Programmazione Mobile

1097606 Morelli Valerio,
1098788 Sbattella Mattia

Luglio 2023

Contents

1 Introduzione

Spotted! è un social che ci aiuta a trovare persone con cui siamo entrati in contatto e di cui non ricordiamo il nome o che abbiamo avvistato solo brevemente una sera e di cui vogliamo conoscere l'identità. Scrivendo una breve descrizione della persona alla quale siamo interessati, specificando il luogo dove l'abbiamo vista e pubblicando il post, possiamo attendere che gli altri utenti iscritti alla nostra applicazione, o magari la persona stessa, rispondano con il nome della persona spottata. L'applicazione si presta benissimo anche ad implementazioni future quali la possibilità di aggiungere spot per le feste, o spot per annunci per la ricerca di coinquilini.

1.1 Definizione degli obiettivi

L'obiettivo principale dell'app *Spotted!* è quello di trovare persone e fare nuove conoscenze. Come qualsiasi altra app social, dopo aver creato un account o semplicemente effettuando l'accesso, l'utente potrà visualizzare in maniera completamente gratuita i post fatti da tutti gli altri utenti registrati. Se ci si riconosce in uno di questi, potrà commentare il post specifico e seguirlo per rimanere sempre aggiornato. Inoltre ogni post, così come ogni utente, ha dei tag. I tag forniscono delle informazioni aggiuntive sulle persone spottate (come ad esempio "Alto" o "Riccio").

Mentre si scrolla la home, si potrà notare nei post la percentuale di corrispondenza tra i nostri tag e quelli del post in questione. Se scrollare non fosse comodo ci sarà la mappa, con tutti gli spot fatti e localizzati. Se l'utente avesse la necessità di concentrarti su una singola località potrà farlo.

L'utente potrà creare un nuovo spot, per cercare la persona desiderata. Sappiamo quanto la privacy possa essere importante, per questo abbiamo pensato di dare la possibilità di poter effettuare post in anonimo. Se si visita un profilo diverso dal nostro, gli spot effettuati in anonimo da questa persona non compariranno nella sezione apposita, per tutelarli al massimo. Nella sezione 'profilo' l'utente avrà anche la possibilità di navigare tra i suoi post e quelli seguiti, avrà la possibilità di cambiare la propria immagine di profilo ed i propri tag.

Infine, nella sezione delle impostazioni, l'utente potrà modificarle a suo piacimento per avere un'esperienza personalizzata al suo stile.

1.2 Sintesi dell'approcio

Per progettare l'applicazione, abbiamo seguito un approccio prevalentemente di tipo *top-down* per i mockup delle schermate, mentre un'approccio *ibrido* per l'architettura delle classi.

In effetti, al fine di ricercare uno stile coerente che legasse insieme tutte le schermate visibili all'utente, come prima cosa abbiamo fissato una palette di colori e delle misure geometriche, come i margini laterali e il raggio dei bordi arrotondati da utilizzare nelle pagine. Inoltre abbiamo fin da subito distinto l'intera applicazioni in due gruppi di schermate, quelle responsabili dell'autenticazione e registrazione per utenti non già autenticati e tutte le altre di interazione con il *social* per gli utenti autenticati. Successivamente abbiamo, per ciascuna delle due sezioni, realizzato una mappa di navigazione su carta e poi replicata attraverso mokup utilizzando lo strumento online *Wondershare Mockitt*.

Termina questa fase di progettazione la determinazione di componenti grafici personalizzati, come il pulsante o l'elemento che rappresenta una tag. La loro modularizzazione risulta assai efficace, poiché ci permette di separare le responsabilità e snellire il codice delle pagine.

Per quanto riguarda invece la progettazione dell'architettura interna, abbiamo iniziato trovando i componenti fondamentali e organizzandoli in directory apposite. Inoltre, poiché era per noi chiara fin dall'inizio la decisione di implementare il pattern architetturale *Model-View-ViewModel*, possiamo dire che metà di questo lavoro era già compiuto. Successivamente, con un approccio *bottom-up* abbiamo raffinato e unito le varie parti; abbiamo ad esempio definito le aggregazioni esistenti tra le classi del model.

Elementi cruciali che sono emersi fin da subito sono i *Managers*, classi specializzate per eseguire azioni specifiche di più basso livello interfacciandosi con servizi remoti o API di sistema, accessibili da qualsiasi parte del programma come previsto dal pattern *Singleton*.

2 Sviluppo Android

2.1 Discussione dei requisiti

Raccogliamo nel seguito i requisiti rappresentanti il punto di partenza delle scelte di progettazione riportate nelle sezioni successive.

2.1.1 Requisiti funzionali

- **Login:** l'utente deve essere in grado di accedere al proprio profilo, attraverso l'e-mail e la password immesse durante la fase di registrazione.
- **SignUp:** l'utente deve essere in grado di registrarsi, attraverso la compilazione di campi obbligatori quali nome, cognome, email e password, e di campi facoltativi quali numero di telefono, genere, nickname instagram.

Una volta effettuato l'accesso, l'utente potrà inoltre:

- **Visualizza posts:** nella home l'utente registrato avrà la possibilità di visualizzare tutti i post, ordinandoli per data o per compatibilità o filtrandoli per luogo dello spot.
- **Gestisci tag:** gli utenti saranno in grado di visualizzare la percentuale di corrispondenza tra i tag dei post presenti nella home e i propri tag, in modo da individuare immediatamente quelli con una maggiore rilevanza.
- **Visualizza post:** l'utente potrà visualizzare tutte le informazioni ad esso associate come la descrizione, il luogo dove è avvenuto l'avvistamento, i commenti di altri utenti o dell'autore, il genere e i tag della persona spottata.
- **Crea post:** l'utente avrà la possibilità di creare un nuovo post con le informazioni essenziali quali la descrizione, il luogo dove è avvenuto l'avvistamento, il genere e i tag della persona spottata. L'utente avrà anche la possibiltà di pubblicare il post in forma anonima, impedendo quindi agli altri utenti di risalire all'autore del post. La form compilata deve essere validata prima della memorizzazione del post, in modo da garantire dei limiti di lunghezza della descrizione e un minimo di tag selezionati.
- **Conferma spot:** L'utente che ha creato il post avrà la possibilità di confermare di aver trovato la persona che stava cercando.
- **Commenta post:** L'utente potrà commentare i post che desidera e cominciare una conversazione con altri utenti per riuscire a trovare la persona che si sta cercando.
- **Segui post:** l'utente potrà seguire un post, per averlo sempre nel proprio profilo e per controllare aggiornamenti in tempo reale.
- **Visualizza mappa:** l'utente sarà in grado di visualizzare tutti gli spot dalla mappa. Sarà inoltre possibile creare spot,prendendo come località le coordinate selezionate.
- **Visualizza profilo:** l'utente sarà in grado di visualizzare il proprio profilo, compresi i proprio post e i post seguiti. Inoltre nel profilo sarà possibile modificare i propri tag o cambiare l'immagine di profilo direttamente dalla galleria.

  L'utente sarà anche in grado di visualizzare il profilo degli altri utenti. Sarà inoltre possibile vedere i post realizzati e quelli seguiti del profilo selezionato. Inoltre nel profilo sarà possibile visualizzare, ma non modificare, i tag e l'immagine di profilo. Se l'utente selezionato durante la fase di registrazione ha immesso il numero di telefono, sarà possibile chiamarlo a messaggiare con lui.
- **Logout:** l'utente sarà in grado di uscire dal proprio account. In questo modo perderà il suo stato di utente autenticato e dovrà eseguire di nuovo la procedura di login per riottenerlo e poter accedere di nuovo al social.
- **Gestisci impostazioni:**l'utente avrà la possibilità di gestire impostazioni riguardanti:
  - **Estetica:** il tema e la dimensione dell'UI.
  - **Filtri post:** gestire i post da visualizzare nella home, scegliendo se includere post spottati e/o i post dell'utente.

– **Mappa:** gestire dimensioni e confini della mappa.
– **Sicurezza:** gestire il cambio password.
– **Logout:** effettuare il *logout*.

2.1.2 Requisiti non funzionali

- **Framework:** primo sviluppo in *Android Studio* utilizzando il linguaggio di programmazione *Kotlin* e secondo sviluppo multi-piattaforma con framework *Flutter*.
- **Database:** realizzato utilizzando le funzionalità di *Firebase Realtime Database*
- **Storage:** persistenza di file con *Firebase Storage*.
- **Authentication:** gestione degli account degli utenti con *Firebase Authentication*.

