

SkiTracker

Progetto di Programmazione Mobile



Federico Arduini - Omar Naja
Anno Accademico 2022 - 2023

Indice

1	Introduzione	2
1.1	Approccio di sviluppo	3
2	Sviluppo con Android Studio	5
2.1	Requisiti funzionali e non funzionali	6
2.2	Architettura dell'app	8
2.3	Database dell'app	9
2.4	Mockup della User Interface	11
2.5	Sviluppo dell'app	14
2.5.1	Creazione e impostazione del progetto	14
2.5.2	Tutorial iniziale dell'app	17
2.5.3	Selezione del comprensorio	18
2.5.4	Area principale dell'app	18
2.5.5	Tracciamento del percorso	24
2.5.6	Problematiche note e conclusioni	26
2.6	Testing delle funzionalità	27
3	Sviluppo con Flutter	30
3.1	Installazione di Flutter	31
3.2	Requisiti funzionali e non funzionali	32
3.3	Architettura dell'app	33
3.4	Database dell'app	35
3.5	Mockup della User Interface	35
3.6	Sviluppo dell'app	36
3.6.1	Definizione dell'area principale	36
3.6.2	Mappa del comprensorio	37
3.6.3	Informazioni sul comprensorio	38
3.6.4	Selezione del comprensorio	38
3.6.5	About Us	39
4	Conclusioni	40

1 Introduzione

Lo sci è uno degli sport invernali più praticati e amati al mondo. In particolar modo, l'Italia è una delle nazioni europee con il maggior numero di comprensori sciistici, mete predilette di milioni di turisti, sia italiani che europei. Gli sciatori più esperti, spesso inclini a cambiare e ad esplorare i vari comprensori che il nostro Paese offre, conoscono bene quanto sia importante avere sempre a portata di mano una mappa che permetta loro di potersi orientare (per scongiurare smarrimenti e perdite d'orientamento) e di scoprire tutte le piste e gli impianti di risalita (seggiovie, skilift, funivie, etc) di cui il comprensorio che gli interessa è dotato.

Con l'avvento della tecnologia mobile e degli smartphone, sono nate nel corso degli anni svariate applicazioni dedicate al mondo sciistico, tutte quante con scopi più o meno differenti: si spazia da applicazioni che si occupano di cronometrare le proprie sciate, a quelle che permettono di segnalare imprevisti e situazioni di emergenza; altre permettono di geolocalizzare l'utente nel comprensorio, per facilitarne l'orientamento, altre ancora di fornire le piste che questo comprende.

Una delle più grandi criticità sentite dagli sciatori, durante le loro attività invernali, è l'uso di una mappa di tipo cartaceo, con tutte le scomodità che si porta appresso. Innanzitutto, le mappe tradizionali sono grandi e generalmente poco pratiche: sono composte da un unico foglio molto grande, che viene ripiegato tante volte; una soluzione certamente poco pratica, sia per la scocciatura di ripiegare il foglio, sia per il fatto che questo sarebbe facilmente soggetto ai venti e alle condizioni avverse che si possono verificare in quota, rendendo la consultazione della mappa difficoltosa e in casi estremi illeggibile. Inoltre, non è naturalmente possibile usufruire di funzionalità avanzate e di grande comodità, come la geolocalizzazione sulla mappa (e quindi poter conoscere la propria posizione precisa direttamente sulla stessa).

La nostra applicazione nasce per venire incontro a queste criticità, sentite personalmente dallo stesso referente di sviluppo del progetto. SkiTracker permette di fornire agli sciatori la **mappa del proprio comprensorio**, direttamente sul proprio cellulare: le mappe cartacee, spesso poco pratiche, vengono sostituite con una **mappa virtuale**, consultabile direttamente sul proprio telefonino in modo molto più pratico e immediato.

La nostra app supporta più di **80 comprensori in tutta Italia**, e per ognuno di questi permette di visualizzare tutte le **piste** di cui sono dotati assieme agli impianti di risalita previsti. Per aiutare l'utente a localizzare la sua posizione (utile, ad esempio, per scongiurare eventuali perdite dell'orientamento in pista), verrà visualizzata la **posizione in tempo reale** (ottenuta tramite GPS) dello sciatore nella mappa.

Inoltre, SkiTracker integra anche **funzioni di tracciamento**: lo sciatore potrà, una volta scelta la pista che intende percorrere, **tracciare la sua attività di sci** e poter così ottenere quanto tempo ha impiegato a percorrerla, a quale velocità (media) e a quale dislivello è stato

soggetto.

Per evitare di incombere in situazioni di manto nevoso estremamente degradato (dovuto allo scoglimento della neve per via delle alte temperature), situazione anomala sempre più comune negli ultimi anni e che può portare anche a pericolose cadute, SkiTracker integra anche **funzioni di segnalazione**: lo sciatore può segnalare a tutti gli altri utenti lo **stato del manto nevoso delle piste**, mettendo in guardia i futuri utenti da piste che sono danneggiate e quindi potenzialmente pericolose.

1.1 Approccio di sviluppo

L'applicazione verrà sviluppata impiegando due tecnologie differenti, e quindi due implementazioni ed approcci diversi.

