui lavorano e per quelle persone che vogliono guardarsi intorno e cercare chi offre il servizio di cui hanno bisogno. L'applicazione presenta diverse funzionalità, specialmente di ricerca, che possano raggiungere lo scopo prefissato ed è pensata come un'applicazione intuitiva e veloce, da utilizzare "al volo" per esigenze momentanee.

Per quanto riguarda l'implementazione è stato utilizzato l'IDE Android Studio, usando il nuovo linguaggio di programmazione Kotlin. Ho così approfittato della situazione per apprendere ed esplorare un linguaggio a me ancora sconosciuto: le mie prime ore di tirocinio sono state occupate appunto dallo studio di Kotlin e di Firebase, il quale mi ha permesso di gestire i dati utilizzati attraverso un database real-time.

La struttura della tesi si divide in due parti principali, il caso di studio e l'applicazione, ognuna contenente due capitoli.

- Il caso di studio:
    1. È stato parzialmente espresso sopra ma verrà approfondito nel primo capitolo includendo l'analisi di obiettivi e requisiti
    2. Il secondo capitolo comprende una descrizione dettagliata delle tecnologie utilizzate
- L'applicazione: progettazione e implementazione dell'app con relativo codice e immagini.

    3. Nel terzo capitolo si vedranno i dettagli della progettazione dove verranno descritti casi d'uso e funzionalità

    4. Si mostreranno in maniera approfondita, nel quarto capitolo, i procedimenti della fase implementativa

Vi sarà inoltre un ultimo capitolo di conclusione dove espongo le mie considerazioni finali al termine di quest'esperienza e in cui rifletto su eventuali sviluppi futuri relativi all'applicazione.

CAPITOLO 1

IL CASO DI STUDIO

Tra le caratteristiche che un'applicazione mobile può avere, l'intuitività è stata sin dall'inizio quella che più mi ha ispirato e portato ad effettuare determinate scelte nella progettazione. Creare quindi un'interfaccia user-friendly era per me uno dei principali obiettivi da raggiungere, dato che l'applicazione non è destinata ad una specifica categoria di persone, ma bensì utilizzabile da chiunque.

Inizialmente l'idea di base era quella di realizzare un'applicazione mirata alla ricerca di quei parrucchieri che effettuano la propria attività in casa, ma la struttura dell'app si adattava bene a tanti altri mestieri, dando così la possibilità di inserire diverse attività che possono essere svolte all'interno di un'abitazione.

In linea generale quindi, per l'utente l'applicazione deve essere in grado di individuare le varie offerte di esperti nelle vicinanze e di poter contattare gli stessi attraverso una chat semplice e intuitiva. Da parte di chi offre un'attività invece, l'applicazione si rivela utile per allargare il proprio bacino d'utenza e raggiungere nuovi clienti. Ovviamente ciò è possibile solamente attraverso i moderni smartphone dotati di geolocalizzazione.

Il dubbio che ci fosse già qualcosa di simile come applicazione non si è mai presentato dal momento che è stata proprio la mia parrucchiera a espormi il fatto che non esistesse nulla del genere dopo aver controllato sui vari store.

Il metodo probabilmente più utilizzato attualmente per l'incontro tra professionisti e clienti è quello di utilizzare siti di annunci o gruppi e pagine Facebook, il che può essere tedioso e meno chiaro rispetto ad un'applicazione mirata al problema. La chiarezza non è sicuramente un elemento che contraddistingue le interfacce di questi siti su smartphone, caso in cui un'applicazione risulta molto più adatta, trasparente e veloce.

La velocità è un altro elemento su cui si basa l'app. Essa si presta infatti molto bene in situazioni in cui in brevi tempi e in un'area vicina all'utilizzatore si vuole trovare qualcuno che soddisfi un'esigenza, permettendo agli utenti di visualizzare le opzioni e contattare chi meglio risponde ai bisogni di quel momento.

Il sistema di messaggistica dell'app inoltre è una funzionalità non presente nei siti di annunci e che facilita la comunicazione tra clienti e professionisti, i quali possono avere quindi un dialogo per organizzare un incontro. Il lato social dell'applicazione viene accentuato anche dalla possibilità di caricare immagini da parte di chi offre un'attività così da poter attirare l'attenzione di eventuali clienti.

CAPITOLO 2

TECNOLOGIE UTILIZZATE

In questo capitolo verranno mostrate le tecnologie che mi hanno permesso di realizzare l'applicazione e tutte le funzionalità di cui essa dispone.

2.1 ANDROID STUDIO: Introduzione

Ho effettuato varie ricerche per decidere e valutare quale ambiente di sviluppo utilizzare. Queste mi hanno portato a scegliere Android Studio, attualmente l'IDE primario di Google per lo sviluppo nativo di app Android [1], per ragioni quali la praticità e la flessibilità. Inoltre, in rete è molto vasta la quantità di informazioni e documentazione a riguardo.
Fattori che hanno ulteriormente confermato questa decisione sono:

- Ambiente di lavoro efficiente ma soprattutto intuitivo
- Presenza di un layout editor semplice per lo sviluppo dell'interfaccia grafica
- Programmazione agevole grazie a strumenti del tutto gratuiti
- Monitoraggio delle risorse di memoria e della CPU utilizzate dall'app

Un'ulteriore ed importante motivazione che mi ha spinto alla scelta di Android Studio è stata la mia personale volontà di apprendere un nuovo linguaggio di programmazione. Informandomi infatti sul web riguardo a questo ambiente di sviluppo, ho notato che supporta

Kotlin, un linguaggio nato da poco che ha ottenuto il favore di molti programmatori. Come già accennato precedentemente ne ho quindi approfittato per programmare in un linguaggio moderno e a me nuovo, di cui parlerò a breve.

2.2 ANDROID SDK

L'SDK (Software Development Kit) è un kit di sviluppo software, in questo caso Android. Si tratta quindi di un insieme completo di strumenti di sviluppo, come per esempio emulatori, debugger ecc.

Android Studio dà quindi la possibilità di interfacciarsi ad ognuna di queste funzionalità, permettendo inoltre, attraverso l'sdk manager, di gestirne la composizione.

Il kit specifico per Android è sin dall'inizio incluso nell'installazione di Android Studio e tra i tools che contiene, uno dei più importanti è sicuramente l'emulatore, che io stesso ho utilizzato abbondantemente nello sviluppo dell'applicazione. Grazie a esso infatti non si è resa necessaria la presenza di un vero e proprio dispositivo mobile ma si è potuta testare l'app direttamente da Android Studio, dopo aver opportunatamente creato un dispositivo virtuale tramite l'AVD (Android Virtual Device) Manager.

2.3 CREAZIONE PROGETTO

Avendo tutti gli strumenti a disposizione, si è pronti per creare un nuovo progetto. Android Studio fa un ottimo lavoro nel guidare l'utente nelle varie fasi necessarie per la creazione del primo progetto.

Tra le varie schermate che ci vengono proposte vi sono due scelte fondamentali per la progettazione dell'applicazione. La prima riguarda la tipologia di applicazione e il contesto applicativo: bisogna quindi effettuare una scelta tra phone, tablet, TV, Wear OS ed altri ancora. Successivamente viene proposta una serie di modelli di base, ovvero i vari template di activity disponibili per lo sviluppo del progetto.

La seconda scelta fondamentale è il nome del progetto. Da questo dipende il nome completo del package e la collocazione del progetto nel file system della macchina di sviluppo. Altra

importante scelta riguarda la selezione del linguaggio di programmazione che si vuole utilizzare: Java o Kotlin.

Dopo questa prima fase guidata ci si trova quindi con un progetto funzionante ma vuoto, di cui Android Studio ne palesa la struttura, spiegata nel dettaglio nel prossimo paragrafo e la cui comprensione è di grande aiuto per lo sviluppatore.

2.4 STRUTTURA PROGETTO

Il progetto è composto da tre parti principali:

- Il manifest file (AndroidManifest.xml)
- La cartella contenente il codice Kotlin
- La cartella res

Infine, è sempre presente una sezione "Gradle Scripts" contenente i file Gradle successivamente esplicati.

La figura 2.1 rappresenta la struttura generale di un qualsiasi progetto in Android Studio.

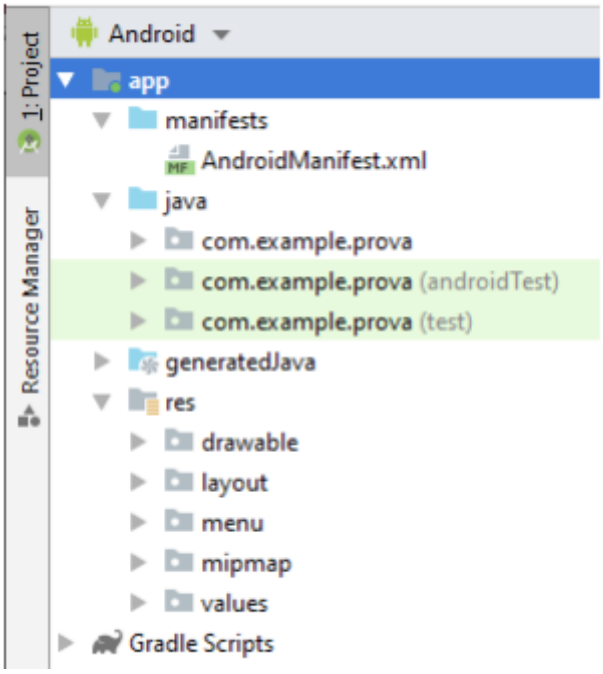

Figura 2.1: Struttura progetto su Android Studio

2.4.1 Manifest File

Si tratta di un file creato in maniera automatica da Android Studio durante l'inizializzazione del progetto.
Ogni progetto deve infatti avere nella cartella origine del progetto un AndroidManifest.xml file con esattamente questo nome e saperlo configurare è fondamentale per dare vita ad un'app. Android. Descrive informazioni essenziali riguardo l'applicazione stessa ed è richiesto per dichiarare il nome del package, i componenti (activities, services ecc.), i permessi e i requisiti hardware e software dell'applicazione. Il manifest file è anche il luogo dove vengono nominate le activity, con i loro dettagli implementativi, e dove viene definita l'icona che rappresenterà l'applicazione sui nostri dispositivi.

2.4.2 Cartella res

Questa cartella ha lo scopo di immagazzinare tutti quei "valori" che vengono messi a disposizione dell'applicazione, dove per "valori" si intendono elementi interni al progetto, come per esempio file XML, immagini ed altro ancora.
In res troviamo diverse sottocartelle, tra le più importanti:

- **layout**: contiene tutti quei file in cui vengono definiti i componenti per costruire un'interfaccia utente. In generale quindi è un insieme di file XML che costituiscono l'architettura e il design dell'applicazione.

- **values**: cartella che contiene stringhe, colori e altri tipi di valori che il programmatore definisce con l'uso di tag XML appropriati. Questi valori vengono direttamente utilizzati nel codice dell'applicazione o all'interno di altre risorse

- **drawable**: contiene le immagini, nei formati più comuni, utilizzate nell'applicazione. Android sceglie le immagini dalle sottocartelle in base al tipo di dispositivo in utilizzo.

2.4.3 Gradle

Gradle è uno strumento per automatizzare le build ed è stato sviluppato in maniera indipendente da Android Studio. Ha quindi lo scopo di scaricare le dipendenze necessarie (come librerie fornite da altri sviluppatori), compilare il codice sorgente, effettuare il packaging degli eseguibili in un file .apk. Si può definire come un tool per l'automazione della build di un'applicazione.

Tutto ciò, in Android Studio, lo si può trovare nella sezione Gradle Scripts dove si trovano due file build.gradle. Il primo dei due si riferisce all'intero progetto mentre il secondo è specifico del modulo di default "app", che è solitamente il modulo in cui il programmatore effettua delle modifiche.

La figura 2.2 mostra il file build.gradle(app) dove tra le sezioni più importanti vi sono la definizione della versione SDK minima dell'applicazione e le dependencies che costituiscono le librerie per espandere le funzionalità del progetto. Ogni qualvolta questi file vengono modificati è necessario andare ad effettuare una sincronizzazione del progetto con le nuove impostazioni.

```
1 apply plugin: 'com.android.application'
2
3 apply plugin: 'kotlin-android'
4
5 apply plugin: 'kotlin-android-extensions'
6
7 android {
8     compileSdkVersion 29
9     buildToolsVersion "29.0.2"
10    defaultConfig {
11        applicationId "com.example.helloworld"
12        minSdkVersion 15
13        targetSdkVersion 29
14        versionCode 1
15        versionName "1.0"
16        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
17    }
18    buildTypes {
19        release {
20            minifyEnabled false
21            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
22            'proguard-rules.pro'
23        }
24    }
25 }
26
27 dependencies {
28    implementation fileTree(dir: 'libs', include: ['*.jar'])
29    implementation"org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
30    implementation 'androidx.appcompat:appcompat:1.0.2'
31    implementation 'androidx.core:core-ktx:1.0.2'
32    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
33    testImplementation 'junit:junit:4.12'
34    androidTestImplementation 'androidx.test:runner:1.2.0'
35    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
36 }
```

Figura 2.2: Gradle file di un'applicazione

2.5 FONDAMENTA ANDROID

In qualsiasi applicazione Android sono presenti dei componenti che ne costituiscono le fondamenta e che svolgono le attività principali dell'applicazione stessa, facendo in modo che essa possa interfacciarsi e integrarsi alla perfezione con l'ecosistema Android.

Si tratta dei concetti di Activity, Services e Intents.

2.5.1 Activity

Non è altro che un'interfaccia utente. Possiamo considerare un'activity come la "pagina" che l'utente sta visualizzando in un determinato momento, di conseguenza è ciò con cui l'utilizzatore ha contatto diretto.

Essendo ogni applicazione costituita solitamente da più activity, la loro interezza costituisce il flusso in cui l'utente si inoltra per usare tutte le funzionalità dell'applicazione [2].

Si rende quindi necessaria, all'interno di ogni activity, una corretta progettazione della navigazione tra di esse per fare in modo che i contenuti possano essere "sfogliati" in maniera coerente.

Ogni activity ha poi un proprio ciclo di vita, la figura 2.3 ne mostra i vari stati.

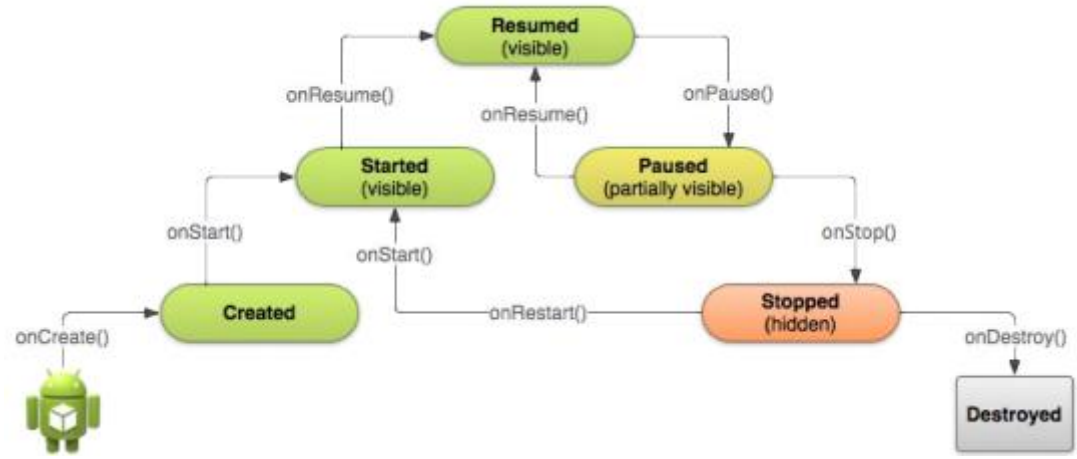

Figura 2.3: Ciclo di vita di un'Activity in Android

Possiamo vedere la presenza di 6 stati, i quali vengono attivati successivamente all'invocazione di metodi di callback da parte del sistema (i nomi sulle frecce che collegano gli stati).

Quando un'activity viene avviata vengono invocati i metodi onCreate(), onStart() e onResume() che rispettivamente servono a configurare l'interfaccia dopo il primo avvio di un'activity, attivare le funzionalità per far visualizzare correttamente le informazioni all'utente ed interagire infine con esso ricevendone gli input.

I restanti metodi servono nel caso l'activity venga messa in pausa o non sia più visibile (metodi onPause() e onStop()) e nel caso in cui l'activity venga definitivamente chiusa con onDestroy().

2.5.2 Service

Un service svolge invece un ruolo quasi opposto a quello di un'activity.
Rappresenta infatti un insieme di operazioni che vengono effettuate in background, generalmente durante un tempo prolungato, senza nessuna interazione con l'utente.
Un esempio che molti di noi usano è quello della riproduzione di musica da parte di un'app anche dopo che la sua interfaccia è stata chiusa. Sono i services ad effettuare il lavoro in questo caso.

2.5.3 Intent

Come è stato spiegato prima, è molto importante la navigazione tra le varie activity che compongono un'applicazione. Per far sì che questo avvenga in maniera corretta e coerente si rende necessario un meccanismo che permetta il passaggio tra un'activity ed un'altra. Questo meccanismo in Android prende il nome di *intent*.
Il sistema operativo utilizza un intent per ricevere una richiesta da parte di una componente di esecuzione di un'azione da parte di un'altra componente. Tutto ciò è possibile grazie ad una caratteristica estremamente importante degli intent, ovvero quello di trasportare dati, i quali vengono chiamati *extras*.
Ovviamente questi dati possono essere letti dal destinatario e la loro gestione negli intent ha il seguente funzionamento: attraverso un metodo *put* si inserisce il valore che si vuole trasportare, etichettato con una chiave, successivamente con i metodi *get* viene poi prelevato

questo valore richiedendolo attraverso la chiave.

La figura 2.4 mostra l'uso più frequente di un Intent, ovvero l'avvio di un' Activity.