2.1.3 Diagramma dei casi d'uso

Una volta descritti nel dettaglio i requisiti che l'applicazione dovrà soddisfare, abbiamo ritenuto utile riassumerli in forma grafica utilizzato il diagramma ULM dei casi d'uso riportato in Fig. 1. Abbiamo scelto, vista la semplicità del software, ma anche per semplicità di lettura, di estrarre i casi d'uso direttamente dai requisiti. Dunque viene omessa nel seguito una descrizione puntuale di ciascuno di essi, in giustificazione del fatto che maggiori informazioni utili a comprendere la responsabilità del singolo caso d'uso possono essere trovate nella Sec. 2.1.1.

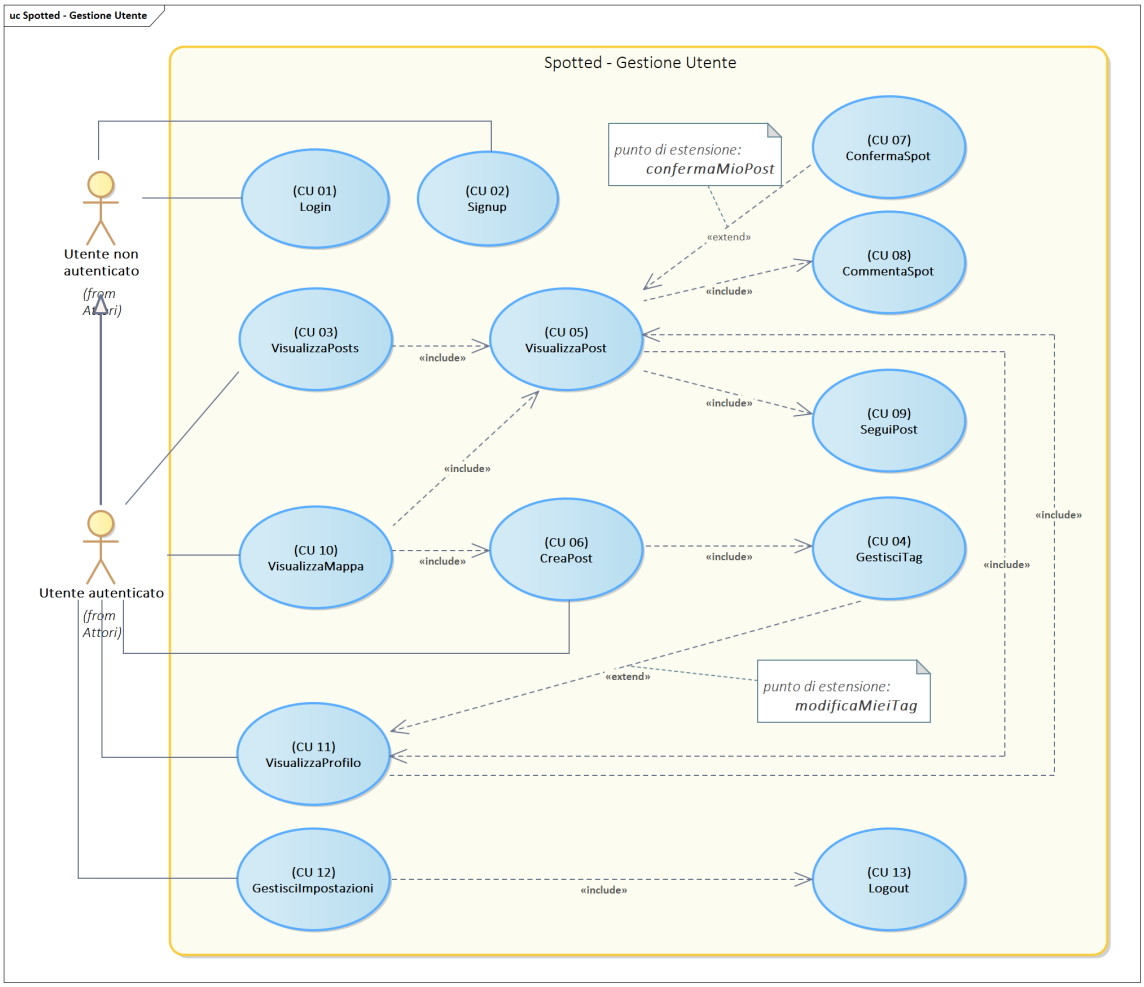

Figure 1: Diagramma dei casi d'uso

Nel diagramma sono state attentamente espresse le dipendenze tra i casi d'uso. Rispetto a ciò che è già stato discusso nei requisiti funzionali, possiamo chiarire i seguenti aspetti che emergono dal diagramma:

- Indichiamo con *Utente non autenticato* e con *Utente autenticato* gli unici due attori del sistema. Importante notare la relazione di generalizzazione che esiste tra essi.
- Il caso d'uso *VisualizzaMappa* include i casi d'uso *VisualizzaPost* e *CreaPost* perché a partire da interazioni sui segnalini della mappa, l'utente potrà visualizzare un post relativo ad un determinato luogo o crearne uno in un luogo personalizzato.
- Il caso d'uso *GestisciTag* rappresenta un punto di estensione per il caso d'uso *VisualizzaProfilo* perché solo in condizione che il profilo che l'utente sta visualizzando è il proprio, sarà possibile gestire i propri tag.
- Il caso d'uso *VisualizzaProfilo* include ed è incluso dal caso d'uso *VisualizzaPost* perché visualizzando un post, l'utente potrà visualizzare il profilo dell'autore (a patto che non sia un post anonimo) e a partire da un profilo, l'utente potrà visualizzare i post creati e seguiti.

2.2 Architettura generale

Una volta compresi i requisiti che delineano il dominio del problema, abbiamo proceduto alla creazione di un'architettura per il progetto. La scelta di questa architettura è stata determinata dalla strategia fondamentale adottata per dare forma al dominio della soluzione. Di conseguenza, come sarà illustrato di seguito, abbiamo deciso di adottare questa stessa strategia anche nella realizzazione del progetto *Flutter*.

Innanzitutto è bene notare che alla base dello sviluppo in *AndroidStudio*, siamo rimasti fedeli al pattern architetturale *Model-View-ViewModel*, il quale fornisce già di per se un'ottima classificazione dei diversi componenti che interagiscono tra loro alla realizzazione dell'interfaccia grafica e al suo popolamento con le informazioni possedute dal *Model*. Per cui non sorprende la presenza delle cartelle `view` , `viewmodel` e `model` all'interno del progetto.

Nel dettaglio, le directory che definiscono l'architettura può essere schematizzata come mostrato in Tab. 1.

Directory	**Descrizione**
`/adapter`	Le sottoclassi delle classi astratte `Adapter<T>` e di `BaseAdapter` . Definiscono gli adapters necessari per popolare i *RecyclerView* e i *GridView* utilizzati all'interno delle varie schermate.
`/enums`	Le enumerazioni di costati dell'applicazione. Il loro utilizzo ci ha semplificato notevolmente lo sviluppo, permettendo, ad esempio, di defire domini di appartenenza per determinati valori di classi del modello. Ne è un esempio la enum `Locations` che elenca i possibili luoghi tra cui scegliere al momento della creazione di un post.
`/extension`	Le classi di utilità per il progetto. Il file `BindingAdapters.kt` ad esempio, contiene funzioni di binding aggiuntive per le viste che risultano in alcuni casi estremamente utili; come ad esempio la possibilità di inserire del testo di *markdown* all'interno di una *TextView*. Il file `ObservableViewModel` invece gioca un ruolo di vitale importanza per la maggior parte dei *view models* dell'applicazione. Infatti l'impossibilità di utilizzare la gerarchia multipla in *Kotlin*, pone dei problemi nel momento in cui vogliamo creare una sottoclasse sia di `ViewModel` che di `Observable` , necessaria per la realizzazione del *two-way data binding*.
`/extension/function`	Le classi contenenti le *Extension Functions* di *Kotlin*. Anche queste sviluppate per migliorare la leggibilità del codice fornendo maggiore astrazione rispetto ad operazioni prolisse e diminuendo la ridondanza.

`/interfaces`	Le interfacce definite nel progetto. Ne è un esempio l'interfaccia `Validable` che implementata da alcuni oggetti del *Model* stabilisce da contratto la presenza di un metodo in carica di verificare che una data istanza di modello è adatta ad essere inserita nel database.
`/managers`	Gli oggetti (classi ad una sola instanza, o *Singleton*) che forniscono le funzionalità di più basso livello su cui abbiamo costruito la logica di business. Questi rappresentano pertanto dei componenti fondamentali dai quali dipendono fortemente le classi di *ViewModel*, come ad esempio le operazioni sul database remoto o la persistenza di informazioni sulla memoria secondaria del dispositivo.
`/model`	Le classi delle entità della logica di business.
`/view`	Le *Activities*, i *Fragments* e i *View Holders* che costruiscono le varie schermate e componenti grafici dell'applicazione. Ciascuno di essi è in relazione con un file *.xml* nella direcory `/res/layout` .
`/viewmodel`	Le classi di *View Model* che realizzano il binding tra i componenti delle viste e gli oggetti del model, comunicando con i *managers*.

