

EBikeRecharge
Sviluppo di un'applicazione
Android e iOS tramite il framework Flutter

Gianluca Viganò, Youssef Zraiba

2019
Aprile

Indice

1	Introduzione	1
1.1	Obiettivi e origine del progetto	1
1.2	Scelta del framework	1
1.3	Organizzazione e sviluppo	2
1.3.1	Git e Bitbucket	2
1.3.2	Trello	3
1.3.3	Visual Studio Code	3
1.4	Struttura generale dell'applicazione	3
2	Flutter	5
2.1	Stateless e Statefull Widget	5
2.2	Principali Widget	8
2.3	Material Design	9
2.4	Animation	13
3	Gestione degli utenti	14
3.1	Firebase	14
3.2	Schermata di avvio, tipologia di Utente e pagina Profilo	14
3.3	Registrazione	16
3.4	Utente anonimo	16
3.5	Codifica	17
4	Gestione e struttura del database	18
4.1	Progettazione iniziale	18
4.1.1	Schema Concettuale	18
4.1.2	Schema Logico	19
4.2	Gestione dati in Firebase	19
4.3	Utilizzare il Cloud Firestore	21
5	Mappa e sue funzionalità	22
5.1	Scelta del servizio API	22
5.2	MapPage	23
5.3	Funzionalità	25

1 Introduzione

Nel 2017 in Italia la produzione di veicoli a due ruote a pedalata assistita è aumentata del 48% e il mercato dell'e-bike ha ottenuto un incremento del 19% per un totale di 148.000 unità vendute. [Lombardo(2018)]

1.1 Obiettivi e origine del progetto

Scopo del progetto è creare un'applicazione che mostri all'utilizzatore, in base alla propria posizione, le più vicine stazioni di ricarica per bici elettriche. Inoltre vengono forniti numerosi servizi aggiuntivi per facilitarne l'utilizzo e la personalizzazione. L'idea è nata da un incontro con il dott. Marco Aceti, biker per passione, nell'estate 2018. Durante il colloquio ha spiegato come il mercato delle bici elettriche sia in continua espansione, e che sempre più ciclisti decidono di passare dalla tradizionale bicicletta a quella con pedalata assistita. Dunque ha raccontato la sua idea: un'applicazione che mette a conoscenza dell'utente la posizione delle colonnine per ricaricare la propria bici.

1.2 Scelta del framework

Il committente dell'applicazione desiderava che il prodotto fosse usufruibile da quanti più utenti possibile, e dunque una delle richieste era che l'app fosse accessibile sia a dispositivi Apple, con il sistema operativo iOS, sia per dispositivi Android.

Per i sistemi Android non ci sarebbero stati problemi relativi al linguaggio da utilizzare in quanto si aveva già avuto esperienza di Java, dei suoi costrutti e della sua sintassi. Inoltre il programma Android Studio avrebbe ulteriormente semplificato le cose. Il problema nasceva nel momento in cui si era deciso di sviluppare un'app per iOS: la nostra conoscenza relativa ai linguaggi (Swift e C#) era pressochè nulla e si sarebbe dovuto investire diverso tempo per impararne in modo appropriato l'utilizzo. Inoltre per poter sviluppare un'applicazione che possa poi essere eseguita su un Iphone si deve essere in possesso di un calcolatore Apple (Macbook) in quanto è necessario il programma Xcode, funzionante solo su quest'ultimo; e questo strumento non faceva parte delle nostre risorse. Non sapendo come procedere, si è deciso di

chiedere maggiori informazioni a diversi professori del corso di Ingegneria Informatica e anche al gruppo Unibg Seclab. Proprio questi ultimi ci hanno indicato come la soluzione al problema fosse un (al tempo) nascente framework di Google in grado di creare app native per entrambe le piattaforme di interesse: Flutter. Il prossimo capitolo è dedicato alla comprensione e all'utilizzo di concetti base per capire come sviluppare software mediante questo framework.

1.3 Organizzazione e sviluppo

Lavorando in un team è stato necessario impiegare del tempo per scegliere i tool relativi allo sviluppo in gruppo, in modo da organizzare il tutto nel miglior modo possibile.

1.3.1 Git e Bitbucket

Bitbucket è uno strumento di gestione del codice Git. Un progetto sviluppato con questi due tool dunque non solo risiede fisicamente sulle macchine sulle quali si testa il software, ma anche sul cloud, in modo da avere maggiore affidabilità delle versioni (con una gestione efficace relativa alle modifiche) e potendo poi accedere con un qualunque calcolatore al progetto. Per aggiungere un nuovo file al progetto è necessario scrivere `git --add nomefile`, se invece occorre applicare delle modifiche inserendo anche un commento per meglio comprendere il lavoro appena concluso basta digitare `git commit -a -m "messaggio"` (dove il parametro `-a` sta a indicare che si vuole applicare a *tutte* le modifiche e il parametro `-m` che si desidera lasciare un commento). Volendo rendere dunque *effettive* le modifiche è necessario dare il comando `git push origin master` in modo tale da inserire nel branch *master* la nuova porzione di codice. Per ottenere le modifiche aggiunte da un altro membro del team bisogna scrivere `git pull`: dopo tale comando sul proprio calcolatore sono presenti tutti i file della repository aggiornati all'ultima versione.

1.3.2 Trello

Trello è un programma che permette in modo molto rapido e soprattutto in maniera intuitiva di organizzare le mansioni e i compiti di ogni membro del team. La pagina principale consiste in una grande bacheca sulla quale sono visibili delle *schede*, ognuna con un titolo e relativa a un particolare gruppo. Per ogni scheda è poi possibile indicare il membro del team al quale è riferito il lavoro indicato, indicare una lista di azioni (in modo da vedere la percentuale di avanzamento di quella particolare scheda) e settare dei promemoria importanti che non devono essere persi.

1.3.3 Visual Studio Code

Il progetto conteneva numerosi file di estensione diversa. Infatti erano presenti file dart (per l'applicazione vera e propria), xml (per lavori specifici lato Android), plist (file descrittivo lato iOS), immagini e altri ancora. Si è quindi deciso di utilizzare un editor testuale il più generale possibile e non legato allo sviluppo di un particolare linguaggio. La scelta è ricaduta su Visual Studio Code, di proprietà Microsoft. Oltre a possedere "out of the box" funzionalità molto comode per lo sviluppo software, è estensibile con migliaia di pacchetti relativi a pressochè ogni linguaggio. Nello specifico l'estensione per Flutter non solo presenta autocompletamenti vari e molto dettagliati, ma anche una comoda funzione per fare il debugging del software gestendo il tutto con una barra che raccoglie i principali comandi.

1.4 Struttura generale dell'applicazione

Di seguito viene riportato un accenno alla struttura, alle pagine dell'intera applicazione in modo da dare un contesto e dare una visione d'insieme al lettore. A partire dal terzo capitolo ogni pagina verrà poi analizzata nel dettaglio in base ai propri componenti e alle sue funzionalità, mostrando anche immagini e codice.

Al primo accesso all'utente viene mostrata una pagina di login nella quale si può inserire la propria mail e password oppure creare un nuovo account. Se l'autenticazione o la crezione di un nuovo utente sono andate a buon fine, viene mostrata la pagina principale, la mappa, che indica tutte le stazioni di ricarica, noleggio e

manutenzione di bici elettriche presenti nel database. Oltre a possedere numerose funzioni che verranno descritte nel proprio capitolo, da questa pagina è possibile accedere all'inserimento di una nuova stazione (che deve essere in ultimo confermata dal gestore del database), e si può infine arrivare alla pagina profilo, dove si possono settare impostazioni personali come la tipologia di mappa che si vuole visualizzare e cambiare la password, si possono vedere le stazioni aggiunte dall'utente attuale ed effettuare il logout, per tornare così alla pagina iniziale di login.