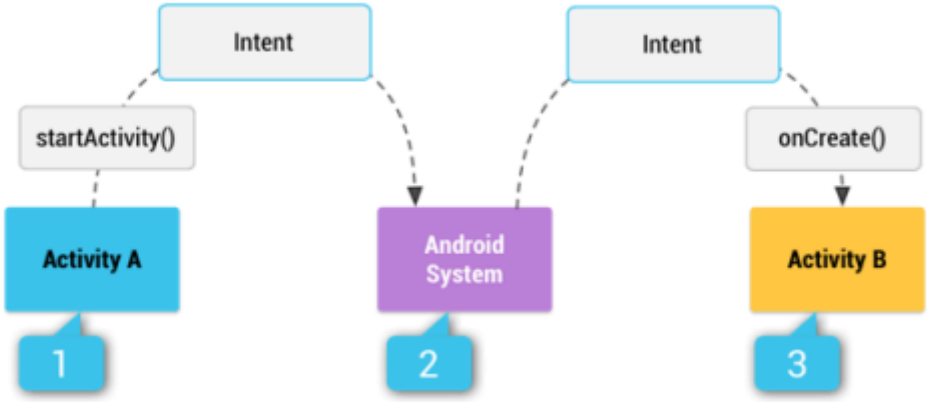

Figura 2.4: Intent per l'avvio di un'activity

2.6 FIREBASE

2.6.1 Introduzione

Di proprietà di Google, Firebase è un pacchetto completo di prodotti che permettono lo sviluppo di app mobile e web, offrendo numerosi servizi con la semplice implementazione di poche righe di codice all'interno del progetto [3]. Al giorno d'oggi immagazzinare data in cloud è estremamente importante e Firebase, essendo un BaaS (BackEnd as a Service) si rivela davvero utile nel costruire funzionalità utilizzando il cloud nell'app mobile.
Un BaaS è un mezzo che ha lo scopo di collegare il web e le applicazioni mobile al backend cloud storage e alle backend API.
Le caratteristiche peculiari di Firebase, che lo hanno reso popolare, sono:

- Sincronizzazione dei dati, oltre che dello storage, in maniera istantanea.
- Presenza di varie librerie client che permettono di integrare Firebase in qualsiasi app Android
- Sicurezza dei dati memorizzati, in quanto essi vengono replicati e continuamente sottoposti a backup

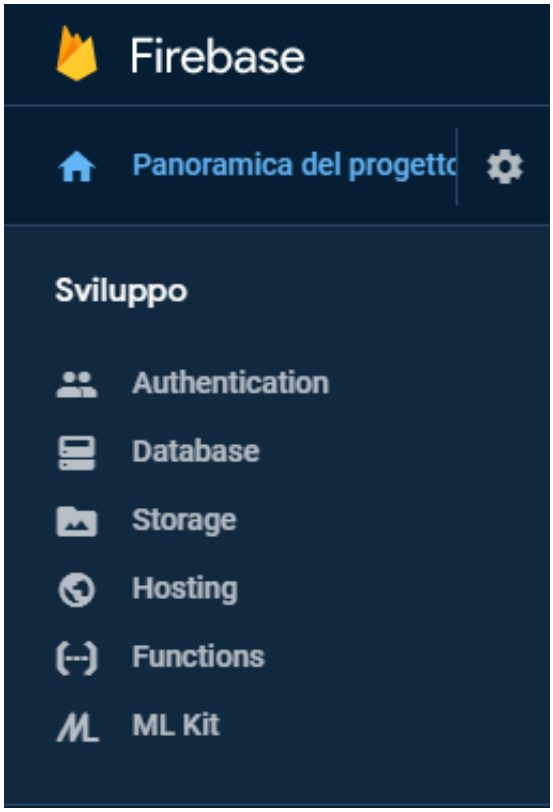

Figura 2.5: Strumenti offerti gratuitamente da Firebase

2.6.2 Perché firebase?

Per lo sviluppo dell'applicazione si è reso necessario l'utilizzo di un database per gestire le varie entità. Non bisogna effettuare molte ricerche per imbattersi in Firebase, la tecnologia da me utilizzata. Si tratta di uno degli strumenti maggiormente diffusi nello sviluppo di applicazioni mobile Android.

Una delle ragioni è dovuta al fatto che Android Studio integra Firebase, semplificando notevolmente i passaggi di inizializzazione del database e di sincronizzazione tra app e database. Firebase risponde a keyword quali innovazione, semplificazione e velocità, altre caratteristiche che hanno definitivamente influenzato la scelta.

2.6.3 Autenticazione

L'autenticazione è sempre un passaggio cruciale e Firebase ne semplifica l'implementazione rendendola molto veloce. Esso propone molteplici metodi per la fase di sign-in o log-in dell'utente, dal classico e-mail e password all'utilizzo di Google, Facebook ed altri popolari siti per effettuare l'accesso.

L'autenticazione con Firebase si rivela estremamente sicura, essendo sviluppata da sviluppatori Google.

Nel caso specifico della mia applicazione, l'unico metodo di autenticazione possibile è quello di e-mail e password. Come primo passaggio quindi si inseriscono le credenziali che verranno passate alla SDK di Firebase Authentication. Una volta verificata la correttezza delle credenziali, è possibile accedere alle informazioni di base dell'utente ed ai suoi dati relativi contenuti magari in altri servizi Firebase, come Firebase Realtime Database o Firebase Cloud Storage.

Inoltre sulla console Firebase si ha la possibilità di modificare il file contenente le regole di accesso ai dati contenuti nel database per ogni utente, e quindi limitarne le autorizzazioni se si vuole. Di default ogni utente ha i permessi di lettura e scrittura.

È importante capire il funzionamento dell'istanza Auth di Firebase: ogni qualvolta un utente si registra o effettua il login, entrando quindi nell'applicazione, questo diventa il "current user" di Auth. Finché l'utente non effettua il log-out, l'istanza mantiene lo stato dell'utente corrente così da tenere memorizzate le sue informazioni ogni qualvolta l'applicazione venga riavviata o si effettui un refresh di un'activity.

La figura 2.6 illustra tutti i possibili metodi di accesso che Firebase offre.

Provider di accesso

Provider	Stato
Email/password	Abilitato
Telefono	Disabilitato
Google	Disabilitato
Play Giochi	Disabilitato
Game Center Beta	Disabilitato
Facebook	Disabilitato
Twitter	Disabilitato
GitHub	Disabilitato
Yahoo	Disabilitato
Microsoft	Disabilitato
Apple	Disabilitato
Anonimo	Disabilitato

Figura 2.6: Elenco di tutti i metodi di accesso supportati da Firebase

2.6.4 Realtime Database

Firebase Realtime Database è un database NoSQL cloud-hosted, ovvero si trova in cloud e l'accesso all'user è lavoro di un service. I dati vengono memorizzati sottoforma di un unico albero JSON (Javascript Object Notation), un formato comodo per memorizzare e trasportare informazioni. Secondo molti sviluppatori è un alleato formidabile nella creazione di applicazioni che includono elementi di messaggistica, live streaming e GPS. Il fatto che sia cloud-hosted non ne limita il funzionamento quando l'utente è offline ma memorizza i cambiamenti e li sincronizza non appena l'utente va online. Dal momento che la sincronizzazione dei dati è effettuata in real-time, ogni update viene visto subito dai dispositivi che usano l'applicazione.

Vediamo nella figura 2.7 la struttura del database della mia applicazione per capirne subito la differenza con un database relazionale.

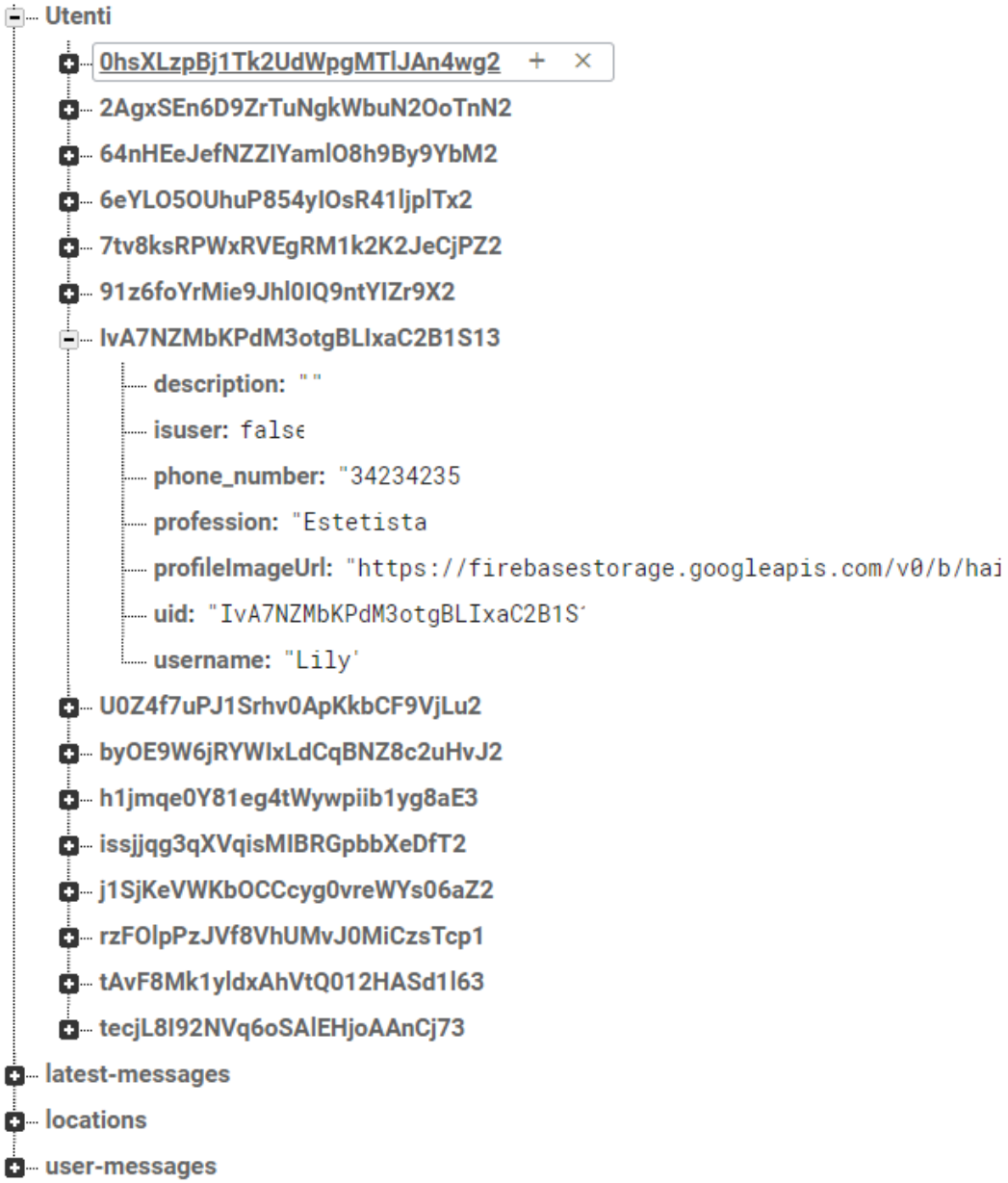

Figura 2.7: Struttura del Realtime Database relativo alla mia applicazione, popolato con dati d'esempio.

2.6.5 Firebase Cloud Storage

Si tratta di un servizio che permette alle applicazioni mobile di immagazzinare file multimediali come foto e video sul cloud. Firebase Cloud Storage è integrato al Firebase SDK, che permette agli utenti di memorizzare file sul cloud nel giro di pochi secondi. Si integra inoltre con Firebase Authentication per garantire all'utente finale un ambiente sicuro. Uso come esempio i dati della mia applicazione per illustrare nella figura 2.8 lo storage delle immagini che essa memorizza.

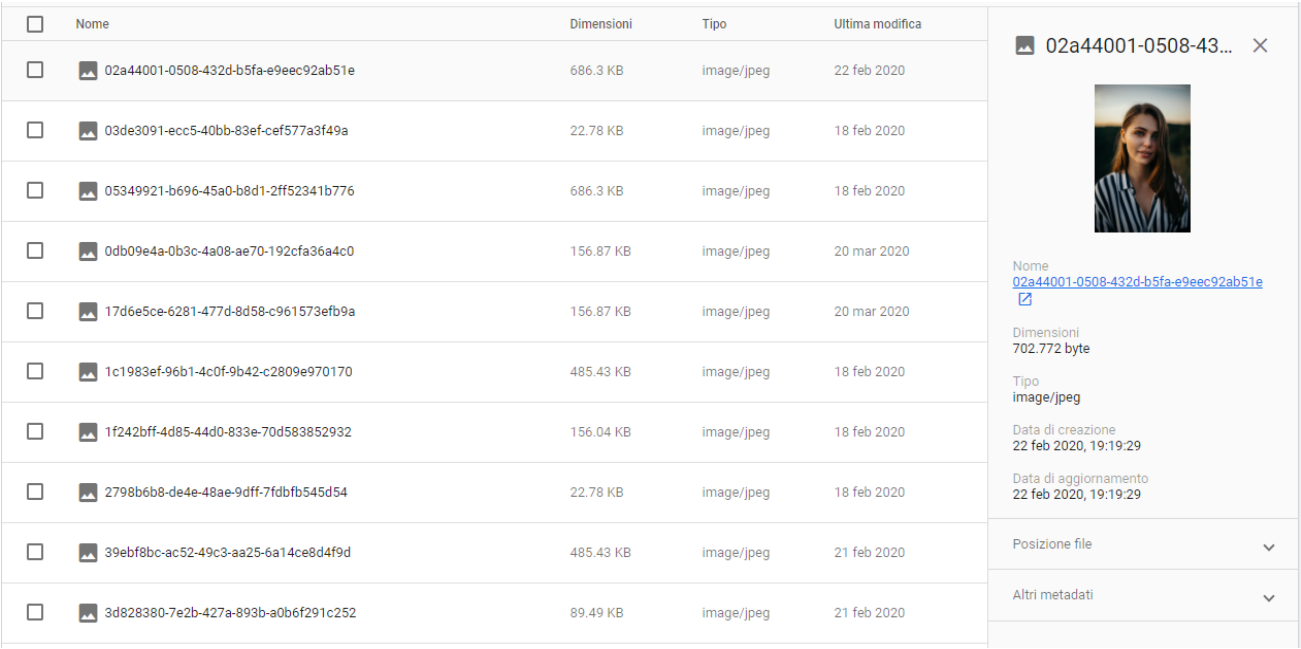

Figura 2.8: Elenco di immagini relative alla mia applicazione memorizzate nello storage di Firebase

2.7 DATABASE NoSQL / RELAZIONALE

I classici database relazionali trattano i dati memorizzandoli in strutture fisse, le tabelle, dove ogni dato contenuto in essa rappresenta un'informazione solo se associato al suo corrispondente attributo.

La tabella deve quindi avere una o più colonne per rappresentare la chiave primaria, essenziale per identificare una riga rispetto alle altre. Tra le tabelle vi sono poi le relazioni e con operazioni logiche quali il Join si mettono in collegamento le tabelle tra loro utilizzando le chiavi esterne. Utilizzare tabelle SQL molto grandi è sconsigliato: il lavoro diventa molto costoso a livello computazionale e difficile da gestire.

I sistemi SQL devono quindi svolgere continue interrogazioni sui dati delle tabelle ed effettuare rigidi controlli sulla validità dei dati stessi per permettere l'integrità dell'intero database. Ovviamente il tutto a discapito della flessibilità (anche per questo motivo spesso i database relazionali si basano sugli schemi entità-relazione).

I database non relazionali invece trattano i dati in maniera diversa: vengono conservati in documenti anziché in tabelle. Un oggetto JSON accennato poco prima, non è altro che un documento da inserire nel database. Non esisteranno più tabelle ma collezioni di documenti [4]. Ogni documento aggregato raccoglie tutti i dati associati ad un'entità, permettendo quindi a qualsiasi applicazione di trattare l'entità come oggetto e valutare contemporaneamente tutte le informazioni ad essa correlate.

Nel caso di DBMS NoSQL a documenti, essendo assenti le relazioni, esistono meccanismi per collegare le informazioni: embedding e referencing. Il primo consiste nell'annidare un oggetto JSON all'interno di un altro, mentre il secondo fa in modo che un documento contenga tra i propri dati l'id di un altro documento.

La figura 2.9 mostra uno schema riassuntivo che evidenzia le principali differenze tra i due tipi di database.

COMMON COMPARISONS BETWEEN

MYSQL & NOSQL

	MySQL	NoSQL
Nature	Relational Database	Non-Relational Database
Design	Based on the concept of tables	Based on the concept of documents
Scalable	Tough to scale due to its relational nature	Easily scalable big data compared to relational
Model	Detailed database model is needed before creation	No need of a detailed database model
Community	Vast community available	Community is growing rapidly, but still smaller compared to MySQL
Standardization	SQL is standard language	Lacks standard query language
Schema	The Schema is rigid	The Schema is dynamic
Flexibility	Not very flexible in terms of design	Very flexible in terms of design
Insertions	Inserting new columns or fields affect the design	No effect on the design with the insertion of new columns or fields

Figura 2.9: Schema che mostra le principali differenze tra un database relazionale ed un database NoSQL

Analizzando le differenze e dopo varie ricerche, si possono già osservare i principali vantaggi di una soluzione NoSQL rispetto ad una SQL.

- Leggerezza computazionale: in un ambiente SQL il peso computazionale cresce con l'ingigantirsi della base di dati, del numero di tabelle e quindi informazioni da trattare. Nel NoSQL non vi è invece nessun limite di dimensioni in questo senso.
- Assenza di schema: con l'assenza delle tabelle manca anche lo schema, i database no SQL sono infatti definiti *schemaless* in quanto il documento JSON contiene tutti i campi necessari e l'aggiunta di nuovi dati ed informazioni nella nostra applicazione non compromette l'integrità del database.

Si tratta di vantaggi non trascurabili che, insieme allo stimolo ed alla curiosità di utilizzare nuovi strumenti non ancora visti nel corso di studi, mi hanno portato ad optare per un database NoSQL.

2.8 GEOFIRE

Come tecnologia di localizzazione è stata scelta Geofire, una libreria open-source che permette di memorizzare ed interrogare un set di oggetti in base alla loro posizione geografica. Una funzione fondamentale della mia applicazione è appunto quella di localizzare i dispositivi e gli utenti in modo tale da poter ricavare utenti nelle vicinanze o effettuare ricerche in base alla loro posizione. Geofire è una diretta estensione di Firebase, motivo per cui ho deciso di utilizzare questo strumento.