Table 1: Le directory che definiscono l'architettura del progetto

2.3 Database

Vediamo nel seguito come abbiamo deciso di strutturare le classi del *model* e come di conseguenza il *database*.

2.3.1 I motivi alla base della scelta

Come precedentemente affermato, la nostra applicazione si appoggia sul *DBMS* non relazione offerto dal servizio di *Firebase Realtime Database*. I motivi di questa scelta risiedono sostanzialmente nella natura dell'applicazione.

Trattandosi essa di un *social*, non ci sono grosse moli di dati che l'utente produce per sé stesso, bensì, eccetto alcuni dati relativi alle impostazioni, i dati vengono scambiati con gli altri utenti, quindi non sorge una particolare esigenza di un DBMS locale. Inoltre, essendo la struttura delle classi del modello decisamente semplice, è stato scelto di utilizzare questo servizio al posto dell'alternativa offerta da *Firebase Firestore* in quanto più flessibile e semplice . Un ulteriore motivo della scelta risiede nella necessità di rilevare modifiche fatte alla sorgente di dati in tempo reale. Ad esempio è necessario sapere quando uno dei post seguiti viene modificato per poterlo notificare, così come è necessario sapere quando altri utenti commentano un post, al fine di creare una chat in tempo reale.

2.3.2 Progettazione della base di dati

Per quanto riguarda la strutturazione delle entità da memorizzare nella base di dati, essendo il database di riferimento non relazionale, abbiamo seguito una metodologia piuttosto *agile*. A seguito della progettazione dell'architettura interna, ci siamo resi conto che le entità che esistono all'interno della nostra logica di business sono le seguenti:

- `User` : un utente con tutte informazioni rilasciate al momento della registrazioni e acquisite nel corso dell'utilizzo dell'applicazione. Possiede riferimenti laschi ai post seguiti e a quelli creati.
- `Post` : un post di ricerca di una persona. Possiede relazioni di composizione con i commenti e riferimenti laschi con l'autore e gli utenti che lo seguono.
- `Comment` : un commento di certo utente ad un certo post. Possiede un riferimento lasco all'utente autore ed è in relazione di composizione con il post di appartenenza.
- `SettingMenu` : una sezione delle impostazioni. Possiede una relazione di composizione con i `SettingItem` .

- `SettingItem` : una singola voce di impostazione con la quale l'utente può interagire.

Eccetto che per le ultime due classi, la nostra applicazione comunica con il database remoto per reperire le istanze necessarie delle classi del model. Essendo al momento le impostazioni statiche e costanti, non è stato necessario caricare sul database le loro istanze, ma le abbiamo salvate come *"hardcoded"* all'interno del *SeederManager*.

Figure 2: La struttura del database remoto

Una volta chiarita la struttura delle classi del modello, abbiamo derivato facilmente la struttura del database. Questa è composta esclusivamente da due nodi, `/posts` e `/users` . Ora, poiché dalla teoria sappiamo che in una base di dati non relazionale il tedioso problema dei *joins* è risolto abolendo di netto l'operazione, è stato necessario da parte nostra inserire comunque dove opportuno dei riferimenti tra gli oggetti per poi utilizzarli per effettuare delle query in cascata. In effetti, caricare un generico post comporta il sequenziale caricamento dell'oggetto relativo all'utente che ne è autore al fine di conoscere il suo nome, la sua immagine di profilo etc. La struttura complessiva è riportata in Fig. 2, dove per chiarezza di esposizione è stato l'asciato un solo oggetto in ognuno dei due nodi.

2.3.3 Strategie di ottimizzazione

Mantenendo chiaro l'obiettivo di minimizzare l'interazione con il *database* riducendo quindi il flusso di rete necessario e i corrispondenti tempi di caricamento, abbiamo adottato principalmente tre strategie di lavoro:

- Meccanismi di cache e approccio "lazy": all'avvio dell'applicazione carichiamo una discreta quantità di utenti che fungeranno da cache successivamente. Una volta approdati alla home, al fine di mostrare nella RecyclerView la lista di post, carichiamo "pagine" di dimensione fissa di posts

coerentemente con la profondità dello scroll da parte dell'utente sulla lista. Quando l'utente cliccherà su di un post per visualizzarne tutte le caratteristiche, carichiamo sul momento seguendo un approccio lazy, gli utenti autori dei commenti del post. Naturalmente, il componente che svolge questa funzione, il *DataManager* , è stato progettato per ricercare sempre in cache il dato prima di richiederlo effettivamente al *databese*.

- Minimizzare frequenza di richieste e massimizzare i richiesti: poiché ogni richiesta comporta di per sé dei tempi di connessione e di processamento del servizio (sebbene Firebase nativamente incanali le richieste multiple su una singola connessione web socket), è buona norma chiedere liste di oggetti invece che singoli oggetti. Per questo motivo utilizziamo il *QueryBuilder* di *Firebase* per ordinare le foglie di un nodo per la loro chiave (lo UID) e caricare insiemi di oggetti.
- Minimizzare il peso degli oggetti: poiché la nostra applicazione contiene potenzialmente un numero elevato di utenti e ancor di più di post è essenziale ridurre al minimo la dimensione delle informazioni memorizzate, pur sempre mantenendole integre. A tale scopo sono state effettuati dei cambiamenti sulle classi del modello nel corso del tempo di sviluppo; i più rilevanti sono i seguenti.
  - Passaggio da `Date` a `Long` per la memorizzazione delle date: la serializzazione di un oggetto `Date` del *Java* comporta la memorizzazione di diverse delle sue proprietà pubbliche. Inoltre mina l'interoperabilità nel caso in cui un'altra applicazione deve riuscire a leggere dalla stessa fonte di dati, come d'altronde è nel nostro caso. Adottiamo dunque, come visibile in Fig. 3, la convenzione di salvare esclusivamente il timestamp, ovvero il numero di millisecondi passati dalla data di riferimento, il 01 gennaio 1970 con in UTC.

Figure 3: Prima e dopo il passagglio da `Date` a `Long` per la memorizzazione delle date.

  Per dovuta comodità, inseriamo all'interno delle classi che adottano queste convenzioni (`Post` , `User` e `Comment`) un *getter* pubblico, ma escluso dalla serializzazione con l'annotazione *@get:Exclude*, che fornisce l'oggetto `Date` a partire dal timestamp.
  - Utilizzo delle annotazioni *@Exclude* e *@JvmField* per escludere campi dalla serializzazione: ogni qualvolta risolviamo e associamo ad un post o ad un commento l'oggetto relativo al loro autore, dobbiamo accertarci che tale oggetto non venga serializzato insieme al post o al commento stesso.
  - Utilizzo delle enumerazioni: come spiegato nella Sec. 2.5.1, delle enum teniamo traccia sul database esclusivamente il nome identificatore dell'istanza, ignorando il valore delle sue proprietà.

2.4 Mokup

Definita la struttura del *database*, continuiamo con la progettazione delle schermate della *UI*. Studiando il diagramma dei casi d'uso in Fig. 1, siamo riusciti a derivare in modo decisamente naturale due navigazioni all'interno dell'applicazione: la prima molto semplice, per un l'attore *Utente non autenticato* consente di spostarsi tra le pagine di login e di registrazione; la seconda, più complessa, permette all'attore *Utente autenticato* di usufruire dei servizi del social.

Per quanto riguarda la prima, riportiamo in Fig. 4 i mockup ad essa relativa. Il *fragment* di registrazione è stato diviso per comodità in due schermate.