2 Flutter

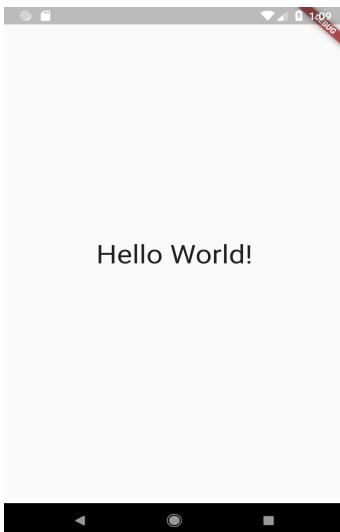
I pregi di Flutter sono numerosi e se ne vogliono citare alcuni.

Per primo si considera il linguaggio. Flutter viene sviluppato con il linguaggio (sempre appartente alla famiglia Google) Dart. Esso è multifunzione, può essere impiegato per lo sviluppo web (tant'è che è nato per sostituire Javascript) e in tale linguaggio una variabile può sia presentare una tipazione statica a compile time, sia essere definita *dynamic* e cambiare tipo più volte nel corso dell'esecuzione. Con Flutter basta scrivere una volta sola il proprio codice in Dart, e il framework penserà poi a creare i file necessari per essere eseguiti sia su iOS che su Android. Da qui ne segue un enorme risparmio di tempo e risorse, in quanto si dimezza il lavoro totale. Un altro aspetto importante è che grazie ad esso è possibile sviluppare in modo rapido, e ciò è permesso dalla funzione *Hot Reload* che consente in qualche secondo di ricaricare la particolare sezione di codice che si sta testando senza dover ricaricare l'intera applicazione. In secondo luogo tale framework presenta facilitazioni notevoli relative al design degli elementi: con poche righe e senza essere esperti di grafica si può creare qualcosa dall'aspetto tutt'altro che banale. È doveroso accennare anche alle prestazioni: i *widget* (elementi) di Flutter gestiscono automaticamente l'implementazione di funzionalità comuni quali lo scrolling, la navigazione, le icone e i caratteri in modo da rendere le performance del software paragonabili a quelle di app native sia su iOS sia su Android.

2.1 Stateless e Statefull Widget

In Flutter esiste una prima importante distinzione fra widget diversi: quelli *senza stato* e quelli *con stato*. I primi possono anche essere chiamati widget statici, in quanto la loro forma non varia durante l'esecuzione. Nell'immagine 2.1 viene mostrato un esempio di widget stateless. Innanzitutto bisogna importare il pacchetto *flutter/material.dart* che contiene tutti i principali widget. Nel metodo main è presente solo una funzione, denominata *runApp*, che ottenuto in input un widget con particolari caratteristiche, esegue sul dispositivo di test tale widget. Nell'esempio esso è chiamato MyApp ed estende la classe StateLessWidget. Un oggetto di questo

tipo deve sovrascrivere il metodo *build* che riceve in input il contesto nel quale l'app si trova (che consiste nel particolare valore delle variabili e di altri oggetti in quel preciso istante di esecuzione), e ritorna il widget vero e proprio che dovrà essere eseguito. Per ottenere una corretta struttura, anche in progetti più complessi di questo semplice esempio, è necessario introdurre la classe `MaterialApp`, e tramite il suo costruttore definire la sua *home*, cioè la pagina principale, con una `Scaffold`, definibile come l'impalcatura del widget. È necessario introdurre nel parametro *body* la classe che si vuole eseguire, la quale a sua volta estende `StateLessWidget` e presenta ancora il metodo *build*. Qui viene ritornato un widget che avrà posizione centrale (`Center`), sarà contenuto in un contenitore (`Container`, utile per decorare e aggiungere funzionalità a specifiche sezioni di codice), e mostrerà un testo con il più classico dei messaggi d'avvio (Hello World!). Questo widget non è interattivo, non cambierà mai graficamente e perciò si dice che non ha stato. Da questo esempio si può già notare come la struttura del codice flutter assuma *graficamente* un'indentazione naturale, con una gerarchia ad albero in cui c'è un padre iniziale e da cui seguono numerosi figli, padri di successivi widget.



```

1 import 'package:flutter/material.dart';
2
3 main(List<String> args) {
4   runApp(MyApp());
5 }
6
7 class MyApp extends StatelessWidget {
8   @override
9   Widget build(BuildContext context) {
10     return MaterialApp(
11       home: Scaffold(
12         body: Home(),
13       ); // Scaffold // MaterialApp
14     }
15   }
16
17 class Home extends StatelessWidget {
18   @override
19   Widget build(BuildContext context) {
20     return Center(
21       child: Container(
22         child: Text('Hello World!', style: TextStyle(fontSize: 35.0),),
23       ), // Container
24     ); // Center
25   }
26 }
```

(a) Esempio di codice di widget stateless

(b) Risultato

Figura 2.1

Nell'immagine 2.2 viene mostrato invece un widget di tipo `Statefull`. Si può notare come la prima parte del codice sia identica, con la classe `MyApp` che è ancora un widget senza stato. Ciò che cambia è la classe `Home` che ora è con stato

e presenta un unico metodo sovrascritto chiamato *createState*. La freccia => in dart è una notazione per scrivere in modo compatto un metodo che ritorna un unico oggetto, in questo caso una classe chiamata *_HomeState*, che estende *State<Home>*. Il trattino basso in dart sta a indicare un oggetto privato, a cui quindi non è possibile accedere se non in quella specifica classe. *_HomeState* possiede una variabile di tipo intero denominata *counter* (anch'essa privata grazie al trattino basso) e ha valore iniziale 0. Presenta poi un metodo denominato *buttonPressed* al cui interno risiede una funzione molto importante, la *setState*. Quando viene compilata una sezione di codice che sta all'interno di tale funzione, viene richiamata la funzione *build* del widget in esecuzione ed è quindi grazie ad essa che l'app diventa interattiva, in quanto tramite azioni dell'utente cambiano i valori delle variabili. Nell'esempio è presente un pulsante (*FloatingActionButton*) che aumenta il valore della variabile *counter*, che viene stampato a schermo. Da notare infine come si possa accedere al metodo *toString* di un oggetto in dart anteponendo al suo nome il carattere \$.

Figura 2.2: Esempio di codice di widget statefull



```

1 import 'package:flutter/material.dart';
2
3 main(List<String> args) {
4   runApp(MyApp());
5 }
6
7 class MyApp extends StatelessWidget {
8   @override
9   Widget build(BuildContext context) {
10     return MaterialApp(
11       home: Scaffold(
12         body: Home(),
13       )); // Scaffold // MaterialApp
14   }
15 }
16
17 class Home extends StatefulWidget {
18   @override
19   _HomeState createState() => _HomeState();
20 }
21
22 class _HomeState extends State<Home> {
23   int _counter = 0;
24   void buttonPressed() {
25     setState(() {
26       _counter++;
27     });
28   }
29   @override
30   Widget build(BuildContext context) {
31     return Scaffold(
32       body: Center(
33         child: Container(
34           child: Text(
35             'Hai premuto $_counter volte il pulsante',
36             style: TextStyle(fontSize: 25),
37           ), // Text
38         ), // Container
39       ), // Center
40       floatingActionButton: FloatingActionButton(
41         child: Icon(Icons.add),
42         onPressed: buttonPressed,
43       ), // FloatingActionButton
44     ); // Scaffold
45   }
46 }

```

2.2 Principali Widget

Di seguito vengono riportati i widget di cui si è fatto maggiore uso all'interno dell'applicazione.