- La prima viene sviluppata utilizzando **Android Studio**, l'Integrated Development Environment (IDE) ufficiale per sviluppare app Android, creato da JetBrains. Con questa implementazione, l'applicazione sarà **nativa**, in modo che sia completamente funzionante unicamente per il sistema operativo Android, in particolare a partire dalla versione 8.0 (Android Oreo). Il codice verrà scritto in **Kotlin**, un linguaggio di programmazione ispirato a Java e Python, nato nel 2011 e sviluppato da JetBrains, divenuto il linguaggio consigliato per lo sviluppo di applicazioni Android sin dal 2019;
- la seconda verrà sviluppata impiegando **Flutter**, un framework open-source creato da Google, che permette di sviluppare un'**applicazione cross-platform**, funzionante quindi sia su Android che su iOS e tutti gli altri sistemi operativi meno conosciuti. Per questa seconda implementazione, l'applicazione conterrà un set ridotto delle funzionalità (rispetto a quelle previste nell'implementazione con Android Studio), e non sarà previsto il testing in fase di sviluppo. Si farà uso del linguaggio di programmazione **Dart**.

Per entrambe le implementazioni, comunque, si partirà dalla **progettazione teorica** dell'applicazione, a partire dallo studio e dall'**analisi dei requisiti**, funzionali e non funzionali, fino ad arrivare alla stesura dei diagrammi per le illustrazioni dell'**architettura dell'app** e dei **database** che verranno impiegati (corredati di descrizione delle tecnologie che sono state impiegate per implementarli); infine, i **mockup dell'interfaccia** forniranno un'anteprima dell'interfaccia grafica che l'applicazione adotterà una volta che lo sviluppo sarà terminato.

È importante precisare che i mockup sono solamente un'anteprima, l'applicazione potrebbe non rispecchiare appieno il layout grafico che viene rappresentato dai mockup. L'applicazione potrebbe, inoltre, risultare incompleta: questo per via di alcuni bug non risolti, o semplicemente

per l'assenza di alcune funzionalità descritte in fase di raccolta ed analisi di requisiti e casi d'uso.

In ogni caso, eventuali situazioni di questo tipo verranno tutte riportate negli opportuni paragrafi e nelle opportune sezioni; sarà nostro impegno evitare queste situazioni spiacevoli e sviluppare un'applicazione completa e di qualità, e che possa fornire un prodotto concreto sia dal lato didattico e sia dal lato pratico.

2 Sviluppo con Android Studio

Lo sviluppo con Android Studio rappresenta la tematica principale in fase di sviluppo: verrà data molta più enfasi a questa implementazione del nostro progetto. Per lo sviluppo dell'app, è stato utilizzato il software AndroidStudio (la cui interfaccia è illustrata in Figura 1) e, come già anticipato, il linguaggio di programmazione utilizzato sarà Kotlin.

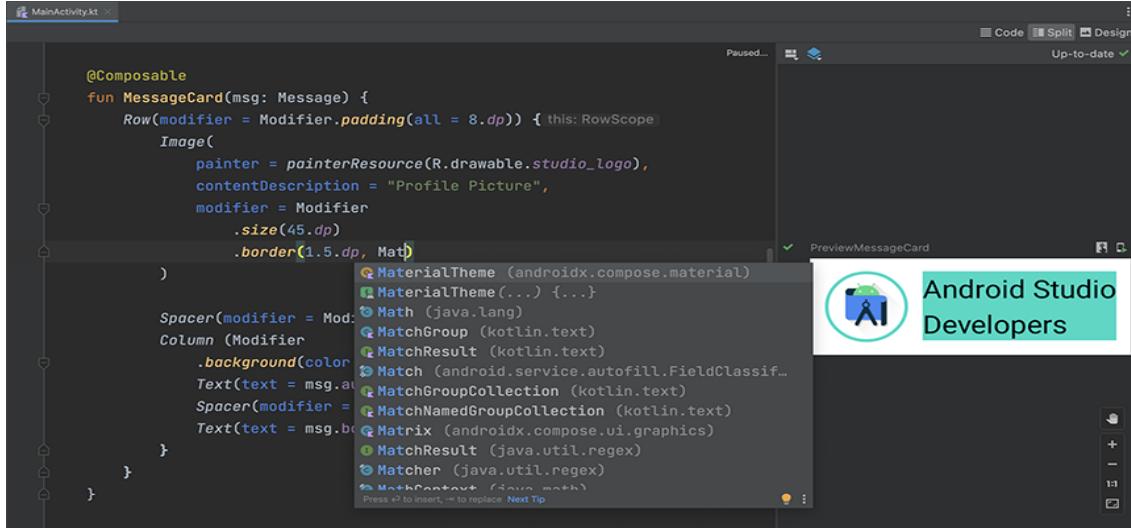


Figura 1: Interfaccia dell'IDE Android Studio.

Android Studio, realizzato da JetBrains per Google, è sin dalla prima release di Android l'**ambiente di sviluppo ufficiale**, impiegato per lo sviluppo di app **native**, quindi progettate e funzionanti esclusivamente per questo sistema operativo.

È un ambiente di sviluppo estremamente completo: ha un ottimo supporto sia per il linguaggio di programmazione Java (usato originariamente per questo scopo) e sia per il più recente Kotlin. Essendo Kotlin compilato ed eseguito sulla stessa virtual machine di Java, e quindi quasi completamente basato su quest'ultimo, Android Studio permette di convertire in tempo reale codice Java in codice Kotlin, semplicemente incollandolo nel file di codice; questo avviene in maniera totalmente automatica e senza perdita di funzionalità, come illustrato in Figura 2.