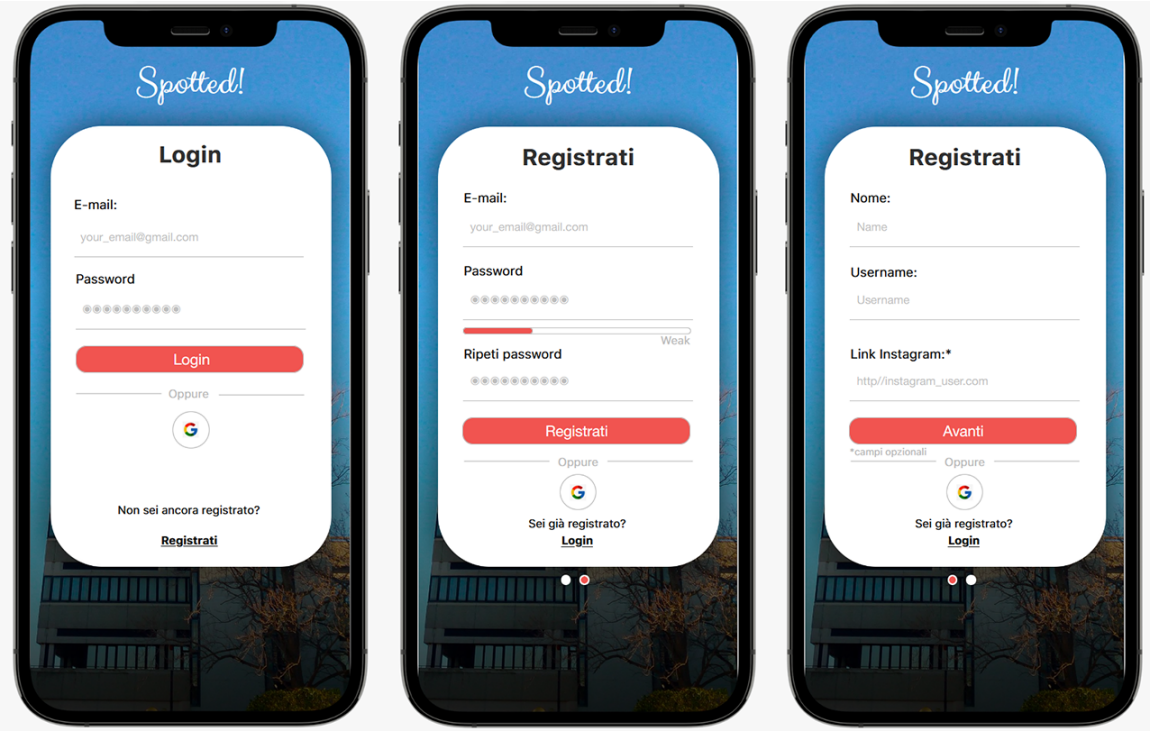

Figure 4: Mokup *LoginFragment* e *SignupFragment*

Seguono invece, in Fig. 5, in Fig. 6 e in Fig. 7, i mockup relativi alla seconda navigazione. Abbiamo omesso i mockup delle schermate della mappa e delle impostazioni. La prima per motivi di semplicità, essendo una schermata estremamente scarna dal punto di vista dei componenti. La schermata delle impostazioni invece, è stata realizzata ricalcando quella del sistema operativo *Android*.

2.5 Sviluppo

Dopo aver progettato l'architettura dell'applicazione e del database, elenchiamo nel seguito i tasselli più importanti dello sviluppo dell'applicazione, mettendo in luce i componenti che hanno richiesto un maggiore sforzo lavorativo.

2.5.1 Il ruolo delle enum

Reputiamo opportuno spendere qualche parola sui due vantaggi principali che abbiamo ottenuto dall'utilizzo delle *enum*:

- In primo luogo è molto comodo lavorare con una enum rispetto che con una classe, perché il precompilatore dell'IDE conosce già le sue possibili instanze, quindi fornisce maggiore supporto.
- Inoltre riducono la quantità di informazioni che vengono memorizzate sul *database*, poichè viene registrato esclusivamente il nome identificatore dell'instanza, invece che tutti i campi come accade per gli oggetti. Ciò ha come conseguenza una migliorata interoperabilità. Nello sviluppo dell'applicazione in *Flutter* infatti, ci siamo resi conto che, dovendoci interfacciare con la stessa sorgente di dati, alcune informazioni non erano perfettamente interoperabili. Ad esempio un grande problema lo abbiamo riscontrato nelle icone. Mentre in *AndroidStudio* utilizziamo una rappresentazione *UNICODE* del carattere del font material font `material_design_icons.ttf` , in *Flutter* occore conoscere il *code point* dell'icona per poi convertirlo in un'istanza di `IconData` . Onde evitare tediose conversioni, abbiamo trasformato la classe `Tag` (che d'altronde possiede un insieme finito e determinato di istanze) nella *enum* `Tags` . In questo modo il *database* è ignaro del valore dei campi `title` e `icon` lasciando al client il compito di risolverli. La differenza tra le due implementazioni è riportata per chiarezza del discorso in Fig 8.

Figure 5: Mokup *FirstPageActivity* e *HomeFragment*

2.5.2 I Managers come classi di basso livello

Come affermato in precedenza, i singleton *Managers* gettano le basi di funzionamento dell'intero software.

Nello sviluppo, non è stata necessaria una particolare flessibilità per i componenti di basso livello. In altre parole, ad ogni funzionalità principale abbiamo associato un oggetto *Manager* che la gestisce. Ciò giustifica la scelta di non anteporre interfacce tra queste classi e quelle di alto livello che le utilizzano, come insegna il principio di inversione delle dipendenze, ma che avrebbe complicato quasi inutilmente l'architettura del progetto.

Per chiarezza, ciascun loro metodo è stato adeguatamente commentato nel codice. Nel dettaglio i managers che soddisfano le funzioni più salienti sono i seguenti:

- `AccountManager` : il componente in carico di gestire l'autenticazione e la registrazione degli utenti. Si interfaccia con il servizio *Firebase Authentication* per la memorizzazione delle credenziali degli accounts. Utilizza la persistenza per memorizzare la sessione dell'utente ed evitare il ripetersi della procedura di *login* ad ogni avvio dell'applicazione. Una volta autenticato l'utente, si utilizza l'*UID* assegnatogli da *Firebase* per reperire le sue informazioni dal nodo `/users` nel database.

- `DatabaseManager` : il gestore della base di dati. Come spiegato nella Sec. 2.3, il nostro progetto si appoggia al servizio *Firebase Realtime Database* per la persistenza dei dati. Poiché sono molteplici

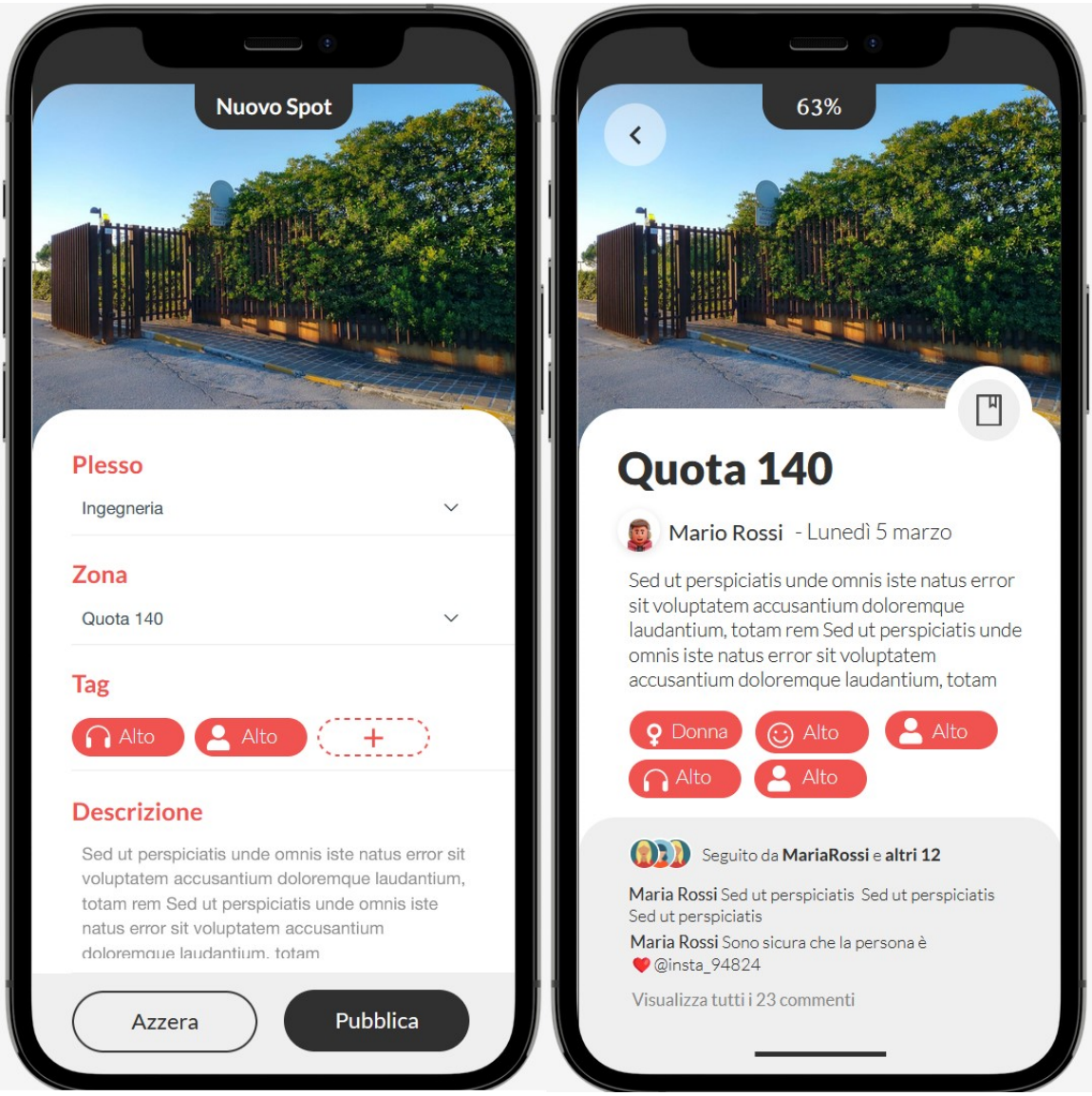

Figure 6: Mokup *NewPostFragment* e *ViewPostActivity*

i tipi di interazione con lo stesso, il manager offre dei metodi estremamente astratti e flessibili.