Text

Questo widget permette di mostrare a schermo le stringhe fornite come primo argomento al suo costruttore (`Text("Testo da mostrare")`). Il testo può svilupparsi su più righe o su una sola a seconda delle costanti di layout della particolare schermata nella quale risiede il widget. Nel costruttore possono essere indicati diversi argomenti opzionali, tra cui lo stile. Se si vuole indicare un particolare stile al proprio testo bisogna inserire un `TextStyle`, cioè un'ulteriore classe che gestisce il font, l'inserimento del grassetto o del corsivo e la formattazione dell'intero testo (giustificato, allineato a sinistra o a destra e centrale).

Row, Column

Questi widget vengono spesso utilizzati per ottenere schermate ordinate e gradevoli dal punto di vista estetico all'utilizzatore. Il parametro principale di entrambe è l'argomento `children` (e non `child` come spesso accade per altre classi), proprio perchè sono pensati per contenere diversi widget disposti rispettivamente in orizzontale o in verticale. La loro grandezza in pixel sarà formata dalla somma della dimensioni di ciò che contengono e si possono ancora una volta inserire diverse preferenze di stile come la posizione rispetto all'asse principale o secondario. Nella figura 2.3 si possono vedere i due widget che possiedono quattro Container figli di diverso colore.

Stack

È simile ai due precedenti in quanto anch'esso contiene diversi widget figli. Se ne differenzia in quanto non privilegia un'unica direzione di posizionamento, ma si può indicare per ogni figlio la posizione esatta in pixel che dovrà avere sullo schermo. Questo passaggio viene effettuato racchiudendo il widget che si vuole inserire all'interno di un'istanza della classe `Positioned` (che sarà uno dei figli dello `Stack`) e indicando gli esatti pixel nel suo costruttore.

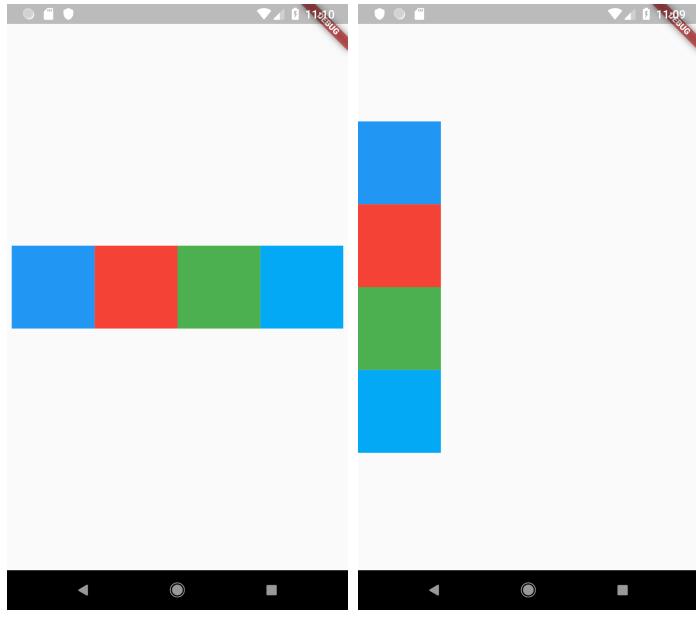


Figura 2.3

Container

Crea un elemento rettangolare che avrà come altezza e lunghezza le minime dimensioni per poter contenere il widget indicato come child. È molto utilizzato nel momento in cui si vuole migliorare graficamente una schermata in quanto tramite il parametro opzionale decoration si può introdurre una nuova classe chiamata Box-Decoration che gestisce un grande insieme di aspetti grafici come i contorni (angoli e ombre) o riempire con un colore o un'immagine il Container.

2.3 Material Design

Il Material Design è un linguaggio visuale che unisce i classici principi di un buon design con l'innovazione della tecnologia e della scienza. [material.io(2019)]

Tale design è interamente sviluppato da Google e fa uso di layout basati su una griglia, animazioni e transizioni. Si prendono ora in considerazione i widget più comuni appartenenti a questo particolare design.

Scaffold

Implementa la struttura (*impalcatura*) del layout visuale di base del material design. Si espande fino a occupare tutto lo spazio disponibile sullo schermo, inoltre quando

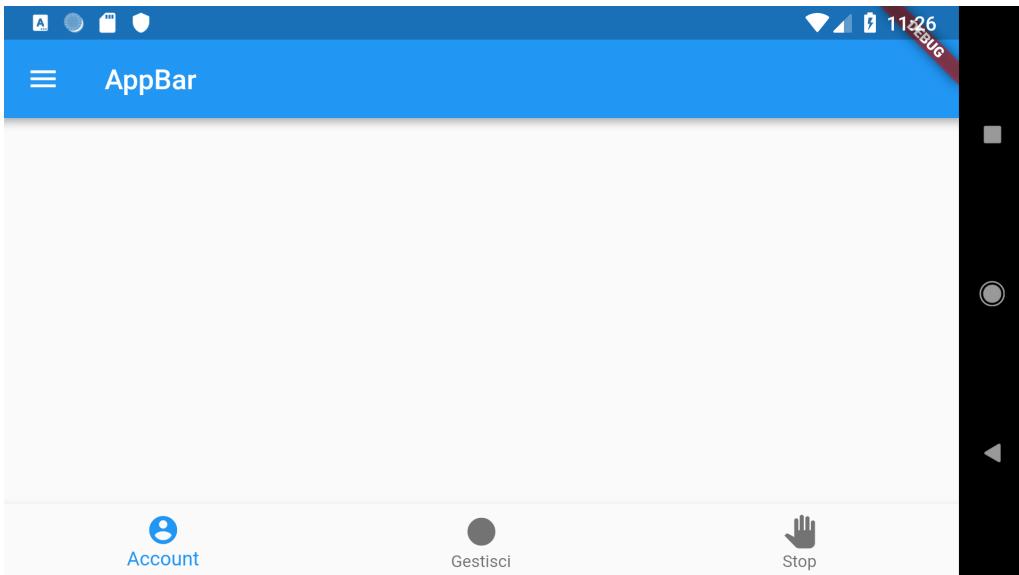


Figura 2.4: Nell'esempio viene mostrata una Scaffold, che fa da widget genitore all'AppBar posizionata in alto, al Drawer in alto a sinistra e alla BottomNavigationBar in basso con tre icone con titolo. Si noti inoltre come venga gestita in automatico la possibilità di ruotare il proprio dispositivo, cambiando l'orientazione della schermata.

appare una tastiera a seguito di un'interazione con l'utente la grandezza di questo widget cambia in base alle dimensioni di quest'ultima. Presenta nel proprio costruttore argomenti per mostare sulla schermata Drawer, Snackbar e BottomSheets, descritte di seguito.

AppBar

Un'AppBar consiste in una linea orizzontale, solitamente posizionata in alto sulla schermata, che contiene diverse azioni (IconButtons) seguite da un menù a comparsa (PopUpMenu). In questo modo si possono inserire in un unico luogo tutte le scorciatoie per passare da una schermata ad un'altra oppure si possono accoppare funzionalità utili a portata di mano per l'utente. Bisogna inserire come argomenti il titolo dell'AppBar e inoltre una serie di azioni (*actions*) sottoforma di una lista di widget.

BottomNavigationBar

È un material widget disposto in fondo allo schermo per selezionare un piccolo numero di schermate, solitamente tra le tre e le cinque. Si costruisce tramite diversi strumenti quali widget di testo, icone o entrambi. Se gli elementi figli sono minori di

quattro, la grandezza di ogni item è fissa, mentre se superiore la grandezza si adatta al numero di widget contenuti.

Drawer

Consiste in un pannello che entra in orizzontale sullo schermo solitamente dal lato sinistro della Scaffold. Come figlio è solito avere una ListView (particolare tipo di Column che gestisce anche lo scroll verticale), in modo da mostrare all'utente utili link per la navigazione all'interno dell'applicazione. L'AppBar mostra in automatico un Drawer vuoto con la classica icona del menù a comparsa se non viene specificato nulla nel suo costruttore.