2.8.1 Geofire e Firebase

Come già accennato, Geofire si integra perfettamente con Firebase: le location dei dispositivi vengono immagazzinate nel Realtime Database di Firebase e vengono aggiornate in tempo reale.
All'interno del database si va a creare quindi un nodo che conterrà tutte le location dei dispositivi, ognuna di esse identificata da una chiave ed espressa attraverso la coppia latitudine-longitudine.
La vera utilità di Geofire sta quindi nel fatto di poter interrogare i dati attraverso una

GeoQuery [5], così da ottenere per esempio solo i dati entro un certo raggio espresso in metri. Nella figura 2.10 viene mostrato come Geofire memorizza i dati all'interno del database, mentre le funzioni di localizzazione saranno spiegate nello specifico successivamente.

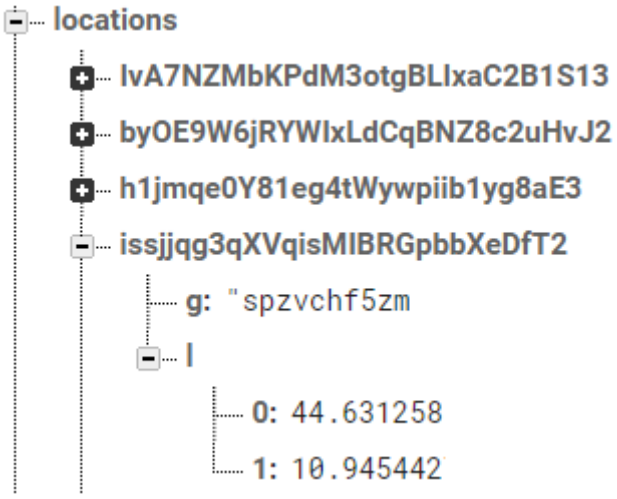

Figura 2.10: Come Geofire memorizza i dati all'interno del Realtime Database

2.9 PICASSO

Picasso è una libreria per Android per la gestione di immagini [6]. Nello specifico si occupa del caricamento semplificando il processo di visualizzazione delle immagini caricate da posizioni remote. Nel nostro caso si è rivelata una libreria fondamentale per caricare le immagini dal Firebase Storage, per esempio per la corretta visualizzazione delle immagini profilo.

Come ogni altra libreria, Picasso deve essere aggiunta tra le dipendenze del file build.gradle e per utilizzarla basta avere un ImageView nel quale inserire l'immagine caricata.

Picasso include varie funzionalità, come la trasformazione di immagini per rispettare un certo layout ed il supporto di rotazione e ridimensionamento.

2.10 KOTLIN

Come già detto nella fase introduttiva della tesi, il linguaggio di programmazione utilizzato per lo sviluppo dell'applicazione è Kotlin. La motivazione ricade puramente nella volontà e curiosità di apprendere un linguaggio non visto durante il corso di studi. Ho effettuato svariate ricerche riguardo a questo nuovo linguaggio, le quali mi hanno definitivamente convinto a scegliere Kotlin invece di Java.

2.10.1 Introduzione a Kotlin

Si tratta di un linguaggio moderno, rilasciato nel 2016, che solo dopo un anno è stato dichiarato da Google come linguaggio ufficiale Android tanto quanto Java (è infatti integrato in Android Studio) [7]. Il linguaggio è stato sviluppato, come spesso accade, in ambito aziendale e non accademico [8] : è nato all'interno di un team di sviluppatori che dopo anni di esperienza acquisita con Java ed altri linguaggi, hanno realizzato un linguaggio mirato a risolvere problematiche concrete e ben conosciute dagli stessi sviluppatori.
Ispirandosi alle problematiche di Java e facendo uso di buone regole introdotte da altri linguaggi imperativi, Kotlin è stato modellato dagli sviluppatori in modo tale da contenere varie funzionalità utili sia a livello sintattico che prestazionale, dando così al programmatore una serie di strumenti, caratteristiche ed implementazioni tanto semplici quanto potenti.

2.10.2 Principali caratteristiche

Kotlin è un linguaggio sintetico, altamente espressivo, potente e dominato da una mentalità moderna, caratteristiche che ne hanno determinato il successo. A determinarne il successo è sicuramente anche l'ottima integrazione del suo plugin con Android Studio. Esso supporta il programmatore in ogni fase della stesura del codice, proponendo soluzioni più concise ed eleganti o addirittura allertandolo in caso di possibili eccezioni a tempo di esecuzione.
A livello di codice lo si può definire come un linguaggio di programmazione general purpose, multi-paradigma e con una **tipizzazione statica e forte**, orientato verso la programmazione ad oggetti.
È importante sottolineare che Kotlin deriva comunque da Java e quindi uno può essere usato

in maniera alternativa all'altro: sono **interoperabili**, altra caratteristica da non sottovalutare. Entrambi i linguaggi possono infatti coesistere all'interno dello stesso progetto e dal momento che tutti e due producono un *Java Bytecode*, è possibile richiamare metodi e parti di codice da Java a Kotlin e viceversa.
Vediamo di seguito alcune delle funzionalità ed altre caratteristiche peculiari di Kotlin che rappresentano dei motivi per cui ha ottenuto un consenso importante, specialmente nel mondo Android, e che possono fare la differenza nella scelta tra i vari linguaggi di programmazione:

- Kotlin è completamente **open-source**: il suo codice sorgente è reperibile su github e può essere usato da qualsiasi sviluppatore in maniera totalmente gratuita.

- Linguaggio **conciso**: si è calcolato che, in media, la quantità di codice scritto in Kotlin è del 40% inferiore a quella di codice scritto in Java.

- **Totale compatibilità con la JVM** (Java Virtual Machine): scrivendo in Kotlin è possibile fare uso di tutte le librerie già sviluppate per java. Si ha così accesso ad una delle più grandi raccolte di framework e funzioni dell'intero ambito informatico.

Osservando il linguaggio secondo un'ottica più tecnica, strettamente legata al codice, si elencano alcune funzionalità che rappresentano una novità rispetto a molti altri linguaggi:

- **Dichiarazioni di variabili:** In Kotlin possono essere definite due tipologie diverse di variabili, *val* e *var*. La prima tipologia include tutte quelle variabili che una volta che vengono definite non possono essere più modificate in alcun modo, quindi il loro valore rimane costante. Le variabili *var* invece non sono soggette a questa restrizione e il loro valore può essere modificato ogni qualvolta lo si desidera all'interno del codice. Come già detto, Kotlin è un linguaggio a tipizzazione statica, ogni variabile ha quindi un tipo ben preciso, che può essere definito al momento dell'assegnazione:

```
var nome_variabile: tipo variabile
val variabile_immutabile: Int = 100
```

- **Null Safety:** La Null Pointer Exception è sicuramente una delle exception più temute dai programmatori Java ma non solo. Si verifica quando l'applicazione tenta l'utilizzo di un oggetto o di un riferimento ad una variabile che contiene il valore null.
  Il sistema di tipi in Kotlin punta ad eliminare il pericolo che questo si verifichi, infatti ***in Kotlin,*** *dove tutti i tipi sono oggetti****, nessuna variabile è settabile di default a "null"***.
  Qualsiasi riferimento nel codice scritto in Kotlin ricade in una di due tipologie: *non-nullable* o *nullable*.
  Per fare in modo che una variabile accetti null come valore si utilizza l'operatore "?".
  Esempio 1 : `val x: Int? = null`
  Esempio 2: `val x: Int`
  Facendo così, nel caso di variabile non-nullable non c'è motivo di temere una *NullPointerException*, mentre ogni volta che si vuole utilizzare la variabile nullable è necessario controllare che essa non sia null attraverso un *if statement*. Alternativamente si usa sempre l'operatore "?" dopo la variabile, che nel caso di riferimento nullo restituisce *null* senza lanciare eccezioni. Si tratta di una soluzione più concisa per evitare il crash dovuto alla Nullpointerexception.
  Si utilizza invece l'operatore *!!* per imporre al compilatore di accettare l'istruzione immessa e di lanciare l'eccezione nel caso di riferimento nullo: questo operatore viene usato quando si è totalmente certi che la variabile in questione sia valida.

- **Classi**: Proprio come Java, Kotlin è orientato agli oggetti e ne mantiene i concetti fondamentali della programmazione. Nonostante questo, molte delle verbosità che caratterizzano la maggior parte dei linguaggi a oggetti vengono rimosse.
  La procedura per istanziare in Kotlin una classe è molto semplice ed intuitiva: occorre chiamare direttamente il costruttore, abbandonando l'utilizzo di parole chiave aggiuntive, come l'utilizzatissima "*new*" in Java. Un semplice esempio di dichiarazione di una classe in Kotlin è il seguente:

```
class nomeClasse
{
      private var nomeProprieta: int

      public fun nomeMetodo(val nomeAttributo): valoreRitorno
      {
            //Codice metodo
      }
 }
```

Le proprietà definiscono lo stato di un oggetto che appartiene alla classe, mentre i metodi definiscono il modo in cui altri elementi esterni del codice possono interagire con l'oggetto.

- **Costruttori:** Qualsiasi classe può definire i metodi costruttori: implementano il codice che viene eseguito non appena un oggetto appartenente alla classe viene creato.
  In Kotlin esistono costruttori primari e secondari, quello primario è uno ed uno solo ed è quello chiamato subito dopo la creazione di un oggetto. Non è necessario che il costruttore contenga codice:

  ```
  class NomeClasse costruttore (nomeVar: String){
     //codice della classe
  }
  ```

- Un elemento caratteristico di Kotlin che personalmente ho avuto piacere nello sfruttare è il concetto di **companion object.** Si tratta di un costrutto che permette di richiamare un metodo o una variabile senza dover necessariamente istanziare una classe. È un'alternativa funzionalmente equivalente ad un metodo o variabile statica, anche se il modo i cui lavorano internamente è completamente diverso. In sostanza possiamo accedere ad un oggetto membro di una classe dichiarato come companion object utilizzando semplicemente il nome della classe.
  Un esempio di ciò appena detto:
  ```
  class Person {
  ```

```
  // il nome del companion object può essere omesso
 companion object {
    fun callMe() = println("Sono stata chiamata.")

  }

}
fun main(args: Array<String>) {

    Person.callMe()

}
```

In questo caso è possibile chiamare il metodo *callMe()* usando direttamente il nome della classe:

```
Person.callMe()
```

La figura 2.11 racchiude i principali vantaggi nell'utilizzare Kotlin come linguaggio di programmazione.

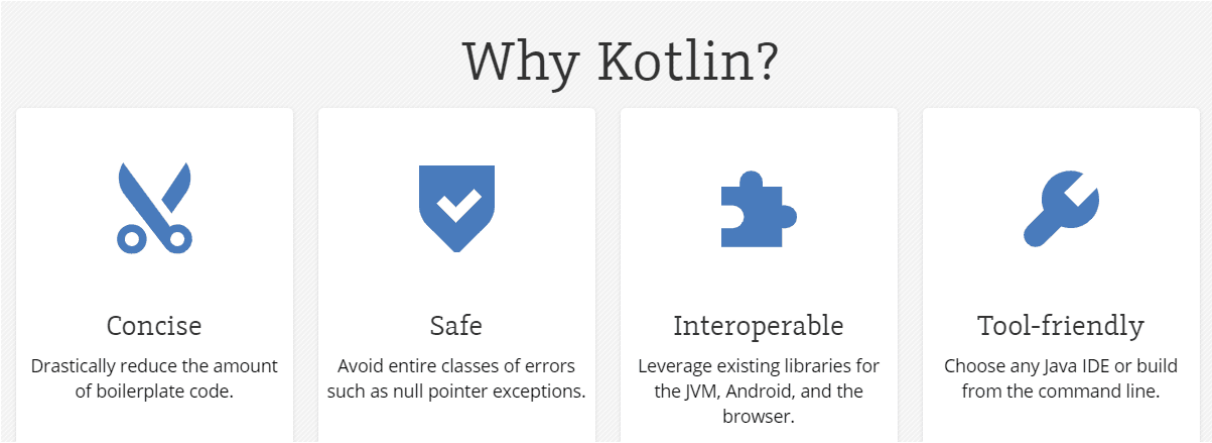

Figura 2.11: Caratteristiche fondamentali di Kotlin

CAPITOLO 3

PROGETTAZIONE

3.1 ANALISI DEI REQUISITI

L'analisi dei requisiti è una fase critica della nascita dell'applicazione e più in generale dello sviluppo software. Per requisito si intende una caratteristica del sistema e lo scopo che si prefigge questa analisi è quello di formalizzare ciò che verrà sviluppato e implementato nelle fasi successive del progetto. I requisiti più generalmente specificano quindi cosa deve essere fatto, indipendentemente dalla tecnologia, dall'architettura o linguaggio di programmazione.

3.1.1 Funzionalità di base

Il primo passaggio è stato quello di analizzare le operazioni principali che l'applicazione dovrà svolgere, ovvero quell'insieme di meccanismi che costituiscono le funzionalità offerte per raggiungere uno scopo.
Il "cuore" dell'applicazione è determinato dalle seguenti funzionalità:

- **Registrazione**: La fase di registrazione è indispensabile sia per chi offre un'attività che per un normale utente. Questa fase avviene attraverso l'inserimento obbligatorio di uno username, di una password, di una e-mail valida e di un'immagine profilo.
- **Login**: L'applicazione dà ovviamente la possibilità agli utenti già registrati di effettuare il login per accedere alle altre funzionalità attraverso la richiesta di e-mail e password.
- **Visualizzazione utenti vicini**: Attraverso meccanismi di geolocalizzazione si dà la possibilità agli utenti di visualizzare una lista contenente i profili degli utenti che si trovano entro un raggio in km, specificato dall'utente stesso.
- **Visualizzazione, modifica ed eliminazione profilo**: Gli utenti devono essere in grado di poter modificare le informazioni inserite durante la fase di registrazione, come per esempio l'immagine profilo, ed eventualmente di cancellarsi.
- **Messaggistica**: L'applicazione deve poter far comunicare gli utenti tra loro attraverso un sistema di chat.
- **Logout**: Gli utenti possono in qualsiasi momento effettuare il logout dall'applicazione

3.1.2 Caso d'uso

Nell'ambito dell'analisi dei requisiti rientra il diagramma dei casi d'uso, un metodo per effettuare in maniera non ambigua la raccolta dei requisiti, focalizzandosi sugli attori (chi ha a che fare col sistema) e su cosa viene fatto (caso d'uso). Il risultato finale sarà una

descrizione degli scenari di utilizzo dell'applicazione da parte degli attori che la utilizzano, avendo così una visione ad alto livello di quello che l'applicazione può fare, senza nessun dettaglio implementativo.

Nel caso specifico della mia applicazione l'attore è l'utente, più specificatamente un cliente o un esperto, mentre i casi d'uso sono rappresentati dalle funzionalità citate nel paragrafo precedente.
Nella figura 3.1 viene mostrato il diagramma dei casi d'uso del software, riassumendo in maniera schematica le azioni fondamentali dell'applicazione, considerando anche ciò che esse comportano e ponendo condizioni per evitare situazioni senza senso logico.

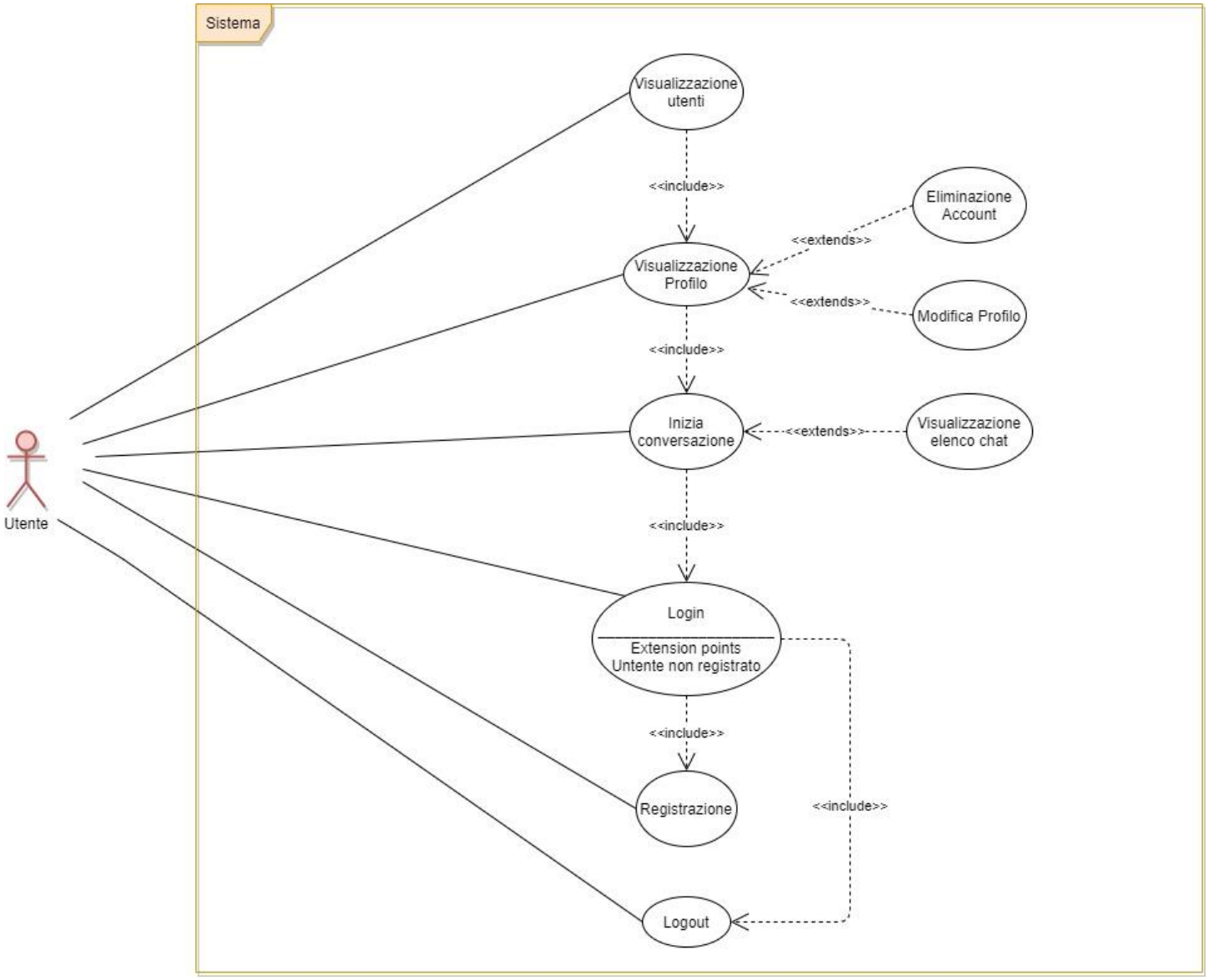

Figura 3.1: Diagramma di caso d'uso dell'applicazione

3.1.3 Diagramma delle attività

Si tratta di un diagramma in UML che modella le attività relative ad una collezione di entità secondo un determinato flusso, descrivendo quindi un processo.