Include, inoltre, svariate facilitazioni per la gestione dello sviluppo di un'app nativa, come un editor grafico delle viste grafiche, un gestionale per la traduzione in varie lingue e delle risorse multimediali, e svariati *helper* per velocizzare e migliorare la scrittura e l'ordine del codice.

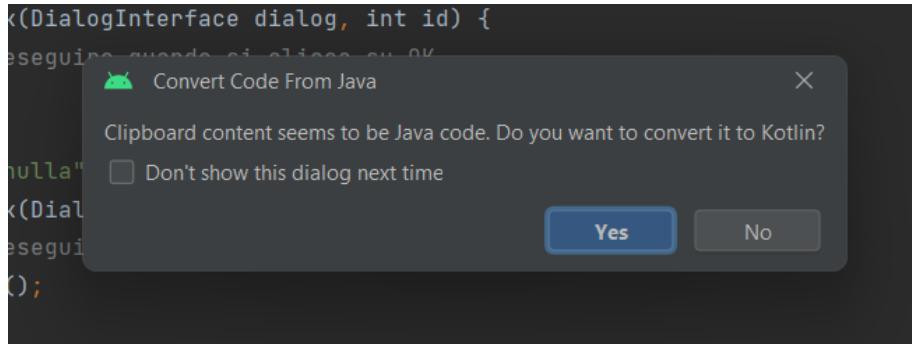


Figura 2: Conversione del codice Java in codice Kotlin tramite Android Studio.

Incluso in Android Studio, vi è anche l'[Android Virtual Device \(AVD\)](#), l'emulatore ufficiale di Google di dispositivi Android (cellulari e tablet), che verrà largamente impiegato in fase di sviluppo per controllare la funzionalità dell'app al manifestarsi delle più svariate condizioni: è possibile simulare, ad esempio, il comportamento dell'app in caso di connessione assente o batteria scarica, oppure simulare la posizione geografica e il movimento del dispositivo stesso.

2.1 Requisiti funzionali e non funzionali

I **requisiti funzionali** permettono di descrivere le funzionalità che il software offre, in relazione alle esigenze che deve soddisfare (ovvero, le richieste dei clienti che ne intendono fare uso); specificano quindi **cosa** il software deve **implementare**. I **requisiti non funzionali**, invece, descrivono le caratteristiche prestazionali e implementative delle funzionalità e delle componenti che verranno implementate dal software; in sostanza, si occupa di specificare il **modo** in cui le **funzionalità** verranno **implementate**.

Riguardo ai requisiti funzionali, quelli che abbiamo individuato per la nostra applicazione (e quindi le funzionalità di base che garantisce) sono i seguenti:

1. **Visualizzazione del tutorial iniziale:** l'app, quando verrà avviata per la prima volta, visualizzerà a schermo una spiegazione (tutorial) delle funzionalità di base offerte dall'app; il cliente sarà libero di saltarla, se lo ritiene non necessario.
2. **Selezione del comprensorio:** il cliente deve poter selezionare il comprensorio di cui intende ottenere le piste e la mappa. I tracciamenti delle attività sciistiche riguarderanno le piste del comprensorio selezionato.
3. **Visualizzazione della mappa del comprensorio scelto:** l'app visualizzerà una mappa che comprenderà tutte le piste del comprensorio scelto dal cliente.

4. **Selezione della pista:** il cliente potrà selezionare la pista di suo interesse per ottenere maggiori dettagli su di questa (nome della pista e difficoltà).
5. **Tracciamento del percorso lungo una pista da sci:** il cliente potrà usare la nostra app per tracciare la sua attività sciistica lungo la pista da sci da lui selezionata: l'applicazione, quindi, registrerà il percorso della sua discesa lungo la pista e altri dati: durata della discesa, velocità media, dislivello, lunghezza del tragitto percorso.
6. **Visualizzazione della cronologia dei percorsi tracciati:** l'app darà la possibilità di vedere la cronologia dei tracciamenti fatti.
7. **Visualizzazione delle informazioni del comprensorio:** dalla nostra app sarà possibile vedere tutte le informazioni del comprensorio sciistico selezionato dal cliente, ad esempio: numero di piste, numero impianti di risalita, massima e minima altitudine sul livello del mare, etc.
8. **Regolazione dello zoom della mappa:** per via di una limitazione delle API impiegate dall'app, in casi particolari il cliente potrebbe imbattersi nella situazione spiacevole in cui nella mappa non vengono visualizzate tutte le piste del comprensorio; è possibile risolvere questo problema regolando lo zoom della mappa ottenuta con le API, funzione che l'app renderà disponibile al cliente nel caso debba esserci necessità.
9. **Visualizzazione delle informazione di una pista sulla mappa:** il cliente potrà selezionare una pista sulla mappa per ottenerne il nome, la difficoltà e lo stato del manto nevoso di questa rilevato nelle ultime 24 ore.
10. **Geolocalizzazione dell'utente sulla mappa:** l'app consentirà la visualizzazione della posizione in tempo reale (ottenuta tramite GPS) del cliente lungo il comprensorio.
11. **Valutazione dell'agibilità della pista:** il cliente, una volta terminato il tracciamento lungo una pista, potrà segnalare lo stato del manto nevoso della pista appena tracciata.
12. **Visualizzazione dettagli di un tracciamento:** il cliente può visualizzare i dettagli di un tracciamento lungo una pista che ha svolto in passato.