SnackBar

Formato da un messaggio con un'azione opzionale visualizzato brevemente nella parte inferiore dello schermo, viene utilizzato per comunicare all'utente una particolare informazione come l'attesa per un caricamento o il manifestarsi di un problema a seguito di un errore, come la mancanza di una connessione internet.

FlatButton e IconButton

Sono entrambi pulsanti che si differenziano per ciò che contengono. Il primo è solitamente formato da un rettangolo (i cui angoli possono essere smussati a piacere) al cui interno risiede un testo esplicativo dell'azione che verrà eseguita a seguito di un tocco da parte dell'utente. Il secondo è invece circolare e contiene un'icona che dovrebbe mostrare direttamente quale sarà la sua funzione. Entrambi sono widget di tipo Statefull, in quanto il loro colore o la loro forma può cambiare a seguito di pressioni sullo schermo.

TextField

Consente all'utente di inserire del testo, attraverso l'uso della tastiera che appare sullo schermo. Il widget chiama la funzione *onChanged* ogni volta che viene modificato il testo nel campo. Se l'utente indica di aver finito di digitare (ad esempio premendo il pulsante di invio), si attiva la funzione *onSubmitted*. Per controllare il testo visualizzato nel TextField si può utilizzare una particolare classe chiamata *TextEditingController*, ad esempio per impostare un valore iniziale già presen-

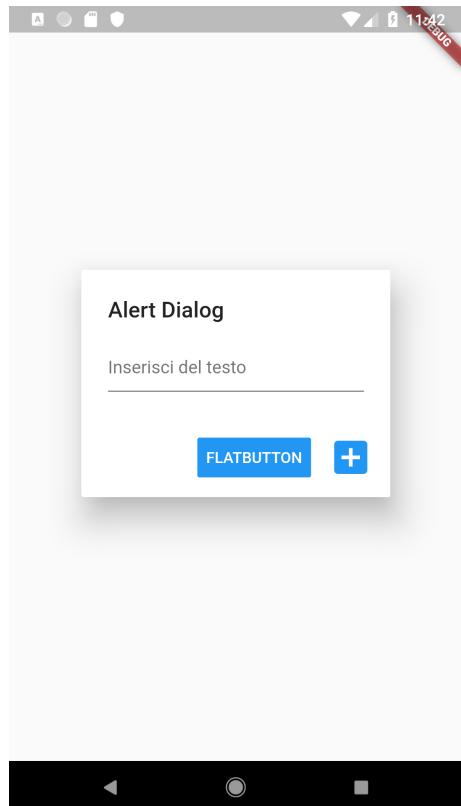


Figura 2.5: Nell'esempio viene mostrata una Scaffold che contiene un'AlertDialog. Quest'ultima presenta un titolo, un contenuto formato da una TextFormField per inserire del testo, e due pulsanti, quello a sinistra è un FlatButton mentre quello a destra è un IconButton.

te all'avvio della schermata o per inserire i caratteri scritti dall'utente in un'altra variabile.

AlertDialog

Una AlertDialog informa l'utente sulle situazioni che richiedono una conferma di un'operazione. Inoltre ha un titolo opzionale e un elenco opzionale di azioni. Il titolo viene visualizzato in alto e le azioni vengono visualizzate sotto il contenuto dell'AlertDialog. Si può anche scegliere che se l'utente tocca al di fuori dell'area dedicata a questo widget esso sparisca, oppure si può decidere che l'AlertDialog si dissolva solo a seguito della pressione di un particolare tasto.

BottomSheet

Il BottomSheet è un particolare widget che appare nella parte inferiore dello schermo e occupa spazio in base alla grandezza del proprio figlio (solitamente una Column o una ListView). Esistono due tipologie: fisso e dinamico. La prima rimane visibile all'avvio della schermata o per inserire i caratteri scritti dall'utente in un'altra variabile.

bile anche quando l’utente interagisce con altre parti dell’app. Può essere creato e visualizzato con la funzione *ScaffoldState.showBottomSheet*. La seconda tipologia è un’alternativa a un menù o a una AlertDialog e impedisce all’utente di interagire con il resto dell’app. I BottomSheet di questo tipo possono essere creati e visualizzati con la funzione *showModalBottomSheet*.

2.4 Animation

Le Animation rendono l’interfaccia più intuitiva, contribuiscono a rendere elegante l’aspetto di un’app e migliorano l’esperienza dell’utente. Il supporto di Flutter semplifica l’implementazione di una grande varietà di tipi di animazioni. Molti widget, in particolare i Material Widget, hanno animazioni standard definite nelle loro specifiche di progettazione, anche se il programmatore può crearne di nuove personalizzando quelle già esistenti.

Sono implementati due tipi di animazioni: Tween e Physic-based. Nella prima tipologia (che è l’abbreviazione di *in-betweening*, traducibile come ”nel frattempo, in mezzo”), vengono definiti l’aspetto e la forma iniziali e finali del widget, insieme alla durata e alla velocità dell’animazione. Il framework calcola poi automaticamente come effettuare la transizione in base ai dati che vengono forniti al costruttore della particolare Animation. La seconda tipologia cerca invece di gestire quegli elementi che vogliono rappresentare comportamenti del mondo reale, cercando di modellare i movimenti e le dinamiche di oggetti concreti. Per esempio per muovere una piccola sfera la velocità con cui essa rimbalzerà dipenderà dalla spinta iniziale che le è stata posta mentre l’altezza del salto dipenderà invece da quanto essa sia lontana dal terreno. Nello specifico esiste una classe chiamata AnimationController che possiede il metodo *animateWith* e inoltre è possibile simulare il comportamento di una molla con la classe SpringSimulator.

3 Gestione degli utenti

Per gestire al meglio la procedura di autenticazione dell'applicazione, la registrazione e le preferenze dei singoli utenti, si è scelto di utilizzare Firebase, servizio di database basato sul cloud appartenente a Google.

3.1 Firebase

Firebase è un ottimo servizio di backend che rende disponibili, tramite API, numerosi servizi tra cui lo storage dei dati, l'autenticazione, notifiche push e molto altro. Una delle caratteristiche più utili di questo servizio è la capacità di sincronizzazione dei dati oltre che di storage: le informazioni vengono infatti aggiornate pressochè all'istante, a patto che l'app web o mobile sia collegata alla rete. Inoltre sono disponibili numerose librerie client che rendono ancora più semplice l'integrazione con il proprio prodotto software. Anche la sicurezza ha un aspetto rilevante, in quanto i dati immagazzinati sono replicati e sottoposti continuamente a backup e la comunicazione con il client avviene in modalità crittografata tramite SSL con certificati a 2048 bit. Esistono tre piani tariffari per utilizzare Firebase. Il primo, gratuito, è chiamato Spark e comprende i servizi base, tra cui l'autenticazione, fino a 100 connessioni simultanee, 1 GB totale di spazio per i dati e 5 GB per lo storage di immagini e file. Il secondo è il Flame Plan nel quale aumentano i limiti (50 GB per lo storage e fino a 2,5 GB per i dati) al costo di 25\$ al mese. L'ultimo è il Blaze Plan nel quale ogni servizio possiede un costo unitario relativamente basso e funziona con una politica *pay as you go*. Per le esigenze del progetto il primo piano (Spark) era più che sufficiente.