Nel nostro caso vogliamo andare a modellare il flusso di un caso d'uso dell'applicazione.

I principali elementi che compongono un diagramma delle attività sono i seguenti:

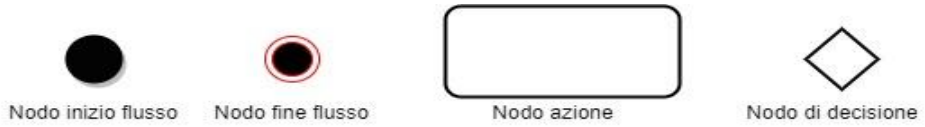

Figura 3.2: Legenda del diagramma attività

Con l'utilizzo di questi componenti possiamo rappresentare le transizioni del flusso di un caso d'uso, descrivendo quindi il comportamento dinamico del sistema considerando i flussi alternativi. La figura 3.3 nella pagina seguente mostra il diagramma delle attività relativo all'applicazione.

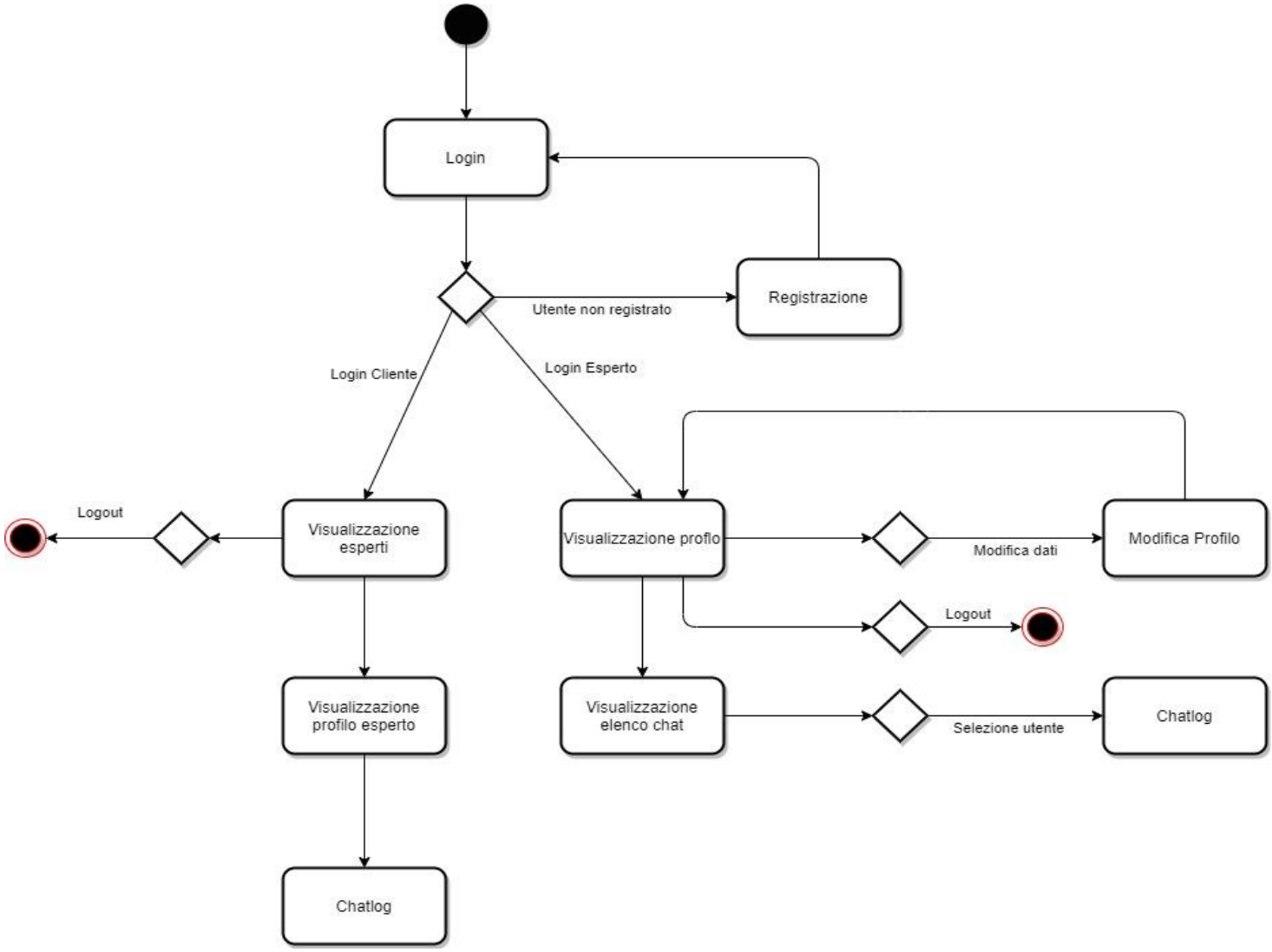

Figura 3.3: Diagramma delle attività completo

3.2 PROGETTAZIONE DATABASE

Come già spiegato nella sezione delle tecnologie utilizzate, si è scelto Firebase come database. La fase di progettazione del database non è quindi avvenuta con l'ausilio di uno schema Entità-Relazioni, strumento incongruente con la filosofia di un database NoSQL.

La progettazione, nel nostro caso, è stata effettuata seguendo la logica NoSQL, dove i dati sono immagazzinati all'interno dei già citati file JSON (file esportabile direttamente dalla console di Firebase del progetto).

La figura 3.4 mostra una rappresentazione schematica delle entità che costituiscono tale file, seguita dal contenuto del file stesso con dati d'esempio.

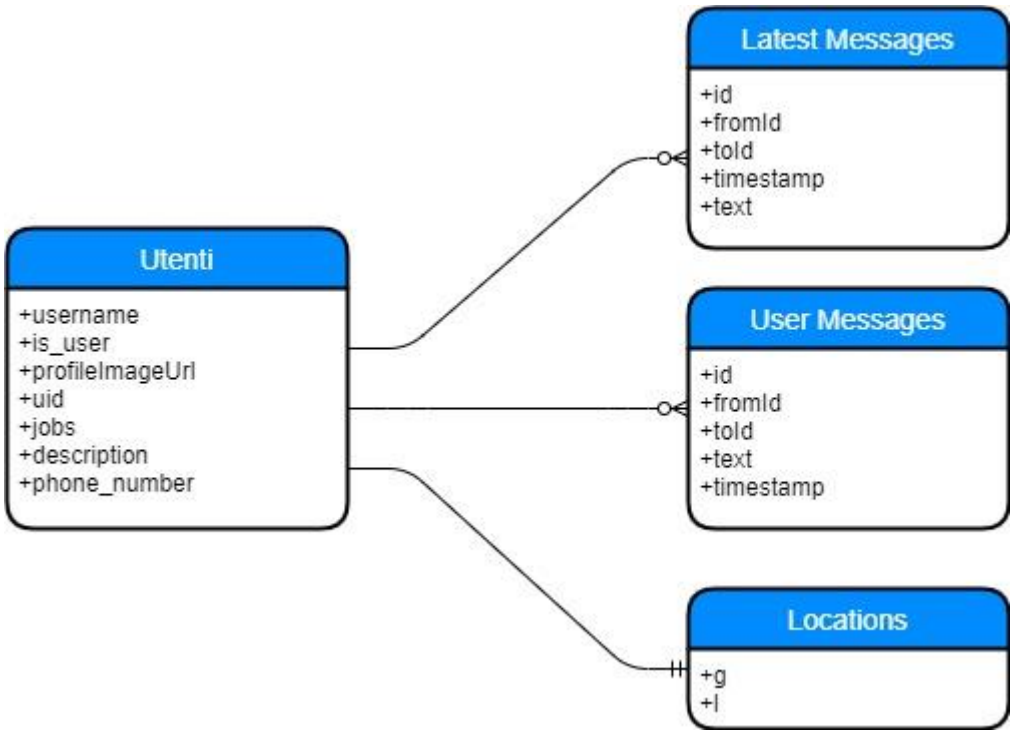

Figura 3.4: Schema entità database

{
  "Utenti" :
"4vDrGT07YFcUtacZDyFA9txpdI92" : {
    "description" : "I am an hairdresser",
    "isuser" : false,
    "phone_number" : "123123",
    "profileImageUrl" : "https://firebasestorage.googleapis.com/v0/b/chat-kotlin-391b8.appspot.com/o/images%2F75196858-b3f1-4ae1-9714-87829948c3f6?alt=media&token=a16cd115-7b55-48c2-889e-50b621906c58",
    "uid" : "4vDrGT07YFcUtacZDyFA9txpdI92",
    "username" : "p2",
    "jobs" : "parrucchiere donna"
  },
"GW3nbS7YnxU064T9gWLX1WRPcL22" : {
    "description" : "I am USER",
    "isuser" : true,
    "phone_number" : "",
    "profileImageUrl" : "https://firebasestorage.googleapis.com/v0/b/chat-kotlin-391b8.appspot.com/o/images%2F2cf0e325-f06c-4f8c-9dac-58a4c39701e3?alt=media&token=63ebbf3d-7601-4548-a499-274704280f3a",
    "uid" : "GW3nbS7YnxU064T9gWLX1WRPcL22",
    "username" : "USER",
    "jobs" : ""
  },
"latest-messages" : {
  "0FwlToRRcIUbkAIMXyCgdt5BPva2" : {
    "SIxplHfaQaWFJc2ElySu2AeCAIy1" : {
      "fromId" : "0FwlToRRcIUbkAIMXyCgdt5BPva2",
      "id" : "-M0Hdv0DzPpiGYxD71u-",
      "text" : "Ciao hai tempo fra 1 ora?",
      "timestamp" : 1581934686,
      "toId" : "SIxplHfaQaWFJc2ElySu2AeCAIy1"
    }
  },
"locations" : {
  "4vDrGT07YFcUtacZDyFA9txpdI92" : {
    ".priority" : "9q9hvumnuw",
    "g" : "9q9hvumnuw",
    "l" : [31.4219983, -125.084]
  },
  "GW3nbS7YnxU064T9gWLX1WRPcL22" : {
    ".priority" : "spzvbym0d3",
    "g" : "spzvbym0d3",
    "l" : [44.6389196, 10.9383601]
  },
"user-messages" : {
  "GW3nbS7YnxU064T9gWLX1WRPcL22" : {
    "LZxS6jC8mtclZBycluvSnh83n9r2" : {
      "-M0IuG52BE-FWywYS-5b" : {
        "fromId" : "GW3nbS7YnxU064T9gWLX1WRPcL22",
        "id" : "-M0IuG52BE-FWywYS-5b",
        "text" : "Ciao",
        "timestamp" : 1581955747,
        "toId" : "LZxS6jC8mtclZBycluvSnh83n9r2"
      }

```
    }
  },
```

3.2.1 Costruzione delle entità

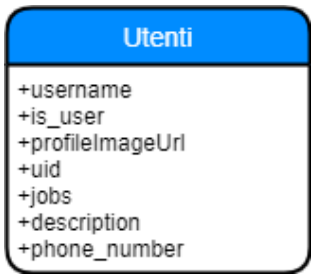

Figura 3.5: Entità Utenti

L'entità Utenti rappresenta un utente registrato all'interno dell'applicazione. L'attributo is_user determina, attraverso una variabile booleana, se l'utente è un normale utilizzatore oppure un esperto che offre un'attività. La chiave che contraddistingue un utente da un altro è "uid", fondamentale per riconoscere e gestire le azioni dell'utente nell'applicazione. L'attributo username rappresenta appunto il nome scelto al momento della registrazione. Importante sottolineare che gli attributi "jobs", "description" e "phone_number" sono attributi esclusivi di quegli utenti il cui "is_user" è settato a false.

Figura 3.6: Entità Locations

L'entità Locations è creata nel database direttamente da Geofire ogni qualvolta viene chiamata la funzione di geolocalizzazione. In Locations abbiamo quindi una chiave per ogni utente , dove per ognuno di essi viene memorizzata in "g" la chiave del luogo, ovvero un codice che identifica coordinate specifiche, oltre alle coordinate stesse immagazzinate invece in "l". Tutte le volte che un utente aggiorna la propria posizione, i dati contenuti in Locations per quell'utente vengono aggiornati di conseguenza.

Figura 3.7: Entità User Messages

In User Messages vengono immagazzinati i messaggi scambiati tra gli utenti. Ogni messaggio contiene tutte le informazioni necessarie per poterlo definire: l'id che lo identifica in maniera univoca, il fromId e il toId per individuare chi sono il mittente ed il destinatario del messaggio, un timestamp memorizzato al momento dell'invio ed infine il corpo del messaggio stesso.
È presente un'istanza di quest'entità per ogni altro utente con cui un utente ha chattato.

Figura 3.8: Entità Latest Messages

Molto simile all'entità precedente, Latest Messages rappresenta l'ultimo messaggio inviato per ogni utente. Quest'entità è servita per far in modo che gli utenti potessero visualizzare in una schermata tutti gli ultimi messaggi dagli utenti con cui è entrato in contatto. Supponendo di essere loggati nell'applicazione, se un utente ci contatta, il "Latest Message" con fromId dell'utente che ci ha contattati viene sovrascritto con le informazioni dell'ultimo messaggio, al contrario di ciò che avviene per gli "user Messages" dove tutti i messaggi vengono invece memorizzati.

La costruzione del diagramma è molto intuitiva in quanto si è cercato di ridurre il numero di entità memorizzate per evitare di sovrappopolare il database con informazioni non necessarie per il corretto funzionamento dell'applicazione. Da come si può vedere nella figura 3.4 ogni user può avere da 0 a N messaggi, 0 nel caso non abbia scritto a nessuno o nel caso non sia

stato contattato da nessuno, mentre deve obbligatoriamente avere una ed una sola location memorizzata, aggiornata ogni volta con le coordinate che rappresentano la sua posizione attuale.

CAPITOLO 4
IMPLEMENTAZIONE

4.1 INTRODUZIONE ALL'IMPLEMENTAZIONE

Questo capitolo tratterà tutti i dettagli implementativi della creazione dell'applicazione. Si osserveranno uno alla volta tutti i file che compongono il progetto, ripercorrendo così il percorso che ha portato all'ultimazione dell'applicazione. Come detto in precedenza, si è utilizzato Android Studio come IDE, Kotlin come linguaggio di programmazione e Firebase per gestire i dati. Le operazioni di controllo e di test per verificare la correttezza delle funzionalità dell'applicazione sono state effettuate inizialmente su un emulatore virtuale, il quale però si è rivelato molto time-consuming. Ho quindi approfittato di Android Studio che mi ha permesso di svolgere queste operazioni direttamente sul mio dispositivo collegato al computer, funzionalità che ho personalmente trovato davvero comoda e che ha velocizzato lo sviluppo stesso dell'applicazione. Il dispositivo in questione è un LG-H930 con sistema operativo Android versione 9 "Pie".

Verranno mostrati per primi la struttura del progetto e il file Manifest, dopodiché il capitolo navigherà tra le varie activity soffermandosi sulle componenti, l'interfaccia e le funzionalità offerte.

4.2 STRUTTURA PROGETTO

Nel capitolo 2 è stato illustrato come Android Studio organizza una generale struttura di un progetto. In questo paragrafo verranno illustrate e analizzate quindi le principali cartelle e i file che le compongono.

In figura 4.1 vengono mostrati i packages che contengono i file *.kt*, ovvero i file con il codice kotlin raggruppati secondo le funzionalità che offrono. Il package *java* è automaticamente creato da Android Studio anche se i file all'interno sono scritti in kotlin.

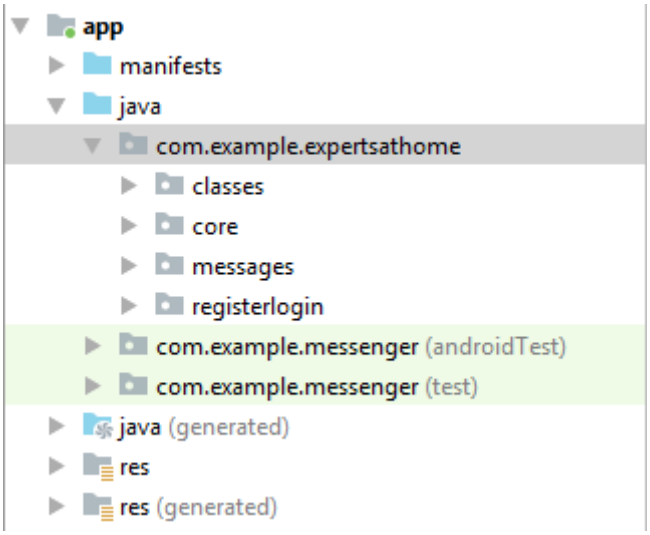

Figura 4.1: Esempio struttura di un qualsiasi progetto Android in Android Studio

- **classes**: Questa cartella contiene la definizione delle classi principali utilizzate all'interno dell'applicazione. Immagazzinare le classi più utilizzate all'interno di un unico package risulta molto comodo nella stesura del codice ed evita di ridefinirle ogni volta che vengono usate. Le classi in questione sono quelle rappresentanti gli oggetti per la manipolazione dei dati, come per esempio un oggetto cliente ed i relativi attributi, costituendo una vera e propria struttura dati dell'applicazione.