Riguardo invece i requisiti non funzionali, sono stati individuati i seguenti:

1. **Scrittura del codice in linguaggio Kotlin:** il codice dell'applicazione verrà scritto in linguaggio Kotlin.
2. **Uso di API esterne:** l'app, per ottenere varie informazioni farà uso di tre API (Application Programming Interface):

- (a) [SkiMap](#): per l'ottenimento della lista dei comprensori e delle loro informazioni.
 - (b) [Nominatim](#): API che usa la tecnologia di [OpenStreetMap](#) per la geocodifica della posizione del comprensorio: fornisce l'area in cui il comprensorio è compreso.
 - (c) [Overpass API](#): API basata sulle tecnologie di OpenStreetMap, che ottiene l'area del comprensorio e la elabora, aggiungendole i punti di interesse, come rifugi, impianti di risalita e le piste.
3. **Salvataggio dati dell'app su un database interno**: i dati necessari al funzionamento dell'app (come il comprensorio selezionato e la cronologia dei tracciamenti) verranno salvati su un database interno all'app, implementato con [Android Room](#).
 4. **Salvataggio stati delle piste su database online**: i dati relativi allo stato delle piste verranno salvati su un database online, realizzato con [Firebase](#).

2.2 Architettura dell'app

Per quanto concerne l'architettura software che si è deciso di applicare, si è optato per il pattern **Model - View - ViewModel** (MVVM). Esso presenta uno schema simile al noto pattern MVC (Model - View- Controller), ma migliorato per garantire una maggiore efficienza per l'ambito mobile. A differenza del Controller, infatti, si impiega il **ViewModel**. Questi risalta l'interfaccia utente, in quanto si occupa di gestisce eventi, eseguire operazioni ed effettuare il *data binding*, che consente di associare e sincronizzare una fonte dati agli elementi dell'interfaccia utente.

Nel precedente MVC, i ruoli di intefaccia utente e di gestione degli eventi sono nettamente distinti, appunto, nella View e nel Controller rispettivamente.

Tornando al pattern MVVM, esso prevede appunto la struttura architettonale secondo tre grandi macrocategorie:

- **Model**: prevede l'implementazione, appunto, del modello di dominio (domain model) dell'applicazione. Include un modello di dati e le logiche di business e di validazione. Sostanzialmente, è la gestione dei dati e della logica dell'applicazione.
- **View**: è responsabile della definizione della struttura, del layout e della *User Interface* che l'utente vede.
- **ViewModel**: è un intermediario tra i due componenti precedentemente descritti; fornisce i dati alla vista e gestisce gli eventi generati dalla vista. Come si è già discusso in precedenza, questa componente fa sì che le interazioni con la View permettano anche la gestione dei dati.

In Figura 3 è illustrato lo schema di funzionamento di questo pattern.

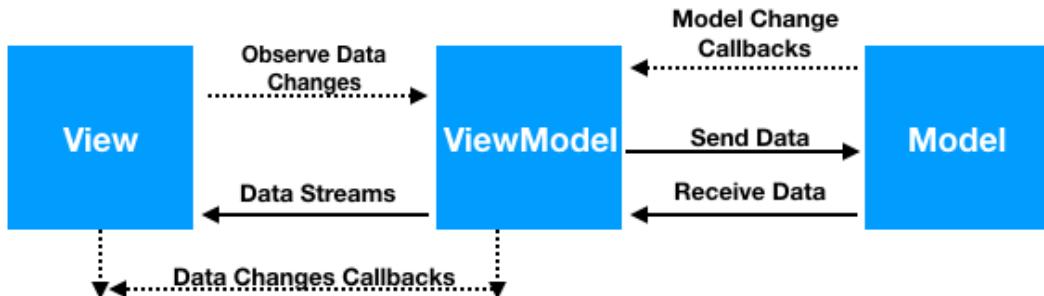


Figura 3: Schema di funzionamento del pattern MVVM.

L'uso del ViewModel, in panorama Android, diventa rilevante in quanto una caratteristica di Android è quella di ricreare un'Activity ogni qual volta che, ad esempio, si compie la rotazione del dispositivo, quindi quando vi è la rotazione dello schermo. Il ViewModel infatti, non è soggetto allo stesso ciclo di vita dell'Activity, ed è quindi capace di garantire persistenza dei dati dell'Activity anche quando questa viene ricreata.

2.3 Database dell'app

L'app fa uso principalmente di un database locale, creato con l'uso di **Android Room**, un componente che permette di facilitare l'uso di **SQLite** per lo sviluppo di applicazioni Android: questi permette di associare le tabelle del database a delle classi, fungendo quindi da strato di astrazione tra l'applicazione e il database SQLite.

Il database interno viene rappresentato tramite l'uso dello schema **Entity - Relationship**, visualizzato in Figura 4.