3.2 Schermata di avvio, tipologia di Utente e pagina Profilo

Nella schermata d'avvio (figura 3.1) l'utente può scegliere di accedere all'applicazione in tre modi diversi: tramite accesso standard (utente standard), tramite il pulsante *Accedi con Google* (utente Google) o tramite accesso anonimo. A seconda della tipologia, le pagine accessibili cambiano, così come i servizi disponibili. Se l'utente è già registrato in modo standard, può inserire la propria mail (usata come

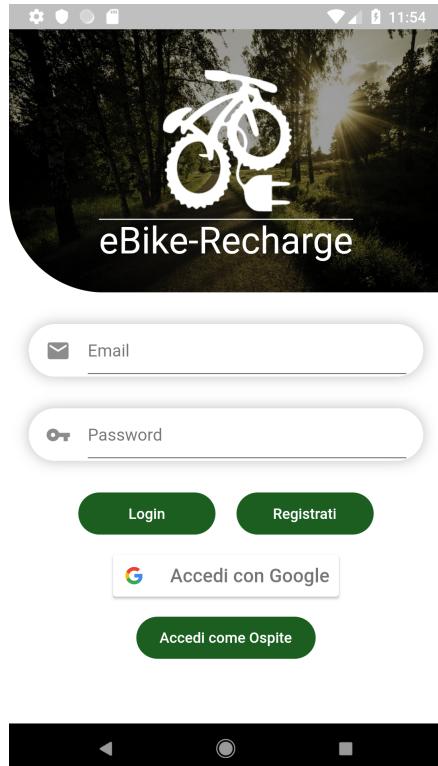


Figura 3.1: Schermata d'avvio

username) e la propria password. Se i dati forniti combaciano con quelli presenti nel database degli utenti si fa accesso alla pagina della mappa che verrà descritta in un prossimo capitolo. Se invece non c’è corrispondenza con un utente già registrato viene mostrato a video un messaggio di errore che spiega cosa è successo.

Si può anche fare accesso all’applicazione utilizzando i servizi Google che permettono di autenticarsi mediante gmail, con il proprio account Google. In questo caso si apre una nuova finestra (codificata grazie ad una API) che mostra gli account gmail utilizzati sul proprio dispositivo e dopo aver inserito la password si fa accesso alla pagina della mappa. In particolare la grande differenza tra i due tipi di accesso appena citati consiste nella pagina profilo utente presente nell’applicazione. Questa è accessibile premendo l’icona *Profilo* presente nella TabBar in basso sullo schermo. In alto viene indicata la mail dell’utente (sia che essa faccia parte di gmail, sia che appartenga ad un altro dominio), e vengono poi mostrati diversi pulsanti ognuno con una specifica funzione. Quelli con scritto ”Tipo Mappa”, ”Stazioni Aggiunte” e ”Log Out” sono presenti in entrambe le versioni e permettono rispettivamente di cambiare il tipo mappa ed esaminare le stazioni aggiunte dall’utente che sta usando

l'applicazione (entrambe queste funzionalità sono discusse nel capitolo riguardante la mappa), e di uscire dal particolare account in uso e tornare alla pagina di Login. Se l'utente è di tipo standard, è presente un ulteriore pulsante con scritto "Cambia Password": esso permette grazie a un'API di Firebase di cambiare la propria password nel caso in cui si sia dimenticata.

3.3 Registrazione

Nel caso in cui l'utente effettui il primo accesso oppure desideri creare un nuovo utente, deve solo premere il pulsante "Registrati", per andare nella pagina di registrazione. Essa mostra tre campi di testo dove bisogna inserire la propria mail, la password, e confermare quest'ultima reinserendo la stessa password. Vengono effettuati dei controlli di integrità dei dati (così come nella pagina di login): infatti tramite una *regular expression* si controlla che la mail inserita possa esistere, cioè presenti almeno una lettera seguita da una @, almeno due ulteriori lettere, un punto e almeno un'altra lettera. Inoltre la password deve possedere almeno sei caratteri e la sua conferma deve ovviamente combaciare con quella inserita prima. Se tutto è corretto l'utente può premere il tasto "Registrati", venendo quindi portato alla pagina della mappa e creando così un nuovo account standard.

Se l'utente non effettua il logout nella pagina Profilo, ogni volta che accede all'applicazione sarà subito portato alla pagina della mappa senza dover passare dalla pagina di login, questo accade perché grazie a Firebase si tiene traccia dello stato della sessione del particolare account.

3.4 Utente anonimo

Nella pagina di login è anche possibile premere il tasto "Accedi come Ospite" in fondo allo schermo, e questo pulsante permette di accedere all'applicazione senza dover registrare nessun account. La struttura dell'app cambia radicalmente se si esegue questo tipo di accesso. Innanzitutto sono presenti solo due pagine accessibili (e non più tre come nel caso standard e Google) e consistono in una pagina mappa con funzionalità estremamente ridotte e una pagina di Upgrade nella quale si mostra

Identificatore	Provider	Data creazione ↑	Accesso eseguito	UID utente
test@test.com	✉	24 gen 2019	25 feb 2019	T1u6TGgz84XZeBGvwozmEOKRz...
t@t.com	✉	28 gen 2019	24 apr 2019	8sCwgJRAObfwVSGYHDknkuAE0...
yous@gmail.com	✉	4 feb 2019	13 feb 2019	xYnGKWCvIAg5gA9yySoP4NPeo0...
zraiba@zraiba.it	✉	13 feb 2019	13 feb 2019	xTV4CR51pjfqJL7AxZXBjuUK883
(anonimo)	👤	26 feb 2019	26 feb 2019	Riy78E5g6jZ4kmo3RlinCrZyzy33
(anonimo)	👤	6 mar 2019	6 mar 2019	UJ8oAnJS3jOtMq0oiDZEPTggwM...
gvigan97@gmail.com	Google	9 mar 2019	20 mar 2019	Qvv4dBOEkBctv57KJEcY34QW0U...

Figura 3.2: La sezione Authentication di Firebase

all’utente i possibili privilegi raggiungibili al momento di un’autenticazione registrata. Nella sezione *Authentication* del servizio online di Firebase (figura 3.2) è possibile prendere visione di tutti gli utenti che hanno fatto accesso all’applicazione almeno una volta indicando la data di ultima visita, la data di creazione e l’id dell’utente. Inoltre ne viene indicato anche il tipo, sia esso standard, Google o un utente anonimo.

3.5 Codifica

4 Gestione e struttura del database

4.1 Progettazione iniziale

Una delle prime attività del progetto è stata la progettazione della base di dati, e fin dall’inizio era chiaro che non si sarebbe trattato di uno schema molto complesso. Infatti con sole tre tabelle si riesce a descrivere in modo appropriato l’ambiente nel quale risiede il progetto.

4.1.1 Schema Concettuale

La tabella principale e da cui si è partiti è quella delle stazioni, che presenta una chiave primaria (Key), un nome con cui è conosciuta, l’indirizzo cioè il nome della via nella quale risiede, la posizione, formata da due coordinate geografiche di latitudine e longitudine che sono realizzate mediante due variabili double, e può ma non è costretta a possedere una descrizione formata da un testo. La stazione possiede poi una Tipologia. Tramite colloquio con il commitente dott. Marco Aceti si è appreso come le stazioni di ricarica possono essere raggruppate in diversi insiemi dovuti al servizio vero e proprio che viene fornito oltre a quello della ricarica. Esistono stazioni nelle quali è possibile noleggiare bici elettriche e altre dove si può portare la propria bici per effettuare della manutenzione. Dunque la tabella TipoStazione contiene per ogni riga una diversa tipologia di stazione, formata dalla combinazione dei diversi servizi appena descritti, in quanto ovviamente esistono negozi che propongono più servizi contemporaneamente. Ogni istanza di questa tabella oltre a possedere una chiave che la identifica, è caratterizzata dal nome (cioè la tipologia) e da un numero intero chiamato Colore che viene utilizzato all’interno dell’applicazione per fornire un colore diverso a ogni tipologia in modo da rendere più semplice agli utenti l’identificazione di ciò che può interessare loro. Nello schema deve poi essere presente una tabella che tenga traccia degli utenti. Si presti attenzione che questa tabella è diversa da quella gestita da Firebase relativa agli accessi e alle tipologie di utenti descritta nel precedente capitolo: qui vengono presi in considerazione il nome dell’utente e oltre alla chiave per identifierlo la propria preferenza sulla tipologia di mappa scelta. Non si esclude che in futuro si possano aggiungere ulteriori campi

relativi a nuove preferenze che potrebbero essere scelte. Nella figura 4.1 è mostrato lo schema Entità-Relazione del progetto. Si noti come una stazione debba per forza appartenere alla relazione Inserimento in quanto deve essere stata inserita da qualcuno, ma un utente può non appartenere a tale relazione in quanto può tranquillamente utilizzare l'app per conoscere la posizioni di stazioni senza mai inserirne nessuna.