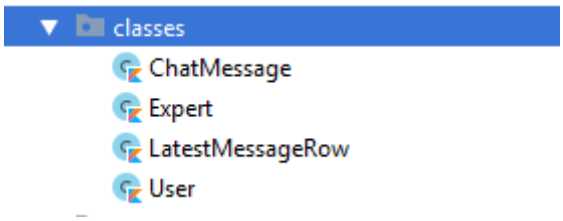

Figura 4.2: cartella classes

- **core:** La cartella core contiene all'interno la definizione delle principali activity dell'applicazione, quindi di quelle classi che hanno il compito di gestire il loro ciclo di vita e che si occupano del caricamento delle interfacce e della corretta visualizzazione degli elementi che le compongono.

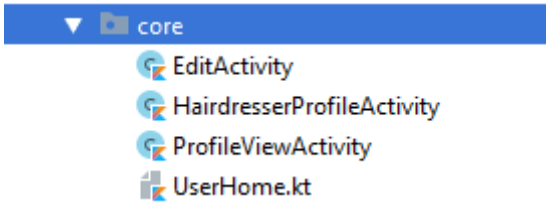

Figura 4.3: cartella core

- **messages:** In questo package troviamo le due classi che si occupano delle funzionalità di messaggistica dell'applicazione e della gestione delle activity ad esse legate.

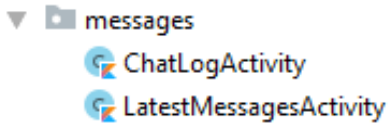

Figura 4.4: cartella messages

- **registerlogin:** in quest'ultima cartella vengono raggruppate le classi che si dedicano alle funzionalità di registrazione e di login, gestendo l'interfaccia e interagendo col database per memorizzare gli utenti dell'applicazione.

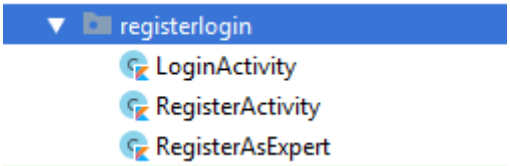

Figura 4.5: cartella registerlogin

Per quanto riguarda la struttura del progetto viene mostrato nella figura 4.6 il contenuto della cartella *res,* di cui si parla nel capitolo 2 relativo alle tecnologie utilizzate.

I file *.xml* in *drawable* contengono componenti estetici utilizzati nell'applicazione, come per esempio i bottoni arrotondati. La cartella *layout* contiene invece tutti quei file che definiscono il layout grafico delle activity, quindi le rispettive interfacce, ciò che l'utente visualizza sullo schermo. In values vengono immagazzinati valori spesso utilizzati nell'applicazione, come colori o font di testo, in modo da non doverli definire ogni qualvolta vengono utilizzati.

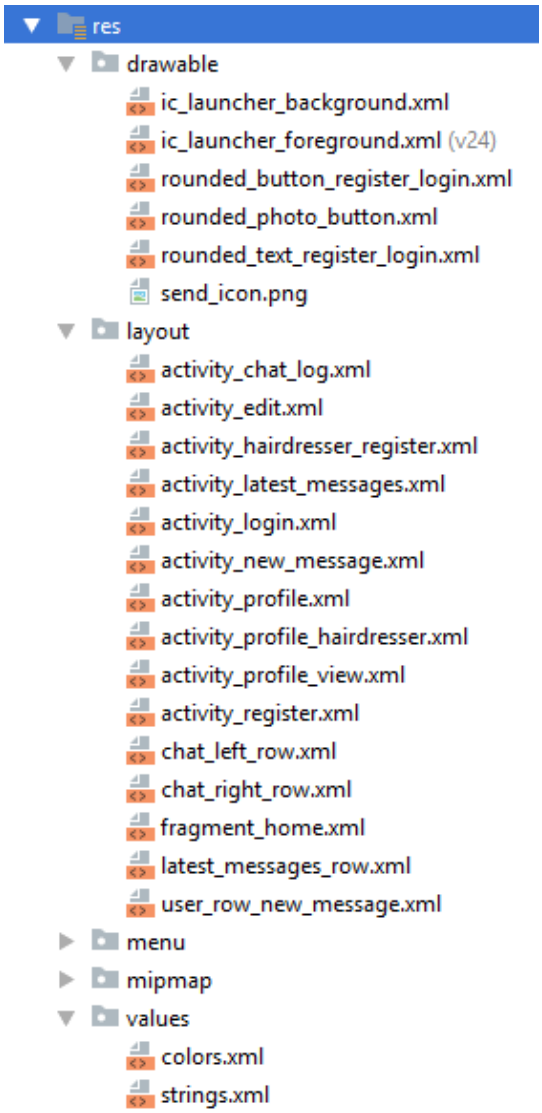

Figura 4.6: Cartella res con relative sottocartelle (drawable, layout e values)

4.3 IMPLEMENTAZIONE DELL'APPLICAZIONE

Verranno mostrati ed esplicati uno alla volta i file principali di cui si compone il progetto, partendo dal manifest file, soffermando l'attenzione su parti importanti di codice e sulle funzionalità di ognuna delle activity.

4.3.1 Manifest File

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.messenger">

   <uses-permission android:name="android.permission.INTERNET" />
   <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
   <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />

   <application
      android:allowBackup="true"
      android:icon="@mipmap/ic_launcher"
      android:label="Experts at home"
      android:roundIcon="@mipmap/ic_launcher_round"
      android:supportsRtl="true"
      android:theme="@style/AppTheme">
      <activity android:name="com.example.expertsathome.core.ProfileViewActivity">
         <meta-data
            android:name="android.support.PARENT_ACTIVITY"
            android:value="com.example.expertsathome.core.UserHome" />
      </activity>
      <activity android:name="com.example.expertsathome.core.EditActivity">
         <meta-data
            android:name="android.support.PARENT_ACTIVITY"
            android:value="com.example.expertsathome.core.HairdresserProfileActivity" />
      </activity>
      <activity
android:name="com.example.expertsathome.core.HairdresserProfileActivity"></activity>
      <activity android:name="com.example.expertsathome.registerlogin.RegisterAsExpert"
/>
      <activity android:name="com.example.expertsathome.messages.ChatLogActivity">
         <meta-data
            android:name="android.support.PARENT_ACTIVITY"
            android:value="com.example.expertsathome.messages.LatestMessagesActivity"
/>
      </activity>
      <activity android:name="com.example.expertsathome.core.UserHome"></activity>
      <activity
android:name="com.example.expertsathome.messages.LatestMessagesActivity">
         <meta-data
            android:name="android.support.PARENT_ACTIVITY"
            android:value="com.example.expertsathome.core.UserHome" />
      </activity>
      <activity android:name="com.example.expertsathome.registerlogin.RegisterActivity">
```

```
                <intent-filter>
                    <action android:name="android.intent.action.MAIN" />

                    <category android:name="android.intent.category.LAUNCHER" />
                </intent-filter>
            </activity>
            <activity android:name="com.example.expertsathome.registerlogin.LoginActivity" />
        </application>

</manifest>
```

Come già accennato nel capitolo 2 che trattava le tecnologie utilizzate, in *AndroidManifest.xml* troviamo le informazioni di base del progetto. Le prime righe definiscono i permessi che l'applicazione deve avere sul dispositivo per poter utilizzare tutte le funzionalità. Nello specifico si tratta dei permessi GPS necessari per l'utilizzo di Geofire dove ACCESS_COARSE_LOCATION permette di ricavare un'approssimazione della location mentre ACCESS_FINE_LOCATION permette di ottenere coordinate precise. Si usano entrambe per riuscire a memorizzare l'ultima posizione conosciuta, metodo per risparmiare batteria sui dispositivi mobili. Nella sezione *application* troviamo le specifiche dell'applicazione, come per esempio il logo che verrà visualizzato sui dispositivi o il nome dell'app. Dopodiché vengono dichiarate tutte le activity che l'applicazione possiede con eventuali rispettive etichette che ne contraddistinguono il funzionamento, come per esempio quale delle activity deve essere lanciata all'avvio dell'applicazione o quale activity bisogna visualizzare alla pressione del tasto *back* sul dispositivo.

4.3.2 Register Activity

Guidando il lettore attraverso le varie activity che compongono l'applicazione, la prima di esse in cui ci si imbatte è la RegisterActivity. Come si intuisce dal nome si tratta della prima cruciale fase che riguarda la registrazione di un utente all'interno dell'applicazione. È uno dei passaggi in cui si è prestata una particolare attenzione nel guidare l'utente ad una registrazione semplice ed intuitiva. Vediamo nella figura 4.7 la schermata con cui si presenta l'applicazione appena la avviamo la prima volta oppure ogni qualvolta non si abbia ancora effettuato l'accesso al suo interno.

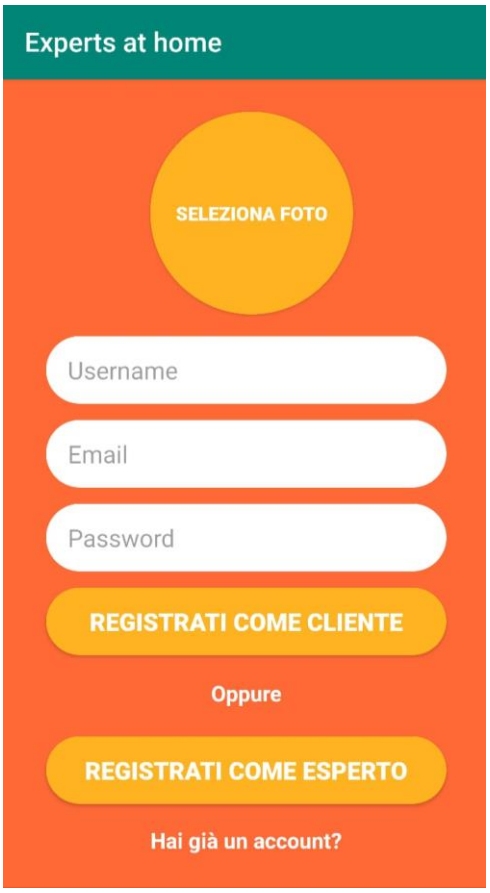

Figura 4.7: Schermata iniziale dell'applicazione che mostra il format di registrazione

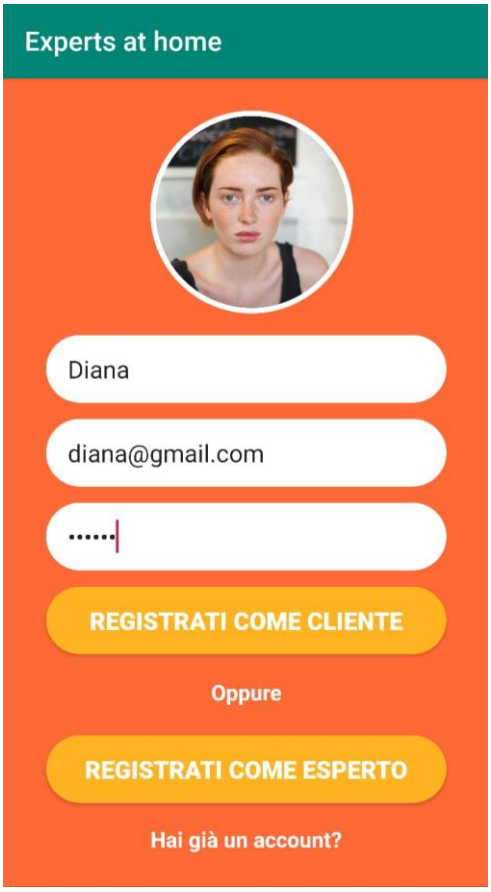

Figura 4.8: Register Activity con campi completati con dati d'esempio

Ognuno dei campi del format di registrazione va obbligatoriamente compilato in quanto tutti costituiscono le informazioni essenziali che definiscono l'utente

all'interno del database di Firebase. Nel caso in cui un campo non venga inserito, si inserisca una mail non valida o una password troppo corta, il sistema avvisa l'utente con un messaggio d'errore. Dopo aver compilato i campi obbligatori l'utente deve completare la registrazione decidendo se registrarsi come cliente oppure come un utente che offre un'attività lavorativa da svolgere in casa. Una variabile booleana memorizzerà la scelta e indirizzerà di conseguenza l'utente nella successiva schermata coerente alla sua scelta. Per la definizione di questa variabile si è scelto di utilizzare il costrutto *companion object* spiegato nel capitolo 2 nella sezione che parla di Kotlin: è un modo semplice ed efficace per definire una variabile con visibilità globale.

```
companion object{
    var isExpert : Boolean? = null
}


override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_register)

    register_registration.setOnClickListener {

        isExpert = false
        if(selectedPhotoUri != null)
            performRegister()
        else Toast.makeText(this, "Inserisci la tua immagine
profilo",Toast.LENGTH_SHORT).show()

    }

    già_account_registration.setOnClickListener {
        Log.d("RegisterActivity" , "Try to show login activity")
        val intent = Intent(this, LoginActivity::class.java)
        startActivity(intent)
    }

    photo_button_register.setOnClickListener {
        Log.d("RegisterActivity", "Try to show photo selector")

        val intent = Intent(Intent.ACTION_PICK)
        intent.type = "image/*"
        startActivityForResult(intent, 0)
    }

    register_as_expert.setOnClickListener {
        isExpert = true
        if(selectedPhotoUri != null)
            performRegister()
        else Toast.makeText(this, "Inserisci la tua immagine
profilo",Toast.LENGTH_SHORT).show()

    }
}
```

Notiamo che ad ogni bottone si assegna il rispettivo listener che chiama il metodo corrispondente. Alla pressione del bottone *"Registrati come esperto"* la variabile booleana assume il valore *true* prima di chiamare la funzione di registrazione del frammento di codice seguente:

```
private fun performRegister() {

    val email = email_registration.text.toString()
    val password = password_registration.text.toString()

    if (email.isEmpty() || password.isEmpty()) {
        Toast.makeText(this, "Inserisci email e password",
Toast.LENGTH_SHORT).show()
        return
    }

    Log.d("RegisterActivity" , "Email is: " + email)
    Log.d("RegisterActivity" , "Password is " + password)

    //Firebase Authentication
    FirebaseAuth.getInstance().createUserWithEmailAndPassword(email, password)
        .addOnCompleteListener {
            if (!it.isSuccessful) return@addOnCompleteListener

            //else if successful
            Log.d("RegisterActivity", "Successfully created user with uid:
${it.result?.user?.uid}")

            uploadImageToFirebaseStorage()
        }
        .addOnFailureListener{
            Log.d("RegisterActivity", "Failed to create user: ${it.message}")
            Toast.makeText(this, "Errore creazione utente: ${it.message}",
Toast.LENGTH_SHORT).show()
        }
}
```

L'applicazione quindi dopo aver controllato la correttezza di e-mail e password chiama il metodo per la creazione dell'utente su firebase attraverso *FirebaseAuth* e dopodiché viene memorizzata l'immagine profilo selezionata sullo storage di Firebase nel metodo *uploadImageToFirebaseStorage,* il quale verifica che un'immagine sia stata effettivamente selezionata. Se questa viene correttamente salvata nel percorso dello storage di Firebase, viene invocato il metodo per salvare definitivamente l'utente sul real-time database indicandone l'url dell'immagine scelta *:*

```
private fun uploadImageToFirebaseStorage() {
    if(selectedPhotoUri == null)
        return

    val filename = UUID.randomUUID().toString()
    val ref = FirebaseStorage.getInstance().getReference("/images/$filename")

    ref.putFile(selectedPhotoUri!!)
        .addOnSuccessListener {
            Log.d("RegisterActivity", "Successfully uploaded image:
${it.metadata?.path}")

            ref.downloadUrl.addOnSuccessListener {
                Log.d("RegisterActivity", "File Location $it")

                saveUserToFirebaseDatabase(it.toString()) //passo l'url alla
funzione
            }
        }
}
```

Avendo verificato la correttezza di ogni campo si procede con il salvataggio delle informazioni dell'utente, nel caso in cui questo si sia registrato come cliente non vi sono più dati da memorizzare e si può procedere di conseguenza alla visualizzazione della prossima activity, *UserHome.* Si definisce il percorso del real-time database in cui si vuole che le informazioni dell'utente vengano memorizzate creando un riferimento al database stesso. Creare un riferimento al database per poter effettuare operazioni su di esso è una procedura fondamentale per potervi interagire attraverso i metodi forniti da firebase. Creiamo quindi un oggetto di classe *Expert* e salviamo nel database nella posizione relativa al suo *uid* tutte le informazioni necessarie. Una nota importante riguarda il metodo di geolocalizzazione che individua la posizione attuale del dispositivo: nel caso di registrazione come cliente viene effettuato subito dopo la corretta memorizzazione dell'utente nel database. Vediamo anche come l'utilizzo degli intent discussi nel capitolo 2 sia fondamentale per la gestione delle activity,

specialmente nel passaggio da un'activity ad un'altra:

```
private fun saveUserToFirebaseDatabase(profileImageUrl: String) {
    val uid = FirebaseAuth.getInstance().uid?: "" //Di deafult è una stringa vuota

    var ref: DatabaseReference?
    val expert : Expert?

    if(isExpert != true){
        ref = FirebaseDatabase.getInstance().getReference("/Utenti/$uid")

        expert = Expert(
            uid,
            username_registration.text.toString(),
            profileImageUrl,
            "",
            true,
            "",
            ""
        )

        ref.setValue(expert)
            .addOnSuccessListener {
                Log.d("RegisterActivity" , "User saved to Firebase database")

                mFusedLocationClient = LocationServices.getFusedLocationProviderClient(this)
                getLastLocation()


                val intent = Intent(this, UserHome::class.java)
                intent.flags = Intent.FLAG_ACTIVITY_CLEAR_TASK.or(Intent.FLAG_ACTIVITY_NEW_TASK)
//per rimuovere dallo stack la precedente activity
                startActivity(intent)
            }
            .addOnFailureListener{
                Log.d("RegisterActivity","Failed to save user to database: ${it.message}")
            }
    }
```

In seguito, illustro i metodi di geolocalizzazione che vengono effettuati ogni qualvolta l'applicazione ricava la posizione attuale del dispositivo (tra questi troviamo anche i metodi che si occupano di richiedere autorizzazioni all'utente)