Le sue caratteristiche principali possono essere riassunte come segue:

- Attraverso l'utilizzo delle *generics* di *Kotlin* e delle *keywords inline* e *reified* siamo riusciti ad eseguire il *marshalling* degli *snapshots* ricevuti dal *databse* indipendentemente dalle classi di destinazione.
- Il *manager* offre anche di metodi per osservare i cambiamenti sotto di un nodo, in modo da poter attivare un *listener* in caso di avvenuti cambiamenti (funzionalità molto utile nelle *chats* di commenti dei post).
- Il *manager* permette anche di eseguire delle *query* complessi e attivare meccanismi di paginazione dei dati. Questa è una funzione strettamente necessaria per poter caricare con un approccio *lazy* i post nella schermata della *home*, considerato il potenzialmente elevato numero di questi oggetti.
- Infine il manager permette anche di eseguire l'*upload* di *files* sul servizio di *Firebase Storage*, funzione sfruttata per permettere agli utenti di caricare una propria immagine di profilo.

- `DataManager` : offre una maggiore astrazione del *DatabaseManager* in modo da codificare operazioni complesse, ma ricorrenti. Si occupa ad esempio di caricare un oggetto utente richiedendo al *database*

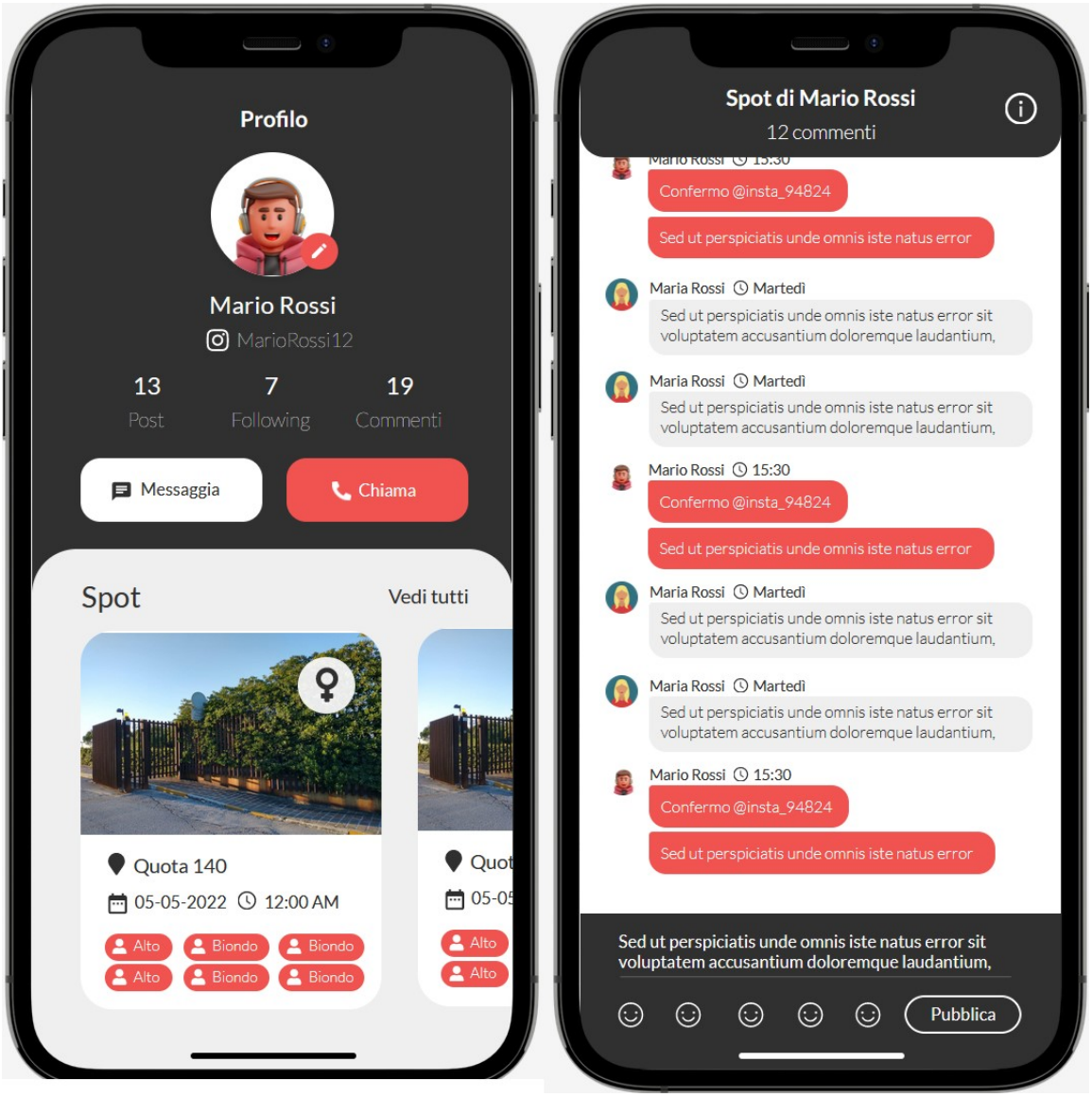

Figure 7: Mokup di *AccountFragment* e *CommentsActivity*

a catena i post da lui seguiti e da lui creati. Offre anche meccanismi di memorizzazione di nuove informazioni che rispettano le anomalie derivanti dalle inevitabili ridondanze di una base di dati *NoSql*.

- `SeederManager` (solo *AndroidStudio*): a scopo di testare l'applicazione, il *SeederManager* si occupa di popolare il *database* con informazioni verosimili, ma casuali e fasulle.
- `IOManager` : offre meccanismi di persistenza sul dispositivo. Nella nostra applicazione, gli unici casi che necessitano di utilizzare della persistenza, sono quelli dove è sufficiente memorizzare piccole stringhe di dati. Per questo motivo abbiamo ritenuto molto conveniente appoggiarci alle *APIs* di *SharedPreferences*. Il *manager* possiede dunque tre metodi, uno per leggere il valore di una data chiave, uno per scriverla e uno per rimuoverla.
- `NotificationManager` (solo *AndroidStudio*): si occupa di gestire le notifiche. Essendo quest'ultima una funzionalità decisamente prolissa in *AndroidStudio*, il *manager* offre un metodo che astrae completamente il codice necessario alla definizione e creazione dei canali di notifica così come il controllo dei permessi necessari.
- `WorkerManager` (solo *AndroidStudio*): proprio come per le notifiche, anche il *deploy* dei `WorkRequest` risulta abbastanza prolisso. Il *manager* offre un metodo per effettuare lo *scheduling* dei

```kotlin
// Kotlin
enum class Tags(val icon: String) {

    ALTO( icon: "\uDB83\uDEFC"),
    BASSO( icon: "\uDB83\uDEFC"),
```

```dart
// Dart
enum Tags {

  ALTO(Icons.height),
  BASSO(Icons.height),
```

Figure 8: La enum `Tags` in *Kotlin* e in *Dart*, con diversi valori e tipi per il campo `icon`

`PeriodicWorkRequest` , l'unica tipologia utilizzata dalla presente applicazione. In particolare facciamo uso del manager, coordinatamente con il `NotificationManager` per controllare eventuali modifiche ai post seguiti da un utente e comunicarle con delle apposite notifiche che se cliccate caricano l'*activity* del post di riferimento; proprio come richiesto dai requisiti funzionali nella Sec. 2.1.1.

2.5.3 Personalizzazione della NavigationBar

L'implementazione della bottom bar, la cui implementazione finale è visibile in Fig. 9, ha richiesto diverse ore di lavoro. Come è possibile notare dai mockup, il layout risulta essere abbastanza particolare. Essa non costituisce la parte inferiore dello schermo, bensì fluttua sopra i *fragments* che vengono scelti agendo su di essa.

Figure 9: La *NavigationBar* realizzata a partire dai mockup

Particolarmente impegnativa è stata la realizzazione dell'animazione del cerchio bianco al cambio di *fragment*. Utilizzando delle funzioni matematiche come interpolatori temporali (raccolte nella enum `TimesInterpolator`) siamo riusciti ad implementare l'animazione voluta lavorando a basso livello.