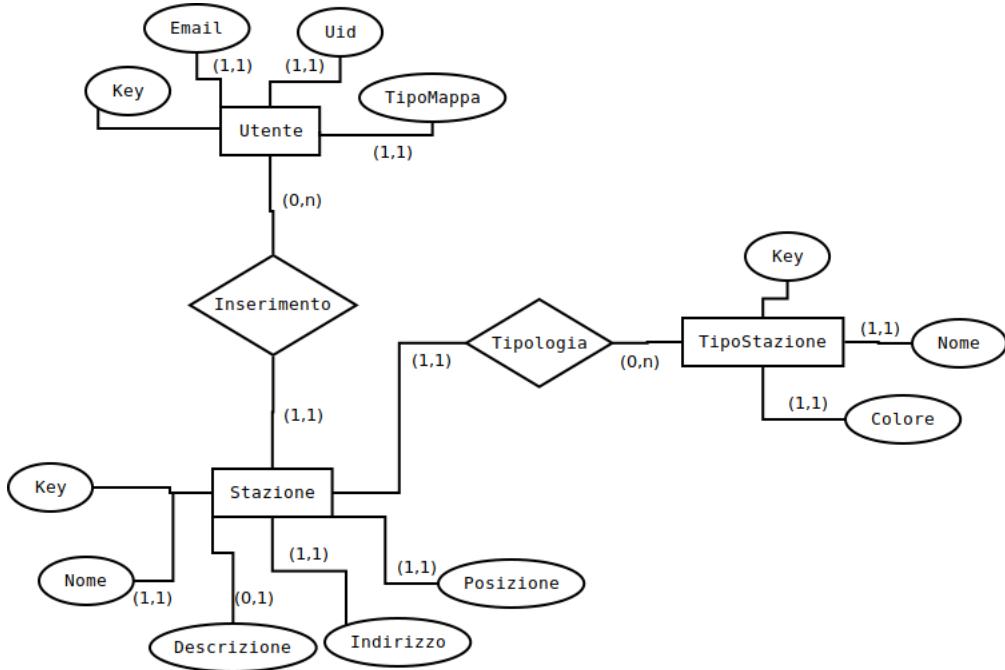


Figura 4.1: Schema Concettuale del progetto

4.1.2 Schema Logico

Data la semplicità dello schema Entità-Relazione la fase di progettazione logica è stata anch'essa priva di ostacoli. Si è posta particolare attenzione alle relazioni "Inserimento" e "Tipologia" perché sarebbero potute diventare delle tabelle ma dato che entrambe sono del tipo (1,1) verso (0,n) si è scelto di renderle degli attributi della tabella Stazione, diventando quindi chiavi esterne (figura 4.2).

4.2 Gestione dati in Firebase

Quando si crea un progetto nella propria Firebase Console una delle prime importanti operazioni è quella di scegliere la tipologia di database. Il servizio Google

STAZIONE(Key, Nome, Descrizione, Indirizzo, Posizione, TipologiaStazione, Utente)

UTENTE(Key, Uid, Email, TipoMappa)

TIPOSTAZIONE(Key, Nome, Colore)

Figura 4.2: Schema Logico del progetto

infatti offre due possibili alternative: il Realtime Database oppure il Cloud Firestore. Fino alla fine del 2018 la prima opzione era anche l'unica disponibile e consiste in una soluzione efficiente, a bassa latenza per applicazioni mobili che richiedono la sincronizzazione di stato tra client diversi in un tempo paragonabile alle prestazioni di un sistema real time. Il secondo, disponibile da minor tempo, è stato sviluppato per migliorare ulteriormente il suo predecessore, cambiando anche il modo in cui i dati vengono organizzati all'interno dello storage. Entrambe queste soluzioni sono database costruiti per tecnologie NoSQL, dunque non sfruttano le classiche tabelle dove ogni riga di una tabella è un record che contiene sempre lo stesso numero e tipologia di attributi. La tipologia Realtime immagazzina i dati in un unico grande albero JSON. Questo comporta una relativa facilità nell'inserimento dati soprattutto se di ridotte dimensioni, ma nel momento in cui la grandezza e complessità del database non sono più banali diventa difficile organizzare una struttura gerarchica. La tipologia Cloud immagazzina invece i dati in documenti chiamati *collections*, ne segue quindi che essa fa parte delle cosiddette basi di dati orientate al documento. Strutture dati modeste sono molto facili da inserire (in modo simile al metodo visto prima tramite JSON) e se queste diventano complesse e di grandi dimensioni usando collezioni e sotto-collezioni si può facilmente organizzare il tutto senza perdere in efficienza. Considerando invece le funzionalità relative all'effettuare query, per il Realtime Database è possibile filtrare o ordinare dati in una sola query, mentre per il Cloud Firestore è possibile effettuare entrambe le operazioni nella stessa query. Inoltre nella prima tipologia il risultato è sempre un albero JSON che contiene tutti i dati, dunque le dimensioni possono assumere una certa importanza, mentre nella seconda le performance della query sono proporzionali alla grandezza del risultato e non dell'intera struttura. A seguito di queste considerazioni si è quindi deciso,

benchè il progetto non presenti strutture particolarmente complesse, di utilizzare Cloud Firestore.

4.3 Utilizzare il Cloud Firestore

5 Mappa e sue funzionalità

La pagina della mappa è stata la sezione dell'applicazione più difficile da implementare, perchè al suo interno risiedono numerosi servizi e funzioni che hanno richiesto diverso tempo per essere sviluppati e inoltre è stato necessario documentarsi per capire come utilizzare le API che la pagina utilizza.

5.1 Scelta del servizio API

In un primo momento si è pensato di utilizzare il servizio mappe di Google Maps che, oltre ad essere uno dei più efficienti e meglio documentati, è anche implementabile facilmente. Il problema è nato nel momento in cui Google ha deciso di cambiare le proprie politiche di utilizzo delle API verso la fine del 2018. Prima di quel momento, sotto a un certo numero di richieste al servizio di geolocalizzazione il programmatore poteva usare liberamente il codice e senza necessità di registrazione, ma in seguito l'azienda di Mountain View ha deciso che chiunque volesse utilizzare il proprio servizio mappe dovesse prima registrare un proprio account ed inserire una carta di credito che eventualmente pagasse mensilmente le risorse di cui si è fatto uso. Questo scenario ha portato a prendere in cosiderazione altri gestori di mappe. La scelta è ricaduta su MapBox, azienda emergente nel proprio campo. Il servizio era gratuito e permetteva un agevole utilizzo senza registrazione ma il problema era formato dall'implementazione vera e propria. Al contrario di Google, non esiste un pacchetto software che implementi il codice MapBox e quindi era necessario fare uso di richieste http tramite valori codificati all'interno di lunghi url. Inoltre la fluidità della mappa non rispettava le direttive del committente dott. Marco Aceti, spesso a seguito di un rapido movimento delle dita per spostarsi in un'altra zona della mappa la schermata rimaneva per qualche secondo completamente grigia, rendendo l'esperienza di utilizzo sicuramente peggiore. Nel seguito, tra Gennaio e Febbraio 2019 è stata rilasciata la prima versione ufficiale di Flutter (versione 1.0.0) e tra le tante novità spiccava la presenza di un widget particolare, chiamato GoogleMap. Semplificando notevolmente l'utilizzo delle mappe, tale widget presenta ottime prestazioni e facilità di implementazione. Si è quindi deciso di dedicare del tempo nell'appren-

dere ogni aspetto delle politiche di utilizzo delle API di Google, capendo quindi che, facendo un numero di richieste minore di una soglia stabilita, non è necessario pagare nulla, anche se si è registrata una carta di credito. Quindi la scelta è ricaduta sulle API di Google e si è importato nel progetto il pacchetto `googlemap`.