```
@SuppressLint("MissingPermission")
private fun getLastLocation() {
    if (checkPermissions()) {
        if (isLocationEnabled()) {

            mFusedLocationClient.lastLocation.addOnCompleteListener(this) { task ->
                var location: Location? = task.result
                if (location == null) {
                    requestNewLocationData()
                } else {

                    val uid = FirebaseAuth.getInstance().uid
                    val ref = FirebaseDatabase.getInstance().getReference("/locations")
                    val geoFire = GeoFire(ref)
                    geoFire.setLocation(uid, GeoLocation(location.latitude, location.longitude))

                }
            }
        } else {
            Toast.makeText(this, "Turn on location", Toast.LENGTH_LONG).show()
            val intent = Intent(Settings.ACTION_LOCATION_SOURCE_SETTINGS)
            startActivity(intent)
        }
    } else {
        requestPermissions()
    }


}

@SuppressLint("MissingPermission")
private fun requestNewLocationData() {
    var mLocationRequest = LocationRequest()
    mLocationRequest.priority = LocationRequest.PRIORITY_HIGH_ACCURACY
    mLocationRequest.interval = 0
    mLocationRequest.fastestInterval = 0
    mLocationRequest.numUpdates = 1

    mFusedLocationClient = LocationServices.getFusedLocationProviderClient(this)
    mFusedLocationClient!!.requestLocationUpdates(
        mLocationRequest, mLocationCallback,
        Looper.myLooper()
    )
}
```

```
private val mLocationCallback = object : LocationCallback() {
        override fun onLocationResult(locationResult: LocationResult) {
            var mLastLocation: Location = locationResult.lastLocation
            Log.d("Registerashairdresser","$mLastLocation")

        }
    }

    private fun isLocationEnabled(): Boolean {
        var locationManager: LocationManager = getSystemService(Context.LOCATION_SERVICE) as
LocationManager
        return locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER) ||
locationManager.isProviderEnabled(
            LocationManager.NETWORK_PROVIDER
        )
    }

    private fun checkPermissions(): Boolean {
        if (ActivityCompat.checkSelfPermission(
                this,
                Manifest.permission.ACCESS_COARSE_LOCATION
            ) == PackageManager.PERMISSION_GRANTED &&
            ActivityCompat.checkSelfPermission(
                this,
                Manifest.permission.ACCESS_FINE_LOCATION
            ) == PackageManager.PERMISSION_GRANTED
        ) {
            return true
        }
        return false
    }

    private fun requestPermissions() {
        ActivityCompat.requestPermissions(
            this,
            arrayOf(Manifest.permission.ACCESS_COARSE_LOCATION,
Manifest.permission.ACCESS_FINE_LOCATION),
            PERMISSION_ID
        )
    }


    override fun onRequestPermissionsResult(requestCode: Int, permissions: Array<String>,
grantResults: IntArray) {
        if (requestCode == PERMISSION_ID) {
            if ((grantResults.isNotEmpty() && grantResults[0] == PackageManager.PERMISSION_GRANTED))
{
                getLastLocation()
            }
        }
    }
}
```

Nel caso in cui l'utente abbia scelto di registrarsi come esperto, la fase di registrazione non è ancora terminata e si procede con l'avvio di un'activity che richiede ulteriori campi da inserire e che verrà illustrata a breve. In ogni caso l'utente viene già memorizzato nel database, con i campi relativi alla professione attualmente vuoti ma che verranno appunto aggiunti nella schermata successiva. Se ci si registra come esperti, l'applicazione non effettua in questa fase il metodo di geolocalizzazione, operazione eseguita successivamente:

```
else {
        ref = FirebaseDatabase.getInstance().getReference("/Utenti/$uid")

        expert = Expert(
            uid,
            username_registration.text.toString(),
            profileImageUrl,
            "",
            false,
            "",
            ""
        )
        ref.setValue(expert)
            .addOnSuccessListener {
                val intent = Intent (this, RegisterAsExpert::class.java)
                startActivity(intent)
            }
            .addOnFailureListener{
                Log.d("RegisterActivity","Failed to save user to database: ${it.message}")
            }
    }
}
```

4.3.3 RegisterAsExpert Activity

L'activity in questione viene visualizzata solo nel caso in cui l'utente, nella schermata di registrazione, si registri come esperto anziché come normale cliente. Vengono quindi richieste all'utente informazioni ulteriori, come telefono e una breve descrizione. Egli deve obbligatoriamente selezionare una tra le professioni proposte nello spinner come si vede in figura 4.9.

Figura 4.9: Spinner per la selezione della professione

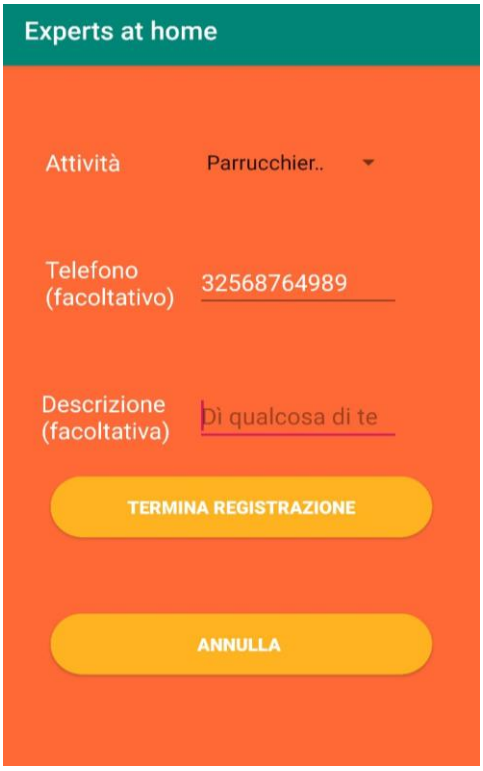

Figura 4.10: Activity RegisterAsExpert con campi compilati

Alla pressione del bottone *termina registrazione* le informazioni relative all'utente che sta effettuando la registrazione vengono correttamente inserite nel database. È anche possibile annullare questa fase di registrazione per tornare alla schermata precedente. Fino ad ora per un esperto non è ancora stata memorizzata la posizione attuale del dispositivo, operazione che viene effettuata solo dopo la corretta registrazione all'interno di questa activity. Anche qui si crea un riferimento al database nella posizione corrispondente all'utente attuale così da poter aggiornare correttamente i

campi con le nuove informazioni inserite. Attraverso l'uso di intent si rimuove dallo stack l'activity corrente per avviare quella successiva:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(activity_expert_register)

    mFusedLocationClient = LocationServices.getFusedLocationProviderClient(this)
    getLastLocation()


    val utenza_array = arrayOf( "Parrucchiere Unisex","Parrucchiere uomo","Parrucchiere donna",
"Estetista", "Sarto/a", "Makeup artist", "Esperto manicure")
    val arrayAdapter = ArrayAdapter (this, android.R.layout.simple_spinner_item, utenza_array)

    // incorporo l'array adapter nello spinner
    spinner_utenza.adapter = arrayAdapter

    var spinnerResult : String? = null
    spinner_utenza.onItemSelectedListener = object :

        AdapterView.OnItemSelectedListener{

        override fun onItemSelected(parent: AdapterView<*>?,
            view: View?,
            position: Int,
            id: Long
        ) {
            spinnerResult = utenza_array[position] //memorizzo la scelta dell'utente in una
variabile

        }

        override fun onNothingSelected(parent: AdapterView<*>?) {

        }
    }

    continue_hairdresser.setOnClickListener {

        val uid = FirebaseAuth.getInstance().uid
        val phone = phone_number_register.text.toString()
        val description = description_register.text.toString()
        if(spinnerResult == null)
            Toast.makeText(this, "Il campo utenza è obbligatorio",Toast.LENGTH_SHORT).show()

        val ref = FirebaseDatabase.getInstance().getReference("/Utenti/$uid") //vado ad aggiornare i
campi
        ref.child("profession").setValue(spinnerResult)
        ref.child("description").setValue(description)
        ref.child("phone_number").setValue(phone)



        val intent = Intent(this,
            ExpertProfileActivity::class.java)
        intent.flags = Intent.FLAG_ACTIVITY_CLEAR_TASK.or(Intent.FLAG_ACTIVITY_NEW_TASK)
        startActivity(intent)


    }

}
```

4.3.4 Login Activity

Un utente già registrato nel database può effettuare un semplice login per accedere alle funzionalità dell'applicazione cliccando sul testo "*Hai già un account*?" presente in fondo alla Register Activity come si vede in figura 4.8.

Per poter effettuare il login, l'applicazione richiede due campi obbligatori: e-mail e password. Se i dati inseriti corrispondo a quelli di un utente nel database l'accesso viene garantito ed ogni qualvolta questo accade, l'applicazione avvia i metodi per l'individuazione della posizione attuale del dispositivo.

Eventualmente l'utente può ritornare alla schermata di registrazione come si vede in figura 4.11.

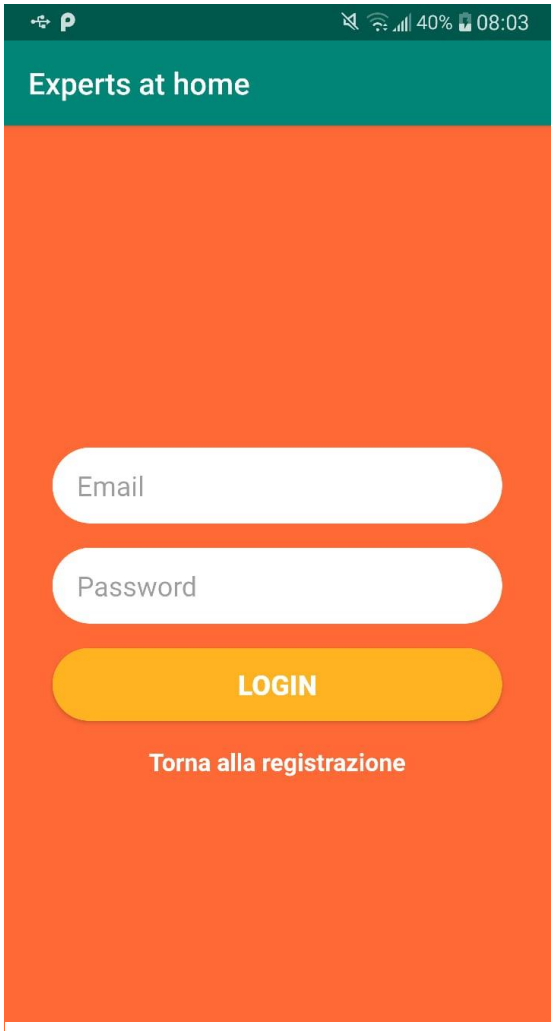

Figura 4.11: Schermata di login

Sfruttando ancora una volta Kotlin, inizialmente si definisce un companion object di classe Expert vuoto. Premendo il bottone *Login,* grazie alle funzioni di autenticazione di Firebase, si riempiono i dati di quest'oggetto con le informazioni relative all'utente corrente, attraverso un listener del riferimento al database che ritorna uno snapshot relativo ai dati dell'utente:

```
class LoginActivity: AppCompatActivity(){

     // variabile globale
    companion object{
        var loginUser: Expert? = null
    }

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    setContentView(R.layout.activity_login)
    login_button.setOnClickListener {
        //mFusedLocationClient = LocationServices.getFusedLocationProviderClient(this)
        //getLastLocation()
        performLogin()

    }

    back_to_register_button_login.setOnClickListener {
        finish()
    }
}
private fun performLogin() {
    val email = email_login.text.toString()
    val password = password_login.text.toString()

    if (email.isEmpty() || password.isEmpty()) {
        Toast.makeText(this, "Inserisci email e password", Toast.LENGTH_SHORT).show()
        return
    }

    Log.d("LoginActivity", "Attempt login with email/password: $email/***")

    //Firebase authentication
    FirebaseAuth.getInstance().signInWithEmailAndPassword(email, password)
        .addOnCompleteListener{
            if (!it.isSuccessful)
                return@addOnCompleteListener

            val uid = FirebaseAuth.getInstance().uid
            val ref = FirebaseDatabase.getInstance().getReference("Utenti/$uid")

            ref.addListenerForSingleValueEvent(object: ValueEventListener {
                override fun onCancelled(p0: DatabaseError) {

                }

                override fun onDataChange(p0: DataSnapshot) {

                    loginUser = p0.getValue(Expert::class.java) //dico che loginUser è quello appena
loggato
                    Log.d("LOGIN", "username: ${loginUser?.username}")
                    Log.d("LOGIN", "profession: ${loginUser?.profession}")
                    Log.d("LOGIN", "isuser : ${loginUser?.isuser}")
```

A questo punto è possibile riconoscere la tipologia di utente che sta effettuando l'accesso, informazione contenuta nel campo *isuser:* se è vera significa che l'utente è un cliente.

Il comportamento dell'applicazione muta di conseguenza, avviando due activity differenti in base alla tipologia d'utente. Se per qualsiasi motivo il login invece non ha successo l'applicazione mostra un *Toast* contenente un messaggio d'errore:

```
if(loginUser?.isuser == false){
                        val intent = Intent(this@LoginActivity, ExpertProfileActivity::class.java)
                        intent.flags =  Intent.FLAG_ACTIVITY_CLEAR_TASK.or(Intent.FLAG_ACTIVITY_NEW_TASK)
                        startActivity(intent)
                    }

                    else {
                        val intent = Intent(this@LoginActivity,  UserHome::class.java)
                        intent.flags = Intent.FLAG_ACTIVITY_CLEAR_TASK.or(Intent.FLAG_ACTIVITY_NEW_TASK)
//per rimuovere dallo stack la precedente activity
                        startActivity(intent)
                    }
                }
            })
        }

        .addOnFailureListener {
            Log.d("main", "Failed to login: ${it.message}")
            Toast.makeText(this, "Failed to login: ${it.message}", Toast.LENGTH_SHORT).show()
        }
}
```

4.3.5 UserHome Activity

Si tratta della schermata principale dell'applicazione, dove diverse funzionalità entrano in gioco. In generale in UserHome l'utente che ha effettuato l'accesso come cliente visualizza tutti gli esperti nelle immediate vicinanze. L'activity include diversi tool per personalizzare la ricerca degli esperti, come una barra di ricerca dove inserire del testo, una barra per impostare il raggio in kilometri entro il quale ricercare utenti, oppure uno spinner per selezionare solo la professione desiderata. In figura 4.12 vengono illustrati tutti questi aspetti.

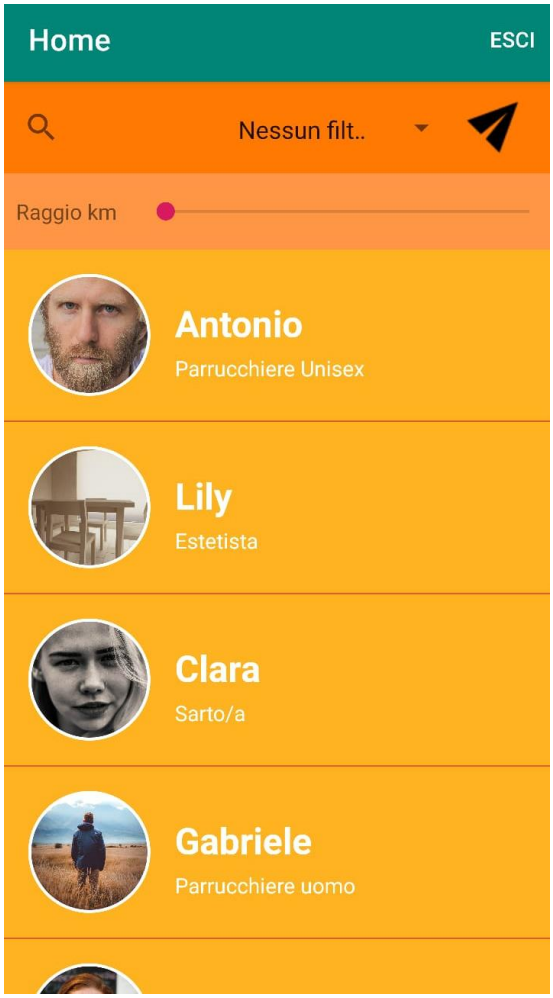

Figura 4.12: Activity UserHome

Gli utenti vengono visualizzati nella schermata all'interno di un *RecyclerView.* Si tratta di un layout che permette appunto di visualizzare una lista dove ogni riga contiene dati, in questo caso ricavati dal database. Attraverso operazioni di filtraggio l'applicazione itera all'interno del

database memorizzando in una struttura dati *HashMap* **solamente** quegli utenti che corrispondono ai criteri di ricerca e che siano esperti. Il layout che compone ogni riga è formato dai dati ricavati dallo snapshot del database: immagine profilo, username e professione. L'applicazione appena avvia l'activity riconosce chi è l'utente corrente, sapendo così quali sono le sue informazioni e la sua posizione. Cliccando su un qualsiasi oggetto del recyclerview, quindi su una qualsiasi riga, l'applicazione ricava i dati dell'utente selezionato per mostrarli correttamente nell'activity successiva, quella che mostra il profilo dell'utente scelto.