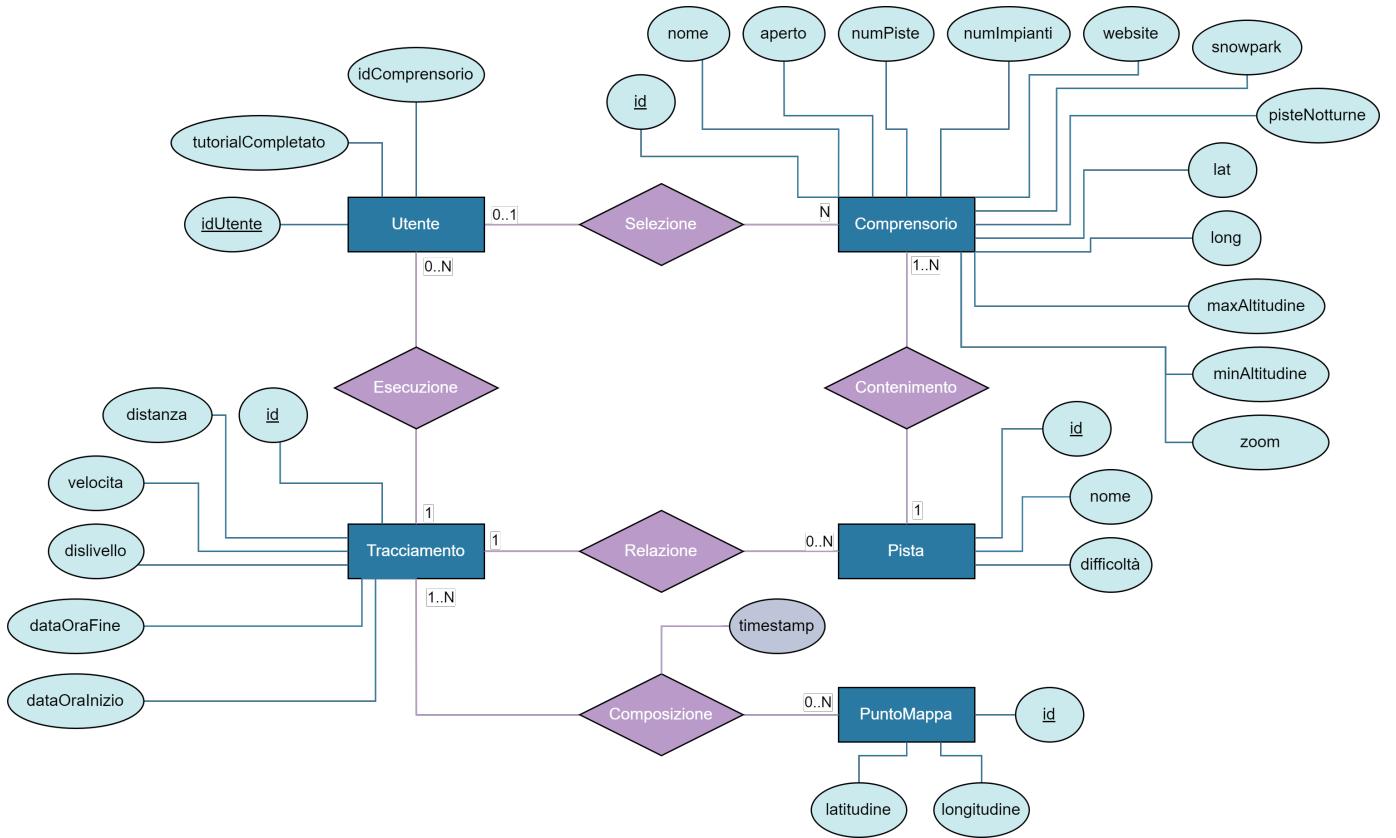
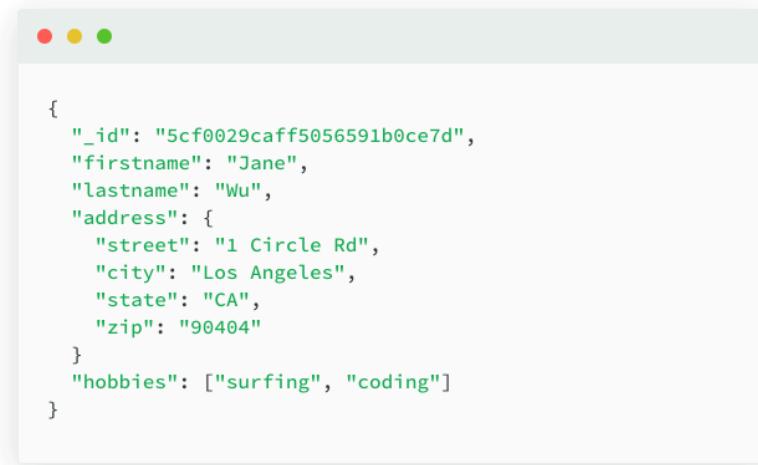


Figura 4: Schema Entity - Relationship del database dell'app.

Principalmente, è stato preferito l'uso di un database locale a uno online poichè l'ambito di utilizzo della nostra app prevederà molti scenari dove è prevista scarsità o totale assenza di connessione ad Internet: con l'uso di un database locale, una volta che l'utente avrà selezionato il comprensorio e che saranno stati scaricati tutti i dati di questo, questi resteranno disponibili anche senza connessione Internet.

L'app prevederà anche l'uso di un database online, impiegato per memorizzare lo stato del manto nevoso delle piste. Per realizzarlo, si impiegherà **Firebase**, una piattaforma di Google per l'hosting di basi di dati, organizzate secondo il paradigma **NoSQL**: la base di dati non sarà organizzata con un modello rigido come quello relazionale, ma avrà una struttura completamente libera e basata sull'uso di **JSON** (JavaScript Object Notation).