5.2 MapPage

Dopo che l'utente ha eseguito correttamente la procedura di login o ha aperto l'applicazione in caso di autoidentificazione con mantenimento dello stato della registrazione, viene mostrata la MapPage. In alto è presente un'AppBar che mostra il titolo `eBike-Recharge`, in basso è presente una TabBar che permette di navigare tra le pagine dell'app. Tra le due, a tutto schermo, viene mostrata la mappa con al centro la posizione attuale dell'utente. Quest'ultima può essere di diversi tipi a seconda delle preferenze indicate nella pagina profilo dell'utilizzatore. In particolare si può scegliere tra *Normale*, cioè la classica modalità di visualizzazione di mappe che viene mostrata di default sull'app Google Maps, *Rilievo*, dove è presente l'altitudine delle montagne indicata con diversi colori in base all'altezza, mancando però di indicare i nomi di edifici importanti, negozi o imprese, *Satellite*, che mostra con una serie di foto satellitari i luoghi indicati dall'alto, in modo da vedere realmente il percorso che si deve intraprendere per raggiungere la propria destinazione, e *Irida*, che può essere considerata l'unione di Satellitare e Normale, poichè oltre a presentare l'altitudine mostra anche i nomi dei luoghi di maggior importanza della zona visualizzata.

Grazie al pacchetto `googlemap` di flutter si è riusciti a implementare un oggetto `GoogleMap` (che verrà descritto in un successivo paragrafo) con cui è molto facile interagire. In particolare, l'utente può aumentare o diminuire lo zoom della mappa con i classici gesti di allontanamento o avvicinamento delle dita, e può spostarsi in diverse zone semplicemente trascinando verso la direzione desiderata. Con una buona o anche solo media connessione internet i tempi di latenza sono pressochè nulli, così che non si riesca a vedere i riquadri grigi che il widget crea nel momento in cui si cambia direzione e che dovranno essere riempiti con le nuove località. Questi ultimi

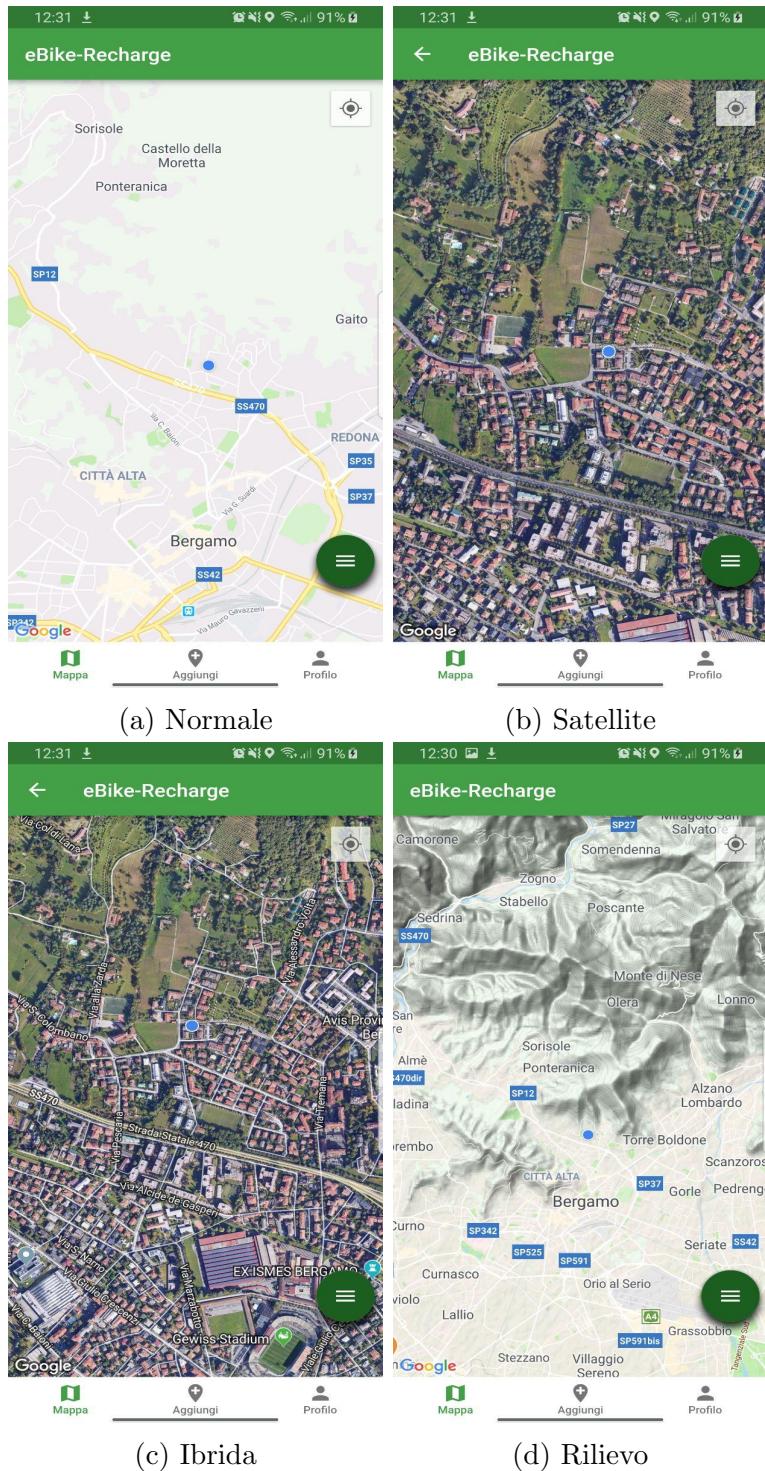


Figura 5.1: I quattro tipi di mappa selezionabili nella pagina profilo

sono pienamente visibili nel momento in cui non si disponga di una connessione adeguata, condizione che si verifica sempre più di rado negli ultimi tempi.

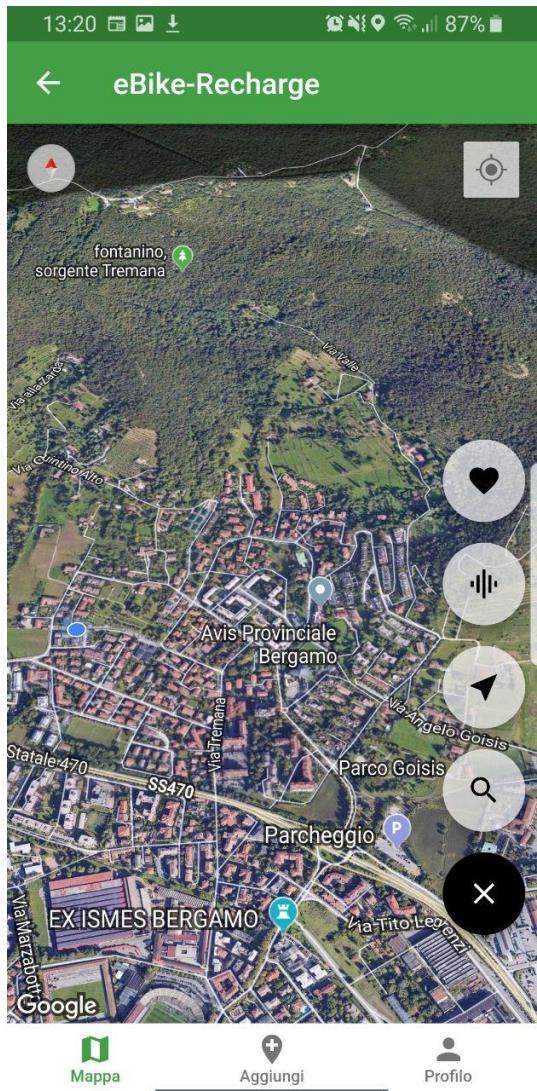


Figura 5.2: Premuto il pulsante, vengono mostrati i 4 FloatingActionButton e il quinto (con la X) per tornare indietro