Si è resa necessaria la definizione di una classe *UserItem,* che definisce il *ViewHolder*. Viene aggiunto un oggetto di classe *UserItem* per ogni utente presente nella lista (utile quindi per riaggiornare la lista dopo una ricerca):

```
class UserItem(val user: Expert): Item<GroupieViewHolder>(){//è chiamata per
ogni utente nella lista
    override fun bind(viewHolder: GroupieViewHolder, position: Int) {
        viewHolder.itemView.username_new_message.text = user.username //fa
visualizzare gli username del database
        viewHolder.itemView.utenza_text_row.text = user.profession

Picasso.get().load(user.profileImageUrl).into(viewHolder.itemView.image_new_mess
age_row)
    }

    override fun getLayout(): Int {
        return R.layout.user_row_new_message
    }
}
```

Di seguito viene illustrato il metodo principale dell'activity, ovvero quello che permette la visualizzazione degli utenti vicini. Per prima cosa l'adapter (il "contenitore" degli oggetti visualizzati) viene pulito da ciò che mostrava in precedenza, dopodiché si accede alla location dell'utente corrente per verificare quali siano gli esperti che si trovano nelle vicinanze. Questa operazione è possibile attraverso i metodi di Geofire. Nello specifico utilizzando un oggetto di tipo *geoQuery* possiamo interrogare la posizione attuale del dispositivo ed è il metodo *queryAtLocation* che ritorna l'insieme degli utenti vicini. Questo metodo richiede in ingresso una latitudine ed una longitudine, ovvero ciò che Geofire ha già memorizzato nel database, ed un raggio in km entro il quale effettuare la ricerca, avente come centro la posizione attuale. Ogni qualvolta *queryAtLocation* trova un risultato, viene chiamata la funzione onKeyEntered, all'interno della quale l'applicazione ricava la chiave dell'utente appena trovato e memorizza

tutte le sue informazioni all'interno della struttura dati creata inizialmente. Per ogni chiave viene aggiunto all'adapter l'oggetto corrispondente. Questa operazione viene effettuata per ognuno degli esperti ricavati. Infine, si aggiorna il layout *recyclerview* con il contenuto dell'adapter. Il frammento di codice relativo alla funzione nel suo intero è illustrato nella pagina seguente.

```
private fun findNearUsers(){
    map.clear()
    adapter.clear()

    val uid = FirebaseAuth.getInstance().uid
    val ref = FirebaseDatabase.getInstance().getReference("/locations")

    val geoFire = GeoFire(ref)

    geoFire.getLocation(uid, object: LocationCallback(), com.firebase.geofire.LocationCallback {
        override fun onCancelled(databaseError: DatabaseError?) {

        }

        override fun onLocationResult(key: String?, location: GeoLocation?) {
            if (location != null) {

                val geoQuery = geoFire.queryAtLocation(
                    GeoLocation(location.latitude, location.longitude),
                    seekBar2.progress.toDouble()
                )
                geoQuery.removeAllListeners()
                geoQuery.addGeoQueryEventListener(object : GeoQueryEventListener {
                    override fun onKeyMoved(key: String?, location: GeoLocation?) {

                    }

                    override fun onKeyExited(key: String?) {
                    }
                    override fun onGeoQueryError(error: DatabaseError?) {
                    }
                    override fun onKeyEntered(key: String?, location: GeoLocation?) {

                        val ref2 = FirebaseDatabase.getInstance().getReference("/Utenti/$key")
                        ref2.addListenerForSingleValueEvent(object : ValueEventListener {

                            override fun onCancelled(p0: DatabaseError) {

                            }

                            override fun onDataChange(p0: DataSnapshot) {


                                val user = p0.getValue(Expert::class.java) ?: return

                                map[p0.key!!] = user

                                adapter.clear()
                                if(spinner_filtro.selectedItem.toString() == "Nessun filtro")
                                    map.values.forEach {
                                        if(it.isuser == false){
                                            adapter.add(UserItem(it))

                                        }
                                    }
                                else
                                {
                                    map.values.forEach {
                                        if(it.isuser == false && it.profession ==
spinner_filtro.selectedItem.toString()){
                                            adapter.add(UserItem(it))
                                        }
                                    }
                                }
                                recyclerview_newmessage.adapter = adapter
                            }
                        })
                    }

                    override fun onGeoQueryReady() {
                        Log.d("GEOQUERYREADY","$key")
                    }
                })
            }
        }
    })
}
```

Come già accennato in precedenza, particolare enfasi è stata posta sui meccanismi di ricerca degli utenti. La barra in alto contiene un listener che serve a definire il raggio in km entro il quale si vogliono cercare le risposte (valore che viene utilizzato nella funzione findNearUsers illustrata nella pagina precedente):

```
val filtro_array = arrayOf( "Nessun filtro","Parrucchiere Unisex","Parrucchiere uomo","Parrucchiere
donna", "Estetista", "Sarto/a", "Makeup artist", "Esperto manicure")
val arrayAdapter = ArrayAdapter (this, android.R.layout.simple_spinner_item, filtro_array)
spinner_filtro.adapter = arrayAdapter
var spinnerResult : String? = null

seekBar2.setOnSeekBarChangeListener(object : SeekBar.OnSeekBarChangeListener {

    override fun onProgressChanged(seekBar: SeekBar, i: Int, b: Boolean) {
        // Display the current progress of SeekBar

    }

    override fun onStartTrackingTouch(seekBar: SeekBar) {
        // Do something
    }

    override fun onStopTrackingTouch(seekBar2: SeekBar) {
        // Do something
        Toast.makeText(this@UserHome,
            "Raggio di " + seekBar2.progress + "km",
            Toast.LENGTH_SHORT).show()
        findNearUsers()
    }
})
```

La barra di ricerca è utile per filtrare gli esperti in base al loro username:

```
search_bar.setOnQueryTextListener(object: SearchView.OnQueryTextListener,
    android.widget.SearchView.OnQueryTextListener {
    override fun onQueryTextSubmit(query: String?): Boolean {
        return false
    }

    override fun onQueryTextChange(newText: String?): Boolean {

        if (!newText.isNullOrBlank()) { //se cerco qualcosa chiama search
            search(newText)
        }

        else{
            Log.d("SEARCH","barra vuota")
            findNearUsers()
        }
        return true
    }
})
```

mentre lo spinner (mostrato in figura 4.13) include tutte le varie possibili attività offerte dagli esperti:

```
spinner_filtro.onItemSelectedListener = object :

    AdapterView.OnItemSelectedListener{

    override fun onItemSelected(parent: AdapterView<*>?,
                                view: View?,
                                position: Int,
                                id: Long
    ) {
        spinnerResult = filtro_array[position]//memorizzo la scelta dell'utente in una variabile
        if(spinnerResult == filtro_array[0])
            findNearUsers()
        else
            search(spinnerResult!!)
    }

    override fun onNothingSelected(parent: AdapterView<*>?) {
        findNearUsers()

    }
}
```

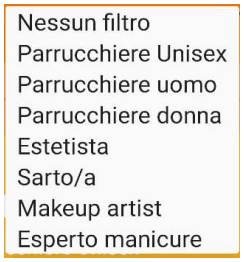

Figura 4.13: spinner dell'activity UserHome per la ricerca in base alla professione

Sia lo spinner che la funzione di ricerca chiamano il metodo *search* solo se il loro valore non è vuoto, ovvero se l'utente ha effettivamente scritto qualcosa all'interno della barra di ricerca o se ha selezionato una professione dallo spinner. Questo metodo prende in ingresso una stringa, quindi o il testo inserito dall'utente o il contenuto selezionato dallo spinner (che viene

opportunatamente convertito in stringa):

```
private fun search(newText: String){ // fa visualizzare solo gli username che contengono ciò che è
scritto nella barra di ricerca o nello spinner

    val ref = FirebaseDatabase.getInstance().getReference("/Utenti")
    ref.addListenerForSingleValueEvent(object : ValueEventListener{
        override fun onCancelled(p0: DatabaseError) {
        }

        override fun onDataChange(p0: DataSnapshot) {
            val adapter = GroupAdapter<GroupieViewHolder>()

            p0.children.forEach{

                val user = it.getValue(Expert::class.java)
                if (user != null)
                    if((user.username.contains(newText) || user.profession == newText) &&
user.isuser==false)
                      adapter.add(UserItem(user))

            }

            adapter.setOnItemClickListener{item,view ->


                val userItem = item as UserItem //per castare item nell'object corretto

                val intent = Intent(view.context, ProfileViewActivity::class.java )
                intent.putExtra(USER_KEY,userItem.user)//passa tutte le informazioni di user, non
solo l'username
                startActivity(intent)

                //finish()
            }

            recyclerview_newmessage.adapter = adapter
        }
    })
}
```

Ogni qualvolta una di queste azioni di filtraggio viene eseguita, la schermata aggiorna istantaneamente il suo contenuto attraverso un update dell’adapter.

Le figure 4.14, 4.15 e 4.16 illustrano esempi di ricerca in riferimento alla schermata visualizzata in figura 4.12.

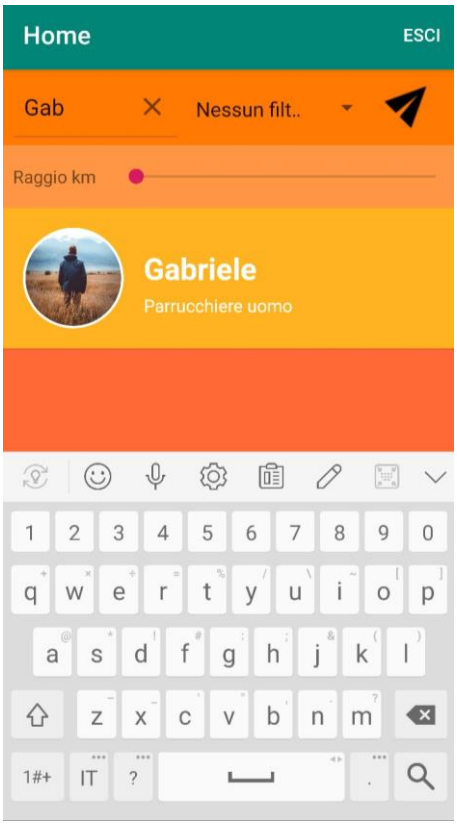

Figura 4.14: Nella barra di ricerca compare la stringa "Gab". L'applicazione filtra gli utenti per Far visualizzare solo coloro che nel proprio username contengono quella stringa.

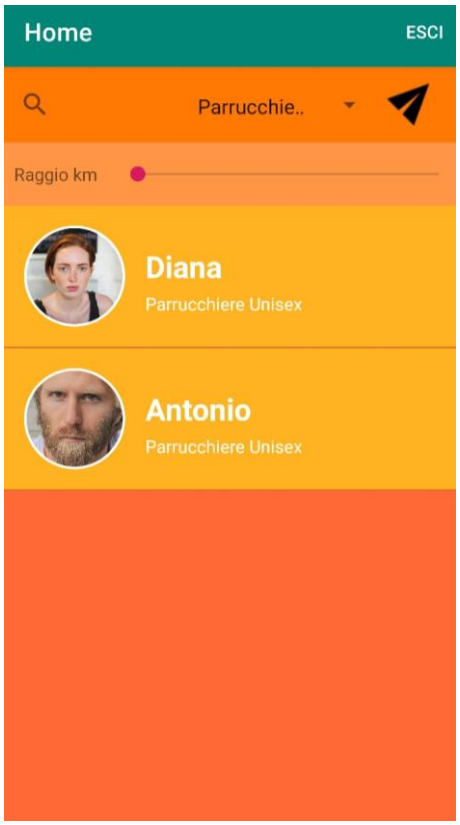

Figura 4.15: Vengono visualizzati solo gli utenti che hanno come professione quella selezionata nello spinner, in questo caso "Parrucchiere unisex"

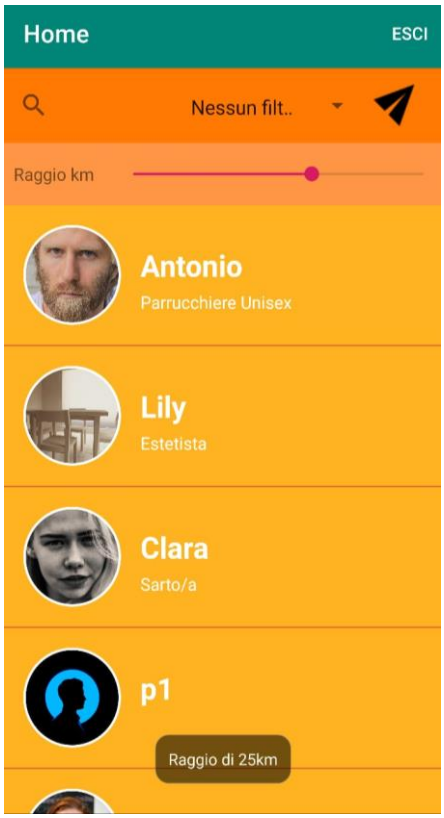

Figura 4.16: L'utente sposta la barra che indica il raggio in km entro il quale si vogliono cercare gli utenti. Un messaggio compare sullo schermo indicando il numero di kilometri. Vediamo che compare l'utente p1 che inizialmente in figura 4.12 non compariva poiché si trovava a più di 1km di distanza dalla posizione attuale.

4.3.6 ProfileView Activity

L'activity in questione viene lanciata dopo aver selezionato un utente nella UserHome Activity, dove ogni riga conteneva un listener per avviare questa nuova activity:

```
adapter.setOnItemClickListener{item,view ->
    val userItem = item as UserItem //per castare item nell'object corretto

    val intent = Intent(view.context, ProfileViewActivity::class.java )
    intent.putExtra(USER_KEY,userItem.user)//passa tutte le informazioni di user, non solo
l'username
    startActivity(intent)


}
```

Attraverso l'uso di un intent è stato possibile quindi ottenere tutte le informazioni relative all'utente selezionato. Più specificatamente in questo caso si utilizza l'oggetto *extra* che attraverso il metodo *putExtra* permette di passare oggetti da un'activity ad un'altra, così che le informazioni dell'utente selezionato possano essere visualizzate nella schermata di visualizzazione del profilo di un esperto, come illustrato in figura 4.17 con dati d'esempio.

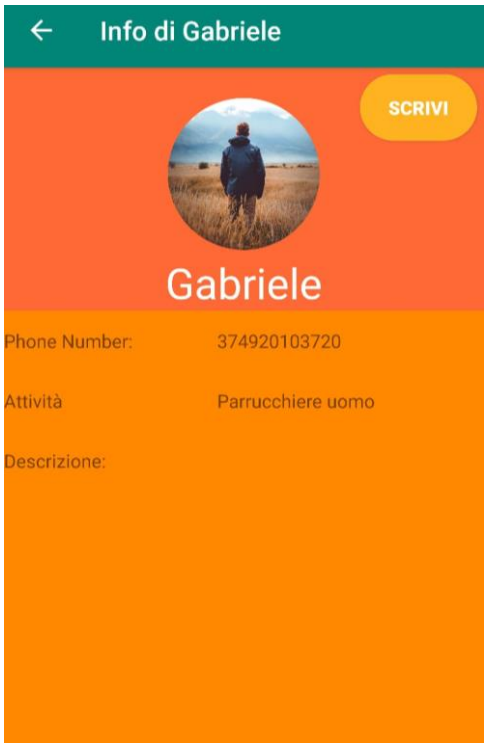

Figura 4.17: Visualizzazione dell'utente Gabriele con le relative informazioni.

Attraverso uno snapshot del database l'applicazione memorizza i dati relativi all'utente di cui si vogliono visualizzare le informazioni e le inserisce nei rispettivi campi del layout. Anche in questo caso si utilizza la libreria Picasso citata nel capitolo 2 per la corretta visualizzazione delle immagini.

```
override fun onDataChange(p0: DataSnapshot) {
    val user = p0.getValue(Expert::class.java)
    Log.d("profile", "current user: ${user?.profession}")

    profile_username_view.text = user?.username
    description_profile_view.text = user?.description
    phone_number_profile_view.text = user?.phone_number
    utenza_profile_view.text = user?.profession


    val uri = user?.profileImageUrl
    val targetImage = image_profile_view
    Picasso.get().load(uri).into(targetImage)
```

In questa schermata il cliente può contattare l'utente di cui sta visualizzando il profilo attraverso il bottone *scrivi* situato in alto a destra. Le activity riguardanti la fase di messaggistica verranno trattate a breve nei prossimi paragrafi.

4.3.7 ExpertProfile Activity

La schermata in figura 4.18 rappresenta l'activity che viene visualizzata subito dopo la fase di registrazione o di login da parte di un esperto anziché di un cliente.

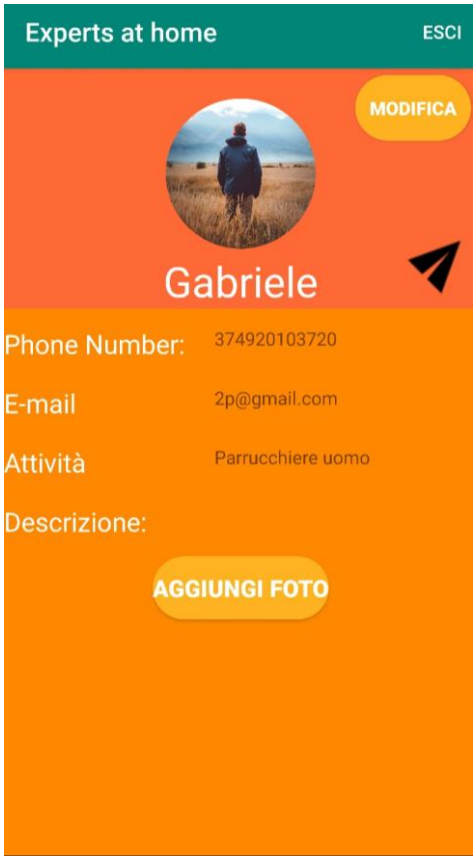

Figura 4.18: ExpertProfile Activity di un utente d'esemipo

Le informazioni del database relative all'utente vengono ricavate in maniera analoga a come è stato fatto nelle precedenti activity, con una leggera differenza: i dati riguardano l'utente corrente e non un utente diverso dall'utilizzatore. La schermata in questione, a differenza di ProfileView Activity, contiene nel suo layout un bottone *modifica* in alto a destra che servirà a mostrare l'activity relativa alla modifica del profilo, un bottone *aggiungi foto* utile nel caso l'utente voglia caricare immagini sul suo profilo ed infine il bottone a forma di aeroplano di carta per visualizzare gli ultimi messaggi scambiati con altri utenti.