A screenshot of a terminal window showing a single JSON document. The document represents a user profile with fields like _id, firstname, lastname, address, and hobbies. The address field is nested within the user object.

```
{
  "_id": "5cf0029caff5056591b0ce7d",
  "firstname": "Jane",
  "lastname": "Wu",
  "address": {
    "street": "1 Circle Rd",
    "city": "Los Angeles",
    "state": "CA",
    "zip": "90404"
  },
  "hobbies": ["surfing", "coding"]
}
```

Figura 5: Esempio di rappresentazione dei dati in un database NoSQL (crediti immagine: MongoDB).

2.4 Mockup della User Interface

I Mockup permettono di fornire un’anteprima preliminare della *Graphic User Interface* dell’app, assieme alle funzionalità che questa metterà a disposizione.

Come già precisato in precedenza, i mockup sono solamente un’anteprima, un concept: la grafica finale potrebbe essere, ad esempio, cambiata o riadattata in base alle esigenze o agli strumenti di sviluppo forniti dalle componenti di Android.

Sono illustrati, di seguito, i Mockup di quelli che sono i risultati attesi dello sviluppo dell’applicazione in Android Studio.

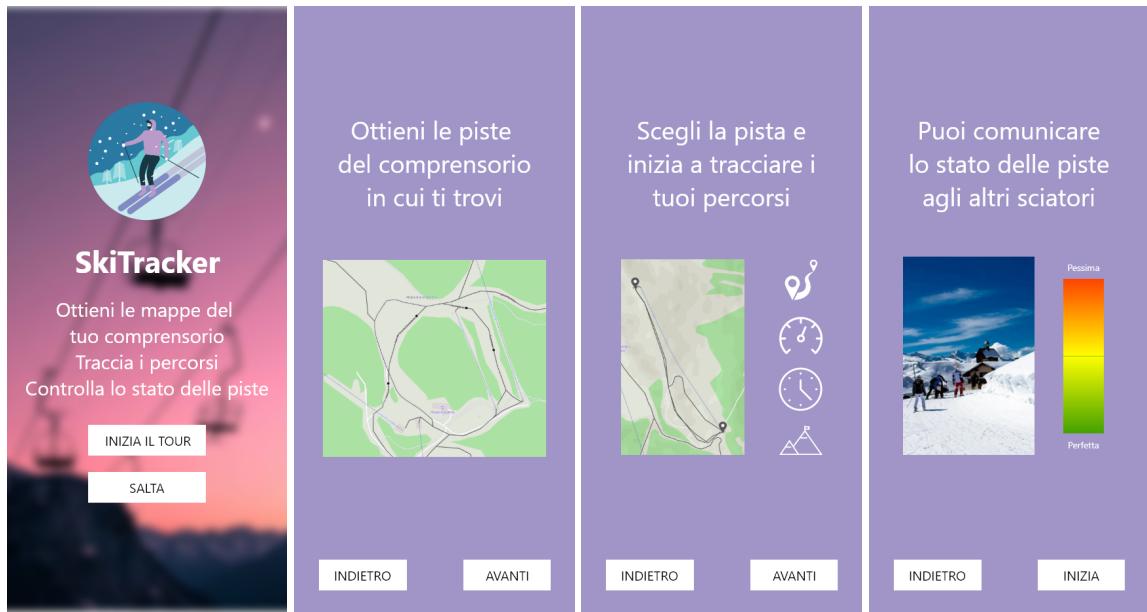


Figura 6: Tutorial iniziale, visualizzato al primo avvio dell'applicazione.

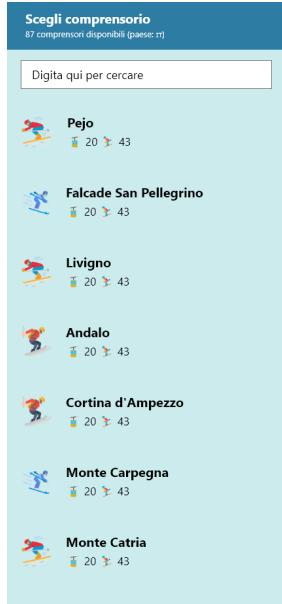


Figura 7: Scelta del comprensorio, visualizzata al primo avvio dell'app e anche se l'utente intende cambiare comprensorio da visualizzare.

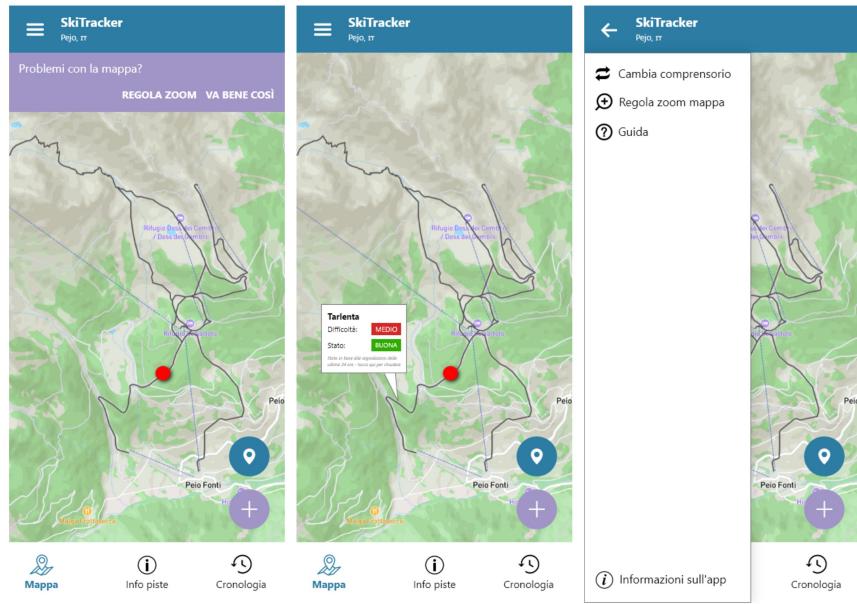


Figura 8: Vista principale, con menu aperto e pista selezionata.