2.5.4 Confini della mappa e posizionamento dei segnalini

Per quanto riguarda la mappa, abbiamo utilizzato le *APIs* del servizio gratuito di *OpenStreetMap*. Grazie alla approfondita documentazione e alla sviluppata comunità di tale servizio siamo riusciti con un po' di sforzo a personalizzarla per rispondere alle nostre esigenze. In particolare sono state due le difficoltà incontrate:

- Centrare la mappa sulla città di Ancona: dal momento che l'applicazione è al momento limitata ai luoghi della città di Ancona, eravamo consapevoli fin da subito che la mappa dovesse rimanere fissa su di essa. Al fine di impedire movimenti al di fuori di essa, ci serviva purtroppo il pieno controllo degli eventi di `onPan` , `onDraw` e `onZoom` . Quindi abbiamo creato una sottoclasse della classe `MapView` di *OSM* e, facendo l'*override* dei metodi `onTouchEvent` e `dispatchDraw` , abbiamo personalizzato il loro comportamento in modo da richiamare i listener richiesti. Successivamente abbiamo importato ed istanziato la nuova classe nel file `.xml` del *fragment*.
- Mostrare segnalini con icone personalizzate: la logica scelta per i segnalini non è banale. Come è possibile vedere in Fig. 10, a seconda del livello di *zoom* della mappa, aumenta e diminuisce la tolleranza per segnalini vicini. Quando due o più segnalini si trovano a distanza inferiore della tolleranza attuale, vengono aggregati in un unico segnalino chiamato *multiMarker*. Il *multiMarker*

a differenza del semplice *marker* non è relativo ad un post, ma ad un gruppo di segnalini semplici. Dunque non mostrerà come questi ultimi l'immagine di profilo dell'utente autore del post, bensì un numero intero rappresentate il numero di post aggregati.

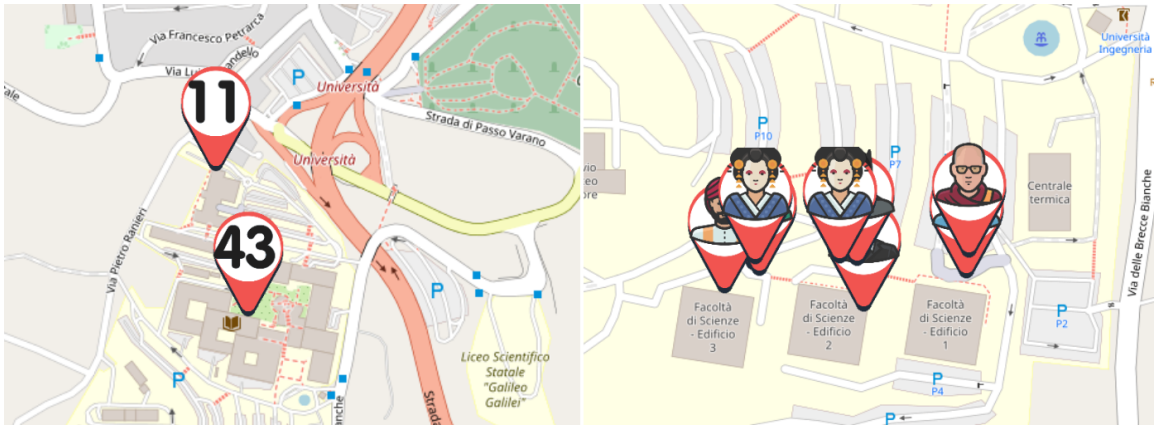

Figure 10: Comportamento dei segnalini in relazione al livello di zoom della mappa

Per far ciò è stato necessario costruire una *bitmap* ad arte da poter essere assegnata al segnalino in questione. Abbiamo innanzitutto realizzato dieci immagini in formato `.png` , uno per ogni cifra, poi, come è possibile vedere nel `BitmapManager` , le abbiamo selezionate e posizionate su di un `Canvas` per costruire l'immagine.

Per quanto riguarda invece la posizione dei post, è stato sufficiente associare delle coordinate di latitudine e longitudine a ciascun luogo contenuto nella enum `Locations` .

2.5.5 Impostazioni modulari

Nella realizzazione della schermata delle impostazioni, abbiamo scelto un approccio estremamente flessibile. Ciò è stato reso possibile dalla modularizzazione delle stesse. Invece che definire ciascun elemento come hardcoded all'interno della vista `SettingsFragment.xml` , come è possibile vedere in Fig. 11, abbiamo creato due nuovi layout, `SettingMenu.xml` per una sezione di impostazioni e `SettingItem.xml` per la singola impostazione, e a ciascuno di essi abbiamo associato una nuova classe. Compiendo questo sforzo aggiuntivo, è ora semplicissimo aggiungere una nuova impostazione. Proprio come detta l'*Open Closed principle*, infatti, basta aggiungere istanze delle due classi, senza dover modificare nulla di quanto già fatto.

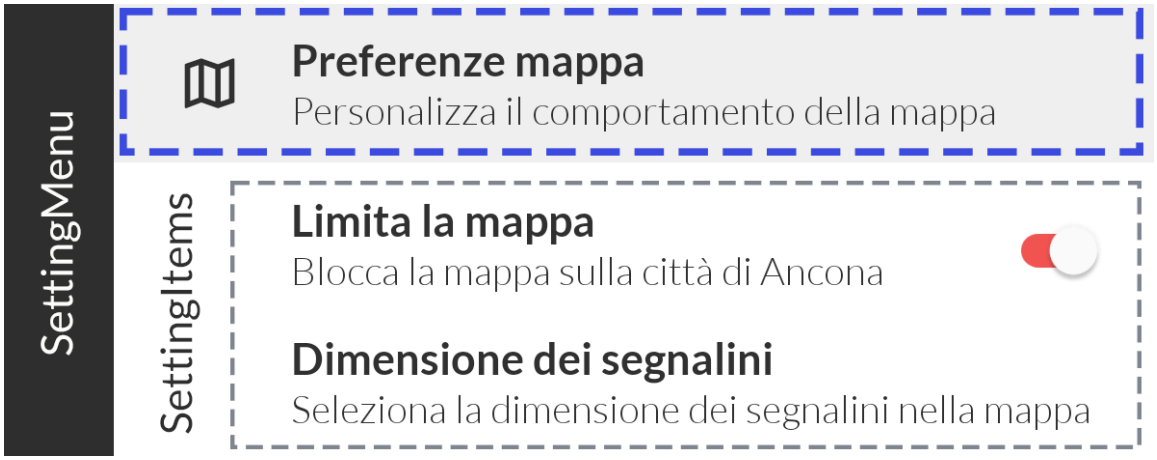

Figure 11: Ciascun `SettingMenu` contiene una lista di `SettingItem`

Al momento le impostazioni non vengono caricate dal database, ma generate come liste di oggetti delle due classi dal `SeederManager` .

2.5.6 Tema chiaro e tema scuro

Una funzionalità che tenevamo ad implementare sin dalle prime fasi di sviluppo è il supporto per il tema scuro. Utilizzando il file `/night/colors.xml` siamo riusciti con relativa semplicità ad implementarla. Invece che utilizzare colori come stringhe hardcoded all'interno delle viste, abbiamo definito la nostra palette di colori per tema chiaro e per tema scuro all'interno dei due *files* `colors.xml` e referenziato queste risorse in tutta l'applicazione. In Fig. 12 mostriamo i risultati di questa funzionalità nella schermata delle impostazioni.