4.3.8 Edit Activity

Figura 4.19: Edit Activity appena viene avviata

Si tratta della schermata visualizzata dopo che l'utente ha deciso di modificare il proprio profilo in ExpertProfile Activity. Qui è possibile modificare tutte le informazioni relative al proprio account, come la foto profilo o la propria attività. Alla pressione del tasto *salva* si ritorna all'activity precedente **solo dopo** che i nuovi dati vengono aggiornati nei rispettivi campi del database. L'attesa è necessaria per la corretta visualizzazione delle modifiche apportate nella schermata del profilo:

```
private fun performSave(){

    if(selectedPhotoUri == null)
        return

    val filename = UUID.randomUUID().toString()
    val ref = FirebaseStorage.getInstance().getReference("/images/$filename")

    ref.putFile(selectedPhotoUri!!)
        .addOnSuccessListener {
            Log.d("Edit", "Successfully uploaded image: ${it.metadata?.path}")

            ref.downloadUrl.addOnSuccessListener {
                Log.d("Edit", "File Location $it")

               editInfo(it.toString()) //passo l'url alla funzione
            }
        }


}


private fun editInfo(profileImageUrl: String){


    val uid = FirebaseAuth.getInstance().uid
    val phone = phone_edit.text.toString()
    val description = description_edit.text.toString()
    val username = username_edit.text.toString()


    val ref = FirebaseDatabase.getInstance().getReference("/Utenti/$uid")
    ref.child("utenza").setValue(spinnerResult)
    ref.child("description").setValue(description)
    ref.child("phone_number").setValue(phone)
    ref.child("username").setValue(username)
    ref.child("profileImageUrl").setValue(profileImageUrl)


    ref.addListenerForSingleValueEvent(object : ValueEventListener {
        override fun onCancelled(p0: DatabaseError) {

        }

        override fun onDataChange(p0: DataSnapshot) {  // Dopo aver cliccato "Salva" aspetto che il
database venga aggiornato. Quindi chiudo l'acitvity solo dopo il cambiamento.
                                                       // I dati non verrebbero visualizzati
altrimenti dato che firebase ci sta qualche secondo ad effettuare modifiche.
            finish()

        }
    })


}
```

Come nella fase di registrazione si controlla che l'immagine profilo venga obbligatoriamente selezionata, altrimenti compare un messaggio d'errore. Premendo il tasto di navigazione *indietro* le modifiche non vengono apportate e si visualizza il profilo com'era prima di avviare l'activity *EditProfile*. Attraverso il tasto *Aggiorna posizione* l'utente può decidere se aggiornare la posizione attuale del suo dispositivo. Questa funzione nel caso di un esperto non viene

effettuata ogni volta che effettua l'accesso all'applicazione (cosa che avviene invece per un normale cliente) dal momento che si suppone che chi offre un'attività lavorativa la svolga sempre nello stesso luogo e non voglia aggiornare frequentemente la sua location di lavoro. Per questo motivo le funzioni di geolocalizzazione illustrate precedentemente, nel caso di update della posizione del professionista, avvengono manualmente attraverso un listener, il quale chiama il metodo *getLastLocation* che come già visto aggiorna in real-time la chiave di Geofire memorizzata all'interno del database:

```
update_location_button_edit.setOnClickListener {
    getLastLocation()
    Toast.makeText(this, "Posizione aggiornata",Toast.LENGTH_SHORT).show()
}
```

L'utente ha infine un'ultima opzione in Edit Activity, ovvero quella di eliminare il proprio account dall'applicazione e di conseguenza dal database. Quest'operazione avviene alla pressione del testo *cancella il tuo account:*

```
delete_edit.setOnClickListener {
        val uid = FirebaseAuth.getInstance().uid

FirebaseDatabase.getInstance().getReference("/Utenti/$uid").removeValue().addOnSuccessListener {
            Log.d("edit activity", "user deleted : $uid")
        }

        intent = Intent(this, RegisterActivity::class.java)
        startActivity(intent)
    }
}
```

Dopo aver eliminato il proprio account l'applicazione avvia la *Register Activity*.

4.3.9 Chatlog Activity

Le ultime due activity non ancora analizzate sono quelle che riguardano la sezione di messaggistica dell'applicazione. La prima tra queste che verrà analizzata è la Chatlog Activity, mostrata nelle figure 4.20 e 4.21.

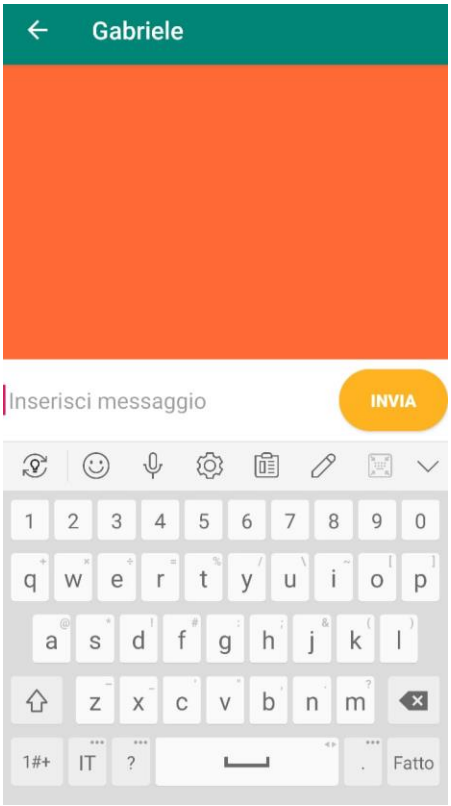

Figura 4.20: Chatlog Activity in cui l'utente sta scrivendo un messaggio ad un altro utente di nome Gabriele.

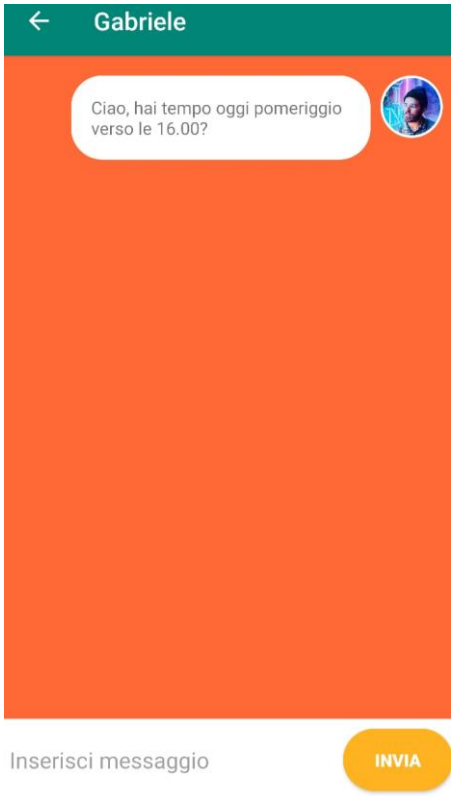

Figura 4.21: Chatlog Activity dopo che è stato inviato il primo messaggio all'utente Gabriele.

L'activity in questione viene avviata alla pressione del bottone *scrivi* dell'activity ProfileView illustrata in figura 4.17.

Si presenta con la classica casella di testo dove inserire il messaggio da inviare e il bottone *invia* per inviare effettivamente il testo all'utente selezionato.

L'invio del messaggio dà inizio a diversi eventi: la visualizzazione sullo schermo del messaggio inviato con rispettiva immagine profilo dell'utente mittente (è stato necessario creare due layout differenti per la visualizzazione corretta del messaggio, nel caso in cui il messaggio

debba essere visualizzato dal mittente o dal destinatario). Inoltre, il messaggio appena inviato e le sue informazioni vengono immagazzinati nel database:

```
private fun performSendMessage(){
    //Invia il messaggio a firebase
    val text = testo_chatlog.text.toString() //Ciò che viene scritto dall'utente
    val fromId = FirebaseAuth.getInstance().uid //l'id dell'utente che ha effettuato l'accesso

    val user = intent.getParcelableExtra<Expert>(UserHome.USER_KEY) //Ridefinisco user per potervi
accedere
    val toId = user.uid //L'id che rappresenta l'utente a cui sto scrivendo

    if (fromId == null) return


    val reference = FirebaseDatabase.getInstance().getReference("/user-
messages/$fromId/$toId").push() //push genera un nuovo nodo nel database

    val toReference = FirebaseDatabase.getInstance().getReference("/user-
messages/$toId/$fromId").push()

    val chatMessage = ChatMessage(
        reference.key!!, // !! serve per risolvere il type mismatch, col controllo, //sopra sul from
Id sono più safe e non devo inserire !! anche lì
        fromId,
        toId,
        text,
        System.currentTimeMillis() / 1000 // il timestamp mi dà il risultato in millisecondi,
dividendo per 1000 ottengo i secondi
    )


    reference.setValue(chatMessage)
        .addOnSuccessListener {
            Log.d("ChatlogActivity", "Chat message saved: ${reference.key}")
            testo_chatlog.text.clear()
            recyclerview_chatlog.scrollToPosition(adapter.itemCount -1) //dopo aver inviato un
messaggio posiziono la schermata sull'ultimo messaggio attraverso l'adapter che è l'insieme delle
righe dei msg
        }

    toReference.setValue(chatMessage)

    val latestMessageRef = FirebaseDatabase.getInstance().getReference("/latest-
messages/$fromId/$toId")
    val latestMessageToRef = FirebaseDatabase.getInstance().getReference("/latest-
messages/$toId/$fromId")
    latestMessageRef.setValue(chatMessage)
    latestMessageToRef.setValue(chatMessage)
}
```

Appena avviata, l'activity effettua il metodo *listenForMessages* per caricare sullo schermo tutti i messaggi già scambiati per i due utenti, ricavando l'oggetto di classe *ChatMessage* dal database, che include mittente, destinatario, timestamp e testo del messaggio:

```
private fun listenForMessages(){
    val fromId = FirebaseAuth.getInstance().uid
    val toId = toUser?.uid
    val ref = FirebaseDatabase.getInstance().getReference("/user-messages/$fromId/$toId")

    ref.addChildEventListener(object: ChildEventListener{

        override fun onChildAdded(p0: DataSnapshot, p1: String?) {
            val chatMessage = p0.getValue(ChatMessage::class.java)

            if (chatMessage != null){
                Log.d("ChatlogActivity", chatMessage.text )

                if(chatMessage.fromId == FirebaseAuth.getInstance().uid) {//per capire da quale lato
deve comparire il messaggio in base agli id
                    val currentUser = LoginActivity.loginUser
                    Log.d("CHATLOG","${currentUser?.username}")
                    adapter.add(
                        ChatRightItem(
                            chatMessage.text,
                            currentUser!!
                        )
                    )
                }

                else{
                    adapter.add(
                        ChatLeftItem(
                            chatMessage.text,
                            toUser!!
                        )
                    )
                }
            }
            recyclerview_chatlog.scrollToPosition(adapter.itemCount -1) //per far partire la
schermata dal basso, si visualizza l'ultimo msg
        }
```

In figura 4.22 viene mostrato il messaggio ricevuto dal destinatario, che a sua volta diventa il mittente: l'applicazione sa sempre così da quale lato visualizzare gli item.

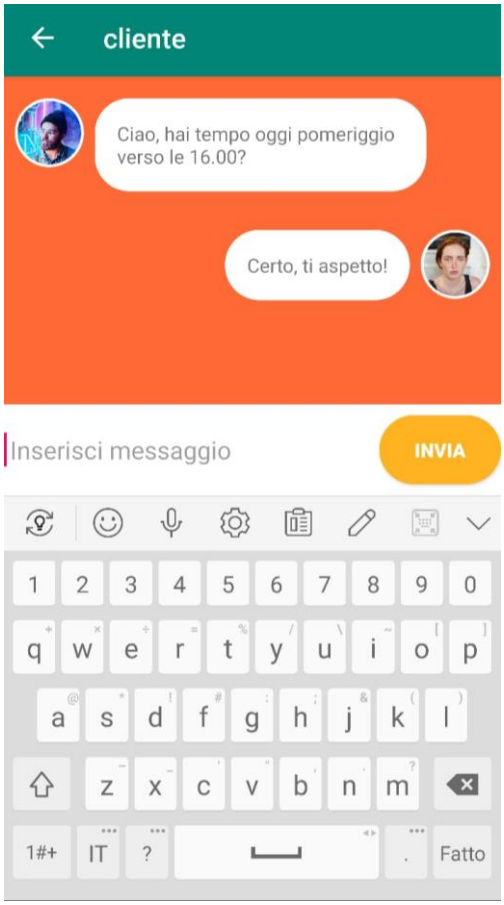

Figura 4.22: Chatlog dal lato dell'esperto, che risponde al messaggio inviato precedentemente dal cliente

4.3.10 LatestMessages Activity

L'ultima activity da analizzare è quella riguardante la visualizzazione della lista dei messaggi scambiati coi vari utenti, come mostrato in figura 4.23.

Figura 4.23: LatestMessages activity

La visualizzazione della schermata segue un funzionamento analogo all'activity *UserHome*: si usa un oggetto *RecyclerView* per impostare una lista dove ogni riga in questo caso contiene diversi elementi, ovvero immagine profilo, username e testo del messaggio.

All'avvio dell'activity viene invocata la funzione *listenForLatestMessages* che ricava dal database le informazioni necessarie per poter costruire la lista. Il concetto è il seguente: per ogni persona con cui l'utente corrente ha chattato, ne viene visualizzato l'ultimo messaggio scambiato. Ogni oggetto trovato viene memorizzato in un hashmap che verrà utilizzata dall'adapter per visualizzare correttamente ogni riga:

```
private fun listenForLatestMessages(){
    val fromId = FirebaseAuth.getInstance().uid
    val ref = FirebaseDatabase.getInstance().getReference("/latest-messages/$fromId")
    Log.d("LatestMessages","$fromId")
    ref.addChildEventListener(object: ChildEventListener{

        override fun onChildAdded(p0: DataSnapshot, p1: String?) {
            val chatMessage = p0.getValue(ChatMessage::class.java) ?: return //converto lo snapshot
in un chatmessage

            latestMessagesMap[p0.key!!] = chatMessage //aggiungo il chatmessage all'hashmap. key
rappresenta gli utenti con cui si messaggia
            refreshRecyclerViewMessages() //per pulire e ricaricare la schermata coi nuovi ultimi
msg

        }

        override fun onChildChanged(p0: DataSnapshot, p1: String?) { // si attiva quando qualcuno mi
scrive e voglio visualizzare l'ultimo messaggio sull'activity
            val chatMessage = p0.getValue(ChatMessage::class.java) ?: return

            latestMessagesMap[p0.key!!] = chatMessage
            refreshRecyclerViewMessages()

        }
```

Ovviamente per ogni riga dell'adapter si imposta un listener alla pressione che fa avviare la *Chatlog Activity* con l'utente selezionato:

```
adapter.setOnItemClickListener{item, view ->
    Log.d("LatestMessageActivity", "123")
    val intent = Intent(this, ChatLogActivity::class.java)

    val row = item as LatestMessageRow //casto l'item in latestmessagerow

    intent.putExtra(UserHome.USER_KEY, row.chatPartnerUser )
    startActivity(intent)
}
```

Infine, un elemento presente in quasi tutte le activity è il bottone *esci* posizionato in alto sulla barra Android. Come si può intuire, permette all'utente corrente di effettuare il logout dall'applicazione, notificando Firebase e avviando nuovamente la Register Activity:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
        when (item.itemId){

            R.id.menu_sign_out -> {
                FirebaseAuth.getInstance().signOut()
                val intent = Intent(this, RegisterActivity::class.java)
                intent.flags = Intent.FLAG_ACTIVITY_CLEAR_TASK.or(Intent.FLAG_ACTIVITY_NEW_TASK)
                startActivity(intent)

            }
        }
        return super.onOptionsItemSelected(item)
    }
```

CONCLUSIONI E SVILUPPI FUTURI

Lo sviluppo e la realizzazione di "*Experts At Home*" rappresenta un percorso di autoapprendimento che ha portato alla creazione di un'applicazione mobile nuova, che metta in contatto professionisti e clienti attraverso una serie di interfacce intuitive. Come spiegato nella fase introduttiva della tesi, lo scopo dell'applicazione è quello di risolvere un problema reale, ovvero la mancanza di mezzi veloci e comodi per raggiungere i clienti se si è una persona che offre un qualsiasi tipo di attività lavorativa all'interno della propria abitazione. L'applicazione fa in modo di dare visibilità a questi professionisti fornendo ai clienti efficaci strumenti di ricerca per soddisfare un'esigenza in maniera veloce. La possibilità di aggiungere immagini da parte di un professionista aiuta ad aumentare la sua visibilità rispetto ad altri utenti concorrenti nelle vicinanze.

Ponendosi nei panni di un eventuale cliente, l'applicazione si può rivelare estremamente utile in quei casi in cui si ha bisogno di trovare una soluzione vicina ed in brevi tempi. Possiamo dire che l'architettura dell'applicazione si presta molto bene ad un utilizzo "al volo". Questo è dovuto anche grazie al rapido sistema di messaggistica, che riflette una struttura comune a tante altre popolari applicazioni di chat, così da avere subito un'aria familiare all'utente. La maggior parte delle funzionalità dell'applicazione sono state implementate rimanendo nell'ottica della velocità: gli utenti dell'applicazione devono trovare una risposta ad un loro bisogno nel minor tempo possibile. Per questo motivo, per esempio, gli strumenti di ricerca sono stati collocati tutti su un'unica schermata ed è subito intuibile il loro funzionamento.

In sostanza, gli obiettivi prefissati nella fase di progettazione dell'applicazione sono stati raggiunti. Nonostante questo, l'applicazione presenta sezioni che possono essere migliorate o implementate diversamente.

Una delle prime migliorie che può essere implementata è uno strumento di recensione degli utenti. L'applicazione attraverso la geolocalizzazione potrebbe riconoscere se un cliente sia

stato nell'abitazione di chi offre un'attività e quindi chiedere al cliente di inserire una recensione relativa all'attività in questione. Inoltre, al momento l'applicazione include un elenco di attività lavorative impostate a priori e la possibilità di selezionarne solamente una: si potrebbe dare l'opportunità agli utenti di scegliere più professioni dall'elenco ed eventualmente di aggiungerne di nuove. *Experts At Home* presenta una struttura ed un format che si possono adattare a tante situazioni differenti: l'applicazione può essere ancora allargata sotto molti aspetti.

BIBLIOGRAFIA

1. Android Studio Documentation: https://developer.android.com/studio

2. Articolo su elementi Android: https://www.html.it/pag/48525/gli-elementi-e-il-funzionamento-di-base-di-unapplicazione/

3. Firebase Documentation: https://firebase.google.com/docs/

4. Articolo su confronto tra SQL e NoSQL: https://www.html.it/articoli/sql-e-nosql-a-documenti-il-confronto/

5. Articolo su GeoFire: https://firebase.googleblog.com/2014/06/geofire-20.html

6. Picasso Documentation: https://github.com/square/picasso

7. Kotlin Documentation: https://developer.android.com/kotlin , https://kotlinlang.org/

8. "*Utilizzo di Kotlin per sviluppo di applicazioni mobili*": https://github.com/alewin/Tesi-Triennale/blob/master/Latex/Capitoli/Kotlin.tex