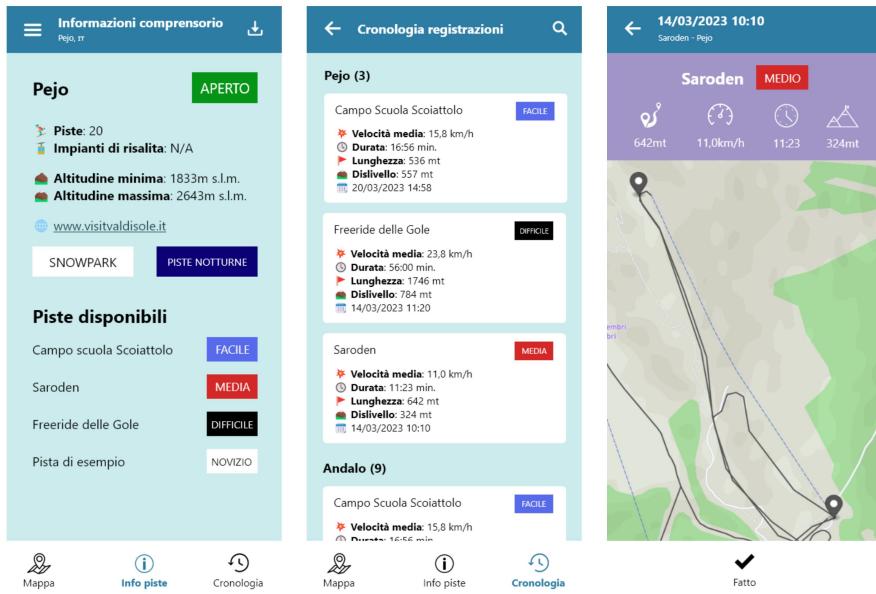


Figura 9: Pagine della home page: Informazioni sul comprensorio, cronologia percorsi tracciati e dettaglio di uno di questi.

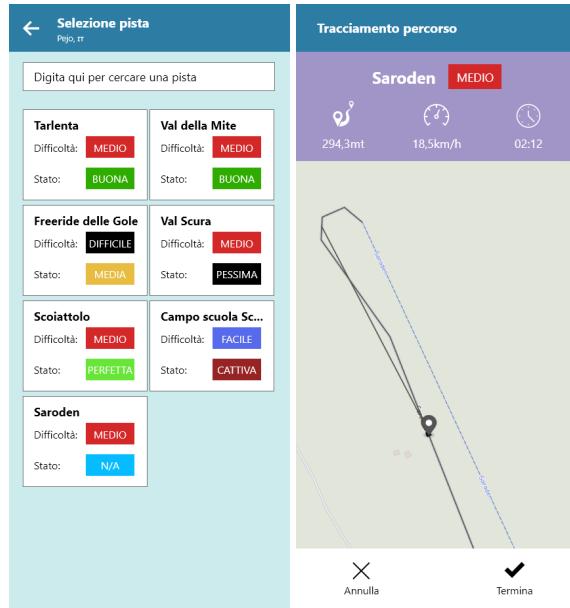


Figura 10: Tracciamento di un nuovo percorso: scelta della pista da tracciare e schermata di tracciamento.

2.5 Sviluppo dell'app

L'app è stata sviluppata utilizzando l'IDE **Android Studio**. La grafica dell'app è stata costruita usando le componenti offerte dalla libreria grafica **Material Design 2**.

2.5.1 Creazione e impostazione del progetto

Una volta installato l'IDE, abbiamo creato un **nuovo progetto vuoto** (come si può vedere in Figura 11), sviluppato in linguaggio Kotlin e che richiede come versione minima dell'SDK Android la 26 (quindi, l'app richiederà almeno Android 8.0 e successivi). Una volta forniti i dettagli di base (tra cui, il nome del package e della stessa applicazione), si può scegliere se partire da un'Activity vuota oppure da un *template*; abbiamo scelto per questo progetto di iniziare con un'**Activity vuota**, in modo tale da poter inserire più facilmente i nostri componenti e, in generale, per avere un punto di partenza e un'area di lavoro iniziale più pulita e minimale.