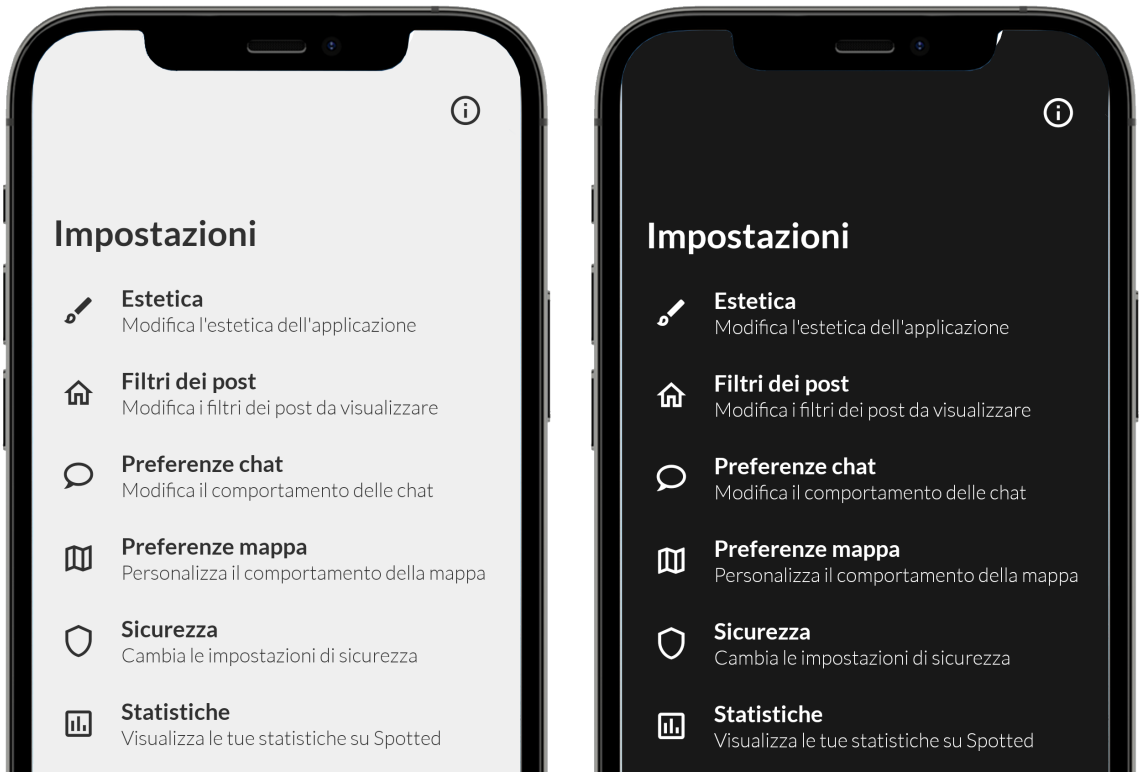

Figure 12: Semplice dimostrazione della dinamicità della palette di colori rispetto al tema di sistema

2.6 Testing

Durante lo sviluppo abbiamo ritenuto opportuno scrivere alcuni test sulle funzionalità che cruciali che emergevano. Questi, non li abbiamo progettati solo per testare quella determinata funzionalità sul momento, ma in quanto certificati che ne garantissero il funzionamento in visione di futuri cambiamenti al resto del codice. I test effettuati ricadono in due categorie: test unitari con libreria *jUnit* e test d'insieme con la libreria *espresso*. Gli ultimi agiscono direttamente sull'interfaccia grafica in autonomia per testare il funzionamento esplicito all'utente di determinate azioni richiedenti la coordinata azione di diversi componenti dell'applicazione.

Iniziamo con il dire che nello sviluppo della prima categoria di test ci siamo fin da subito imbattuti in difficoltà legate al testing dei componenti più importanti nella nostra applicazione, i *managers*. Perseverante era l'errore `java.lang.RuntimeException` che abbiamo poi scoperto scaturire dalla necessità delle librerie di *Firebase* di lavorare su un'applicazione realmente in esecuzione nell'ambiente *Android*. Abbiamo risolto questi problemi utilizzando la libreria di *mock* per *Kotlin* *MockK*. *MockK* ci permette di creare oggetti "fasulli" che simulano il funzionamento di alcuni metodi necessari di queste APIs che non possiamo normalmente richiamare da un test *jUnit*. Un esempio di applicazione di questa metodologia può essere trovata nel test `IOManagerTest.kt` , dove eseguiamo il mock delle *SharedPreferences* le quali APIs naturalmente necessitano di comunicare con il sistema operativo *Android* per poter funzionare.

Altri *jUnit* test sono stati condotti sulla validazione dei post, sulle *extension functions* che vengono utilizzate di frequente in vari punti del codice e sulle strategie di caching del `DataManager` nel reperire oggetti utente.

Per quanto riguarda i test sulla *UI* ci siamo spostati nella directory convenzionale `/src/androidTest/java` e abbiamo scritto dei test per garantire il funzionamento della funzionalità di logout, quella di login e la correttezza della validazione al momento dell'inserimento di un nuovo post.

Seppur abbiamo incontrato alcune difficoltà nell'attendere la fine del caricamento di una nuova *activity* prima di continuare ad eseguire delle azioni automatizzate su di essa, siamo poi riusciti a venirne a capo utilizzando la comoda funzione `Intents.intended()` di *Espresso* che permette di catturare gli *intent* che vengono lanciati dall'applicazione e attendere la *activity* di riferimento.

3 Sviluppo Flutter

L'applicazione in Flutter cerca di replicare, per quanto possibile, la versione in Android. In effetti sebbene abbiamo introdotto delle piccole novità e cambiato alcune semplici dinamiche, quello di cambiare le funzionalità dell'applicazione non era un nostro scopo.

3.1 Discussione dei requisiti

La logica dell'applicazione è stata mantenuta coerentemente con quanto visto precedentemente, ma con un livello di dettaglio minore. Come conseguenza di ciò, abbiamo ritenuto opportuno elencare nel seguito le differenze rispetto ai requisiti visti nella Sec. 2.1 per l'applicazione in *AndroidStudio*, invece di ripetere gran parte di essi.

3.1.1 Requisiti funzionali

Nello sviluppo dell'applicazione in *Flutter* coerentemente con quanto richiesto dalle specifiche d'esame, abbiamo implementato esclusivamente le funzionalità essenziali all'utilizzo del social.

L'utente deve certamente potersi autenticare e creare un account al fine di utilizzare l'applicazione. Quindi i requisiti e i derivanti casi d'uso relativi all'attore *Utente non autenticato* devono rimanere inalterati.

Ciò che invece abbiamo tagliato sono le seguenti funzionalità:

- Possibilità di filtrare i post nella home oppure di ordinarli per campi diversi dalla data di creazione. Deve comunque rimanere possibile per l'utente la visione di tutti i post pubblicati sulla piattaforma. Inoltre l'utente deve poter visionare tutte le informazioni relative ad un post in un'apposita schermata.
- Possibilità di commentare un post in una chat in tempo reale. Deve comunque rimanere possibile per l'utente la visione dei commenti al post, magari con una sezione dedicata nella pagina di visualizzazione del post. Inoltre l'utente deve continuare a poter seguire un dato post agendo su un apposito pulsante.
- Possibilità di visualizzare i post sulla mappa. La funzionalità della mappa è certamente utile, ma fine a se stessa e indipendente dal resto dell'applicazione per tanto è possibile escluderla dai requisiti funzionali.
- Possibilità di modificare le impostazioni dell'applicazione. Non è quindi necessaria l'implementazione della schermata di riepilogo delle impostazioni.
- Possibilità di cambiare la propria immagine di profilo. Deve comunque rimanere possibile per l'utente la visione di tutte le informazioni relative al proprio account o all'account di un altro utente in una schermata apposita.
- Possibilità di ricevere notifiche in tempo reale sulle modifiche dei post seguiti.

3.1.2 Requisiti non funzionali

Per quanto concerne invece i requisiti non funzionali, possiamo affermare che rimangono pressoché invariati rispetto a quelli elencati e descritti nella Sec. 2.1.2, con l'unica differenza del terzo requisito. Infatti non essendo più richiesta la funzionalità di *upload* di immagini di profilo, non facciamo più uso delle APIs fornite dal servizio di *Firebase Storage*. Tale servizio lo utilizziamo ancora però, per visualizzare le immagini relative ai luogi degli spot.

3.2 Architettura

Per chiarezza nello sviluppo abbiamo cercato di ricalcare l'architettura impostata nel progetto in *AndroidStudio* e spiegata nel dettaglio nella Sec. 2.2. Naturalmente la riduzione dei requisiti funzionali ha comportato, per forza di cose, una riduzione dei *files* e delle *directories*. In particolare, all'interno della cartella `/lib` , quella contenente il codice puramente *cross platform*, sono stati omessi i seguenti componenti:

- Directory `/adapter` : la costruzione delle liste e delle griglie è in *Flutter* decisamente più ad alto livello rispetto a quella in *AndroidStudio*. Per cui abbiamo ritenuto opportuno popolare le funzioni `itemBuilder` direttamente sul luogo di dichiarazione del *widget*.
- Alcune classi *singleton* nella directory `/managers` : come è possibile vedere nella Sec. 2.5.2, alcuni di queste classi non sono risultate necessarie. Naturalmente il `DatabaseManager` , il `DataManager` e l'`AccountManager` sono stati tradotti per intero in *Dart* utilizzando come appoggio l'esplicativa documentazione di *Firebase* per *Flutter*.

Riteniamo doveroso, nel seguito, spendere alcune parole sull'approccio scelto riguardo l'adozione di pattern architetturali in *Flutter*. A differenza di altri *framework* come ad asempio *Xamarin* oppure lo stesso *Android Studio*, è chiaro che *Flutter* non impone uno schema rigido allo sviluppatore, bensì garantisce estrema libertà sulla scelta dell'architettura di progetto. Abbiamo ritenuto poco conveniente pertanto la scelta di adottare il pattern architetturale *MVVM* usato per lo sviluppo in *AndroidStudio* che avrebbe complicato certamente la progettazione.