5.3 Funzionalità

Nelle immagini precedenti si è forse notato un pulsante nella parte inferiore dello schermo verso destra. Se toccato (figura 5.2), quest'ultimo mostra una serie di pulsanti con diverse funzionalità. Il primo è un'icona a forma di cuore e, interagendo con esso, viene mostrato un BottomSheet (spiegato nel secondo capitolo, Flutter) che mostra una Card (una piccola scheda con titolo e icona) con le stazioni di ricarica preferite dall'utente. Per esprimere la propria preferenza per una stazione è necessario entrare nella scheda di quest'ultima e selezionare il cuore presente in alto a destra.

Il secondo pulsante ha la funzione di filtro e al tempo stesso di legenda. Premendolo, viene mostrata a schermo una AlertDialog (fig. 5.3b) che indica quali tipologie di stazioni si desidera cercare. Sulla destra sono presenti dei Button di tipo check (che possono essere True o False): se si vede la casella spuntata allora si vedrà la relativa tipologia sulla mappa, se tale casella è vuota allora la mappa verrà filtrata e non sarà possibile cercare tutte le stazioni di quel tipo.

Il terzo pulsante mostra un'icona con la classica freccia indicante la navigazione e, se premuto, fa apparire un BottomSheet con Card contenente le dieci stazioni più vicine alla posizione attuale dell'utente. Come per tutte le Card anche nelle altre funzionalità, è possibile definire un'azione in seguito al tocco della scheda. In seguito all'interazione dell'utilizzatore con una specifica Card, la mappa ruota e cambia il proprio centro mostrando l'icona indicante quella stazione che si è toccata. In questo modo l'utente risparmia tempo poiché viene direttamente condotto alla stazione di interesse.

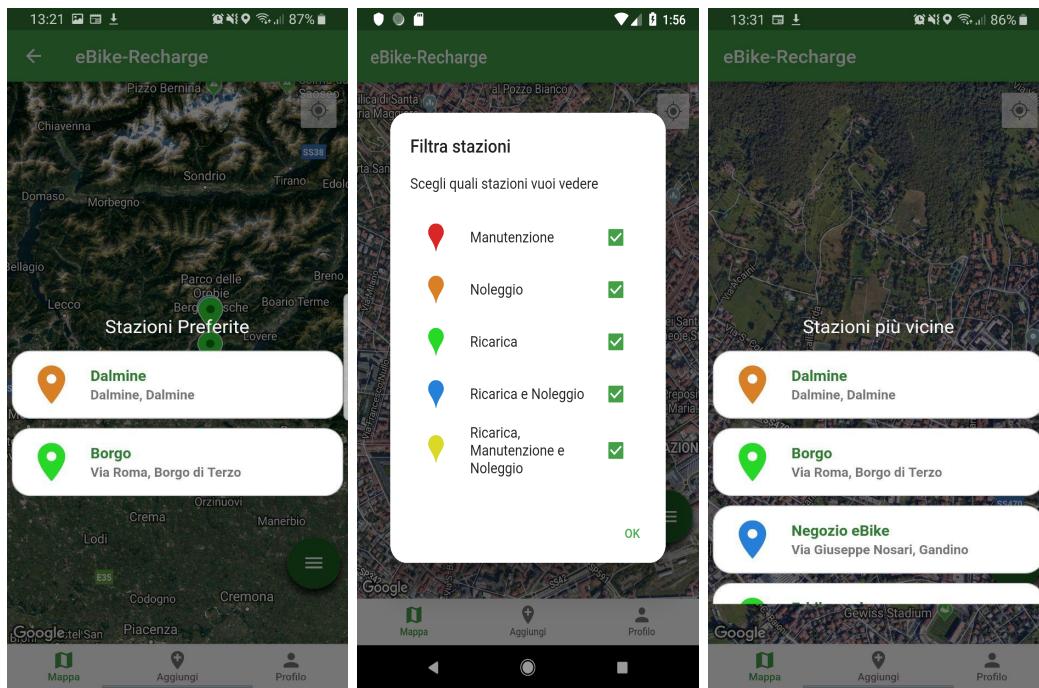


Figura 5.3

L'ultimo pulsante è accompagnato da un'icona rappresentante una lente di ingrandimento, simbolo universale di ricerca. Premendolo, in alto sullo schermo appare

un TextField con impresso la frase "Cerca Indirizzo". L'utente è quindi invitato a digitare la via o la località che desidera vedere. Mentre si forma la scritta, grazie al servizio API Google Places l'app è in grado di mostrare diverse Card a schermo con suggerimenti, fungendo quindi da auto-completamento. Se l'utente clicca su una di queste sezioni viene richiamata la stessa funzione delle Card prima mostrata: la mappa cambia centro e porta l'utente nella via desiderata. Questo servizio costa una frazione di centesimo per ogni singola chiamata alla funzione di auto-completamento. Si è quindi deciso di introdurre un limite mensile al numero di volte che l'app può far uso di questa API, in corrispondenza del numero massimo di chiamate che si possono fare senza dover pagare direttamente Google.

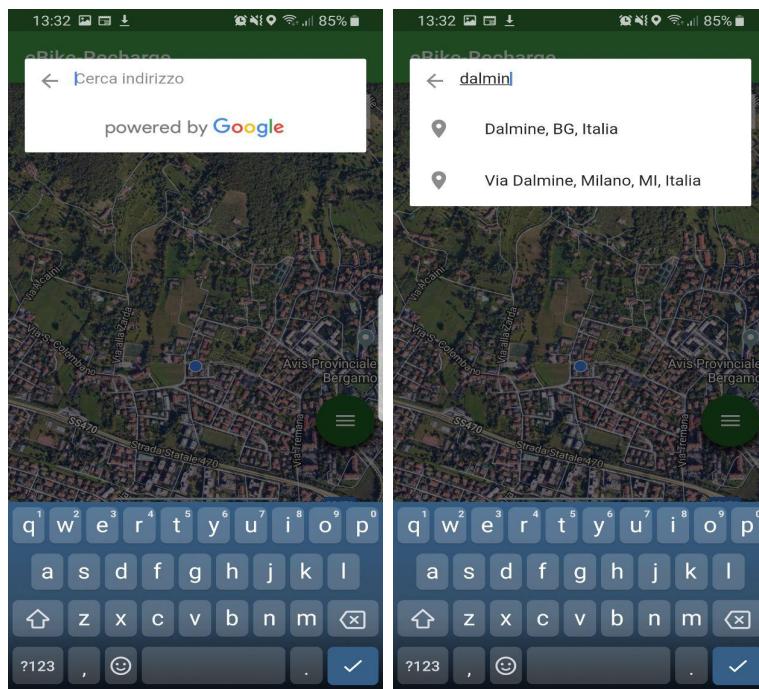


Figura 5.4: La funzione di ricerca e autocompletamento dell'API Google Places

Riferimenti bibliografici

[Lombardo(2018)] Gianni Lombardo. Ecco i dati ufficiali ancma 2017: l'e-bike vola anche in italia e segna un ulteriore +19%, 2018. URL <https://www.bicitech.it>.

[material.io(2019)] material.io. Material design, 2019. URL <https://material.io/design/introduction>.