Seppur non approfondito in modo particolare, l'architettura sviluppata si avvicina invece di più al pattern *MVU* che è sicuramente più vicino alla filosofia di *Flutter*. A giustificazione di quanto affermato, abbiamo cercato di strutturare in modo preciso le sottoclassi di `StatefulWidget` . Nel costruttore chiediamo in ingresso lo stato, quindi variabili o oggetti reperiti dal *database* (il model dell'*MVVM* in *AndroidStudio*). Nel metodo `build` definiamo la gerarchia dei *widget* rappresentante la *UI* della vista ed estraiamo in metodi appositi le funzioni di *update*, che vengono azionate dall'utente e potenzialmente ricreano lo stato della vista. Infine, nei casi in cui necessitiamo di modificare lo stato di `StatefulWidget` esterni rispetto a quello corrente, ricorriamo alla programmazione funzionale per definire e scambiare funzioni di *callback*.

3.3 Database

Il modello rimane lo stesso. Le uniche novità introdotte a questo punto dello sviluppo del progetto sono i piccoli cambiamenti descritti nella Sec. 2.3, dovuti ad un necessario incremento di interoperabilità dal momento in cui due *framework* e linguaggi diversi si trovano a dover leggere ed interpretare gli stessi dati. Quindi in breve:

- utilizzo di enum quando possibile per garantire l'indipendenza dai valori e dai tipi di dati memorizzati dalle stesse,
- e sostituzione di oggetti *Java* di tipo `Date` in tipi primitivi *long*.

3.4 Mokup

Per quanto riguarda i mockup non abbiamo qui alcuna differenza da far notare. Abbiamo infatti deciso di lasciare inalterate le scelte di progettazioni riguardanti lo stile e i colori delle schermate dell'applicazione.

3.5 Sviluppo

Abbiamo iniziato sin da subito lo sviluppo implementando le funzionalità essenziali dei *Managers*, quali la comunicazione con la base di dati di *Firebase Realtime Database* e quella con il servizio di *Firebase Authentication* per l'autenticazione degli utenti. Abbiamo poi continuato implementando le viste e popolando il loro stato a partire dagli oggetti restituiti da questi *managers*.

Riportiamo nel seguito alcuni punti salienti emersi dallo sviluppo in *Flutter*.

3.5.1 Serializzazione e deserializzazione di oggetti dal database

Abbiamo sin da subito incontrato dei problemi di deserializzazione di oggetti dal database. In effetti le APIs di *Firebase Realtime Database* restituiscono, a seguito di una richiesta di un nodo, un oggetto *Dart* di tipo `_InternalLinkedHashMap<dynamic, dynamic>` . Il primo step è stato quello di effettuare la conversione al tipo sicuramente più maneggevole `Map<String, dynamic>` ; scelta giustificata tra l'altro dal fatto che ciascuna chiave è sempre una stringa. Per adottare tale conversione, dopo diverse strategie, abbiamo scelto come più conveniente quella di codificare e decodificare l'oggetto restituito in formato json. Utilizzando le funzioni `jsonDecode` e `jsonEncode` , infatti, riusciamo ad effettuare un *casting* nel formato desiderato, non solo dell'oggetto di partenza, ma ricorsivamente di tutti gli oggetti al suo interno aggregati.

Successivamente siamo riusciti a convertire la mappa nell'oggetto di destinazione utilizzando la comoda libreria *json_serializable* che utilizza la libreria *Dart source_gen* per ovviare alla mancanza di *reflection* di *Flutter*. La libreria, crea in autonomia, per ciascuna classe del modello, delle classi parziali dotati di metodi per la serializzazione e deserializzazione. Rimane ora sufficiente definire, utilizzando il supporto nativo al pattern creazione *Factory* di *Dart*, il costruttore in carica di creare l'istanza dell'oggetto a partire dalla mappa *Json*.

3.5.2 Effetto "shimmer" per il caricamento delle viste

Utilizzando il pacchetto *shimmer*, siamo riusciti con facilità a donare un ottimo "look and feel" ai caricamenti dei post nell'applicazione creando il ben noto effetto "shimmer" mostrato in Fig. 13.

Figure 13: Effetto "shimmer" per il caricamento dei post

3.5.3 Carosello per i post nel profilo

Per evitare di ripetere la *HorizontalScrollView*m già implementata nel progetto in *AndroidStudio* abbiamo deciso di applicare una soluzione più estetica per visualizzare gli ultimi post creati dall'utente nella sua pagina profilo.

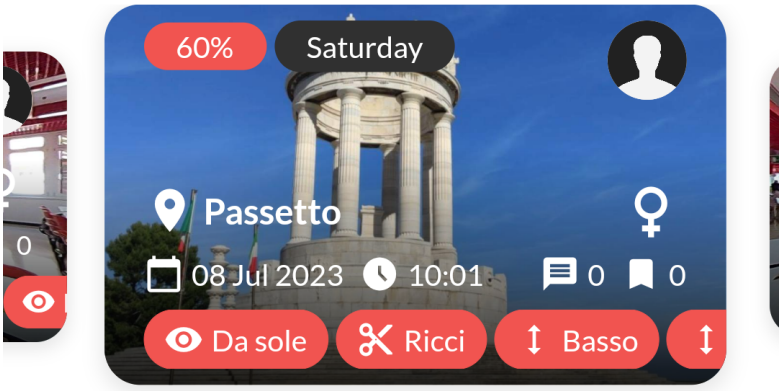

Figure 14: Carosello animato nel profilo dell'utente

A tal fine abbiamo utilizzato il pacchetto *carousel_slider* che permette di realizzare un'animazione "carosello" esattamente come la avevamo pensata. In Fig. 14 mostriamo il risultato grafico di tale scelta.

3.5.4 Supporto per il tema scuro

Anche in *Flutter* con in *AndroidStudio* abbiamo preso sin da subito la decisione di donare all'applicazione il supporto per il tema scuro. A tal fine abbiamo definito un'istanza personalizzata di `ThemeData` , mostrata in Fig. 15, e passata al parametro `theme` del *widget* `MaterialApp` , radice dell'intera applicazione.

```
// App color scheme
Palette.theme = ThemeData(
    brightness: brightness,
    splashFactory: InkRipple.splashFactory,
    splashColor: Colors.white.withAlpha(130),
    colorScheme: ColorScheme(
      brightness: brightness,
      primary: isLight ? Palette.red : Palette.white,
      onPrimary: isLight ? Palette.white : Palette.black,
      secondary: isLight ? Palette.white : Palette.black,
      onSecondary: isLight ? Palette.black : Palette.white,
      error: Colors.red,
      onError: Colors.black54,
      background: isLight ? Palette.grey : Palette.darkBlack,
      onBackground: isLight ? Palette.black : Palette.grey,
      surface: isLight ? Palette.black : Palette.darkBlack,
      onSurface: isLight ? Palette.white : Palette.darkBlack,
    ));  // ColorScheme, ThemeData
```

Figure 15: Definizione dello schema dei colori

4 Conclusioni

In conclusione a questo progetto, possiamo affermare che sono molteplici le competenze acquisite per quanto concerne lo sviluppo di applicazioni mobile. A parere del tutto personale, siamo rimasti sorpresi della facilità di utilizzo del framework *Flutter* e di come, lo sviluppo di parte dell'applicazione che ci aveva richiesto circa un mese intero di lavoro in *Android Studio*, è stato portato a termine nell'arco di una settimana.

Invece, dal punto di vista del flusso di lavoro, possiamo affermare di esserci impegnati a realizzare un'applicazione che, se da un lato mira a realizzare i casi d'uso, dall'altro non si accontenta di farlo con la filosofia del "è sufficiente che funzioni". Abbiamo infatti sempre cercato di fare quello sforzo in più che andasse oltre la risoluzione del problema del momento, ma con una visione ad ampio raggio, mirasse a trovare un buon compromesso tra l'astrazione (e quindi flessibilità) della funzionalità che si stava implementando, e l'importanza della stessa ai fini di soddisfare i requisiti proposti.

Sempre fedeli ai principi acquisiti al corso di ingegneria del software, in particolare ai principi *SOLID* per quanto concerne la progettazione delle classi, abbiamo ad esempio progettato i *managers* di comunicazione con le APIs dei servizi *Firebase* e con la memoria di massa del dispositivo, in modo completamente astratto e indipendente dal particolare uso che ne facciamo all'interno del codice. Lo stesso si può affermare per quanto riguarda la modularizzazione dei componenti come le impostazioni, come discusso in precedenza. Siamo fiduciosi sul fatto che la qualità di un progetto software vada scritta giorno per giorno nel lavoro quotidiano e che questo sforzo ci ripagherà con interessi quando rimetteremo mano in futuro sul codice scritto oggi per implementare nuove funzionalità e rendere l'applicazione pronta ad una potenziale pubblicazione sui canali di distribuzione.