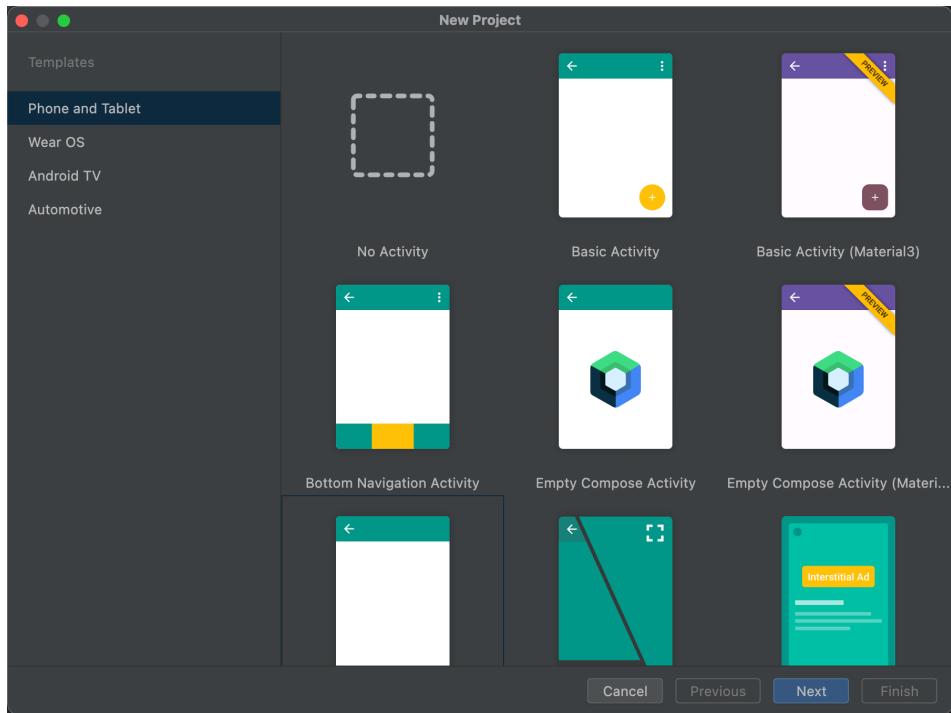


Figura 11: Selezione di un progetto "empty" (vuoto)

A questo punto, è stata creata un'applicazione "vuota".

Dopo questa configurazione iniziale, è stata **creata la prima activity**, relativa alla schermata di partenza da cui si inizierà (o si salterà) il tutorial.

In generale, per creare una nuova activity, come si può vedere in Figura 12, è sufficiente, una volta selezionato il package ove collocarla, richiamare la funzione `New → Activity → Empty Activity`, facendo clic destro sul nome del package.

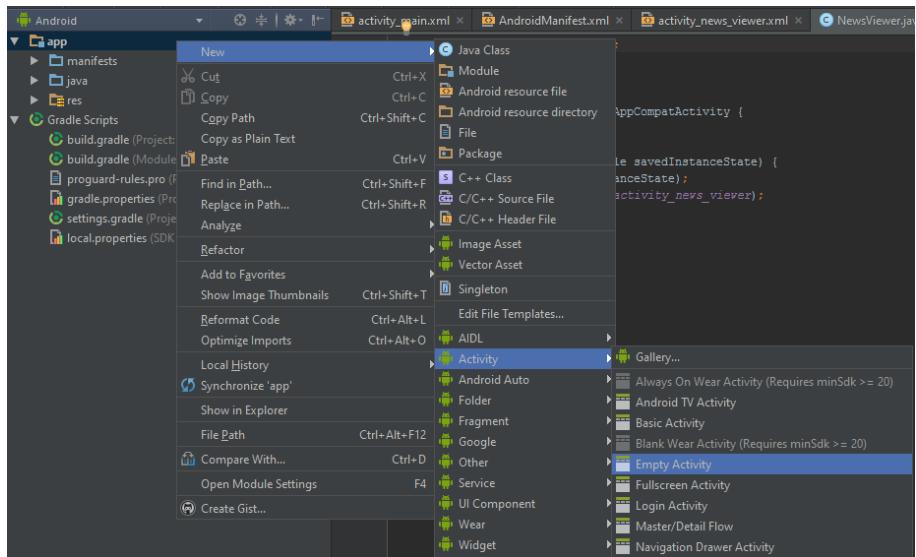


Figura 12: Come inserire una nuova activity

Ci si è mossi, quindi, nella realizzazione dei vari *Fragment*, relativi a questa nuova Activity. Ricordiamo la differenza tra Activity e Fragment:

- **Activity:** si tratta di una singola schermata, con cui l’utente può interagire. La si può pensare come un’intera finestra dell’applicazione.
- **Fragment:** si tratta di una porzione di interfaccia utente, collocata all’interno di una Activity. Permette di realizzare varie UI per l’applicazione, contenute direttamente in un solo contenitore, che è l’Activity.

Volendo essere più chiari, seppur anche più informali, nelle Activity sono presenti tutte le funzionalità comuni di una specifica attività, mentre nei Fragment sono presenti gli aspetti dettagliati di ogni singola funzionalità.

Un’applicazione, nel nostro progetto, di questa architettura Activity - Fragment è, ad esempio, nella realizzazione della schermata principale.

La schermata principale dell’app è infatti realizzata dall’Activity `MapActivity`, che contiene 3 relativi Fragment, relativi alle schermate che sono raggiungibili nell’area principale. La Activity contiene come “valori comuni” il `Navigation Drawer`, posto in alto a sinistra, e la `Bottom Navigation Bar`, posta in basso, che permette di passare tra i vari Fragment.

Nei Fragment vengono implementate le funzionalità specifiche delle tre sezioni, ossia *mappa del comprensorio, informazioni sul comprensorio e cronologia tracciamenti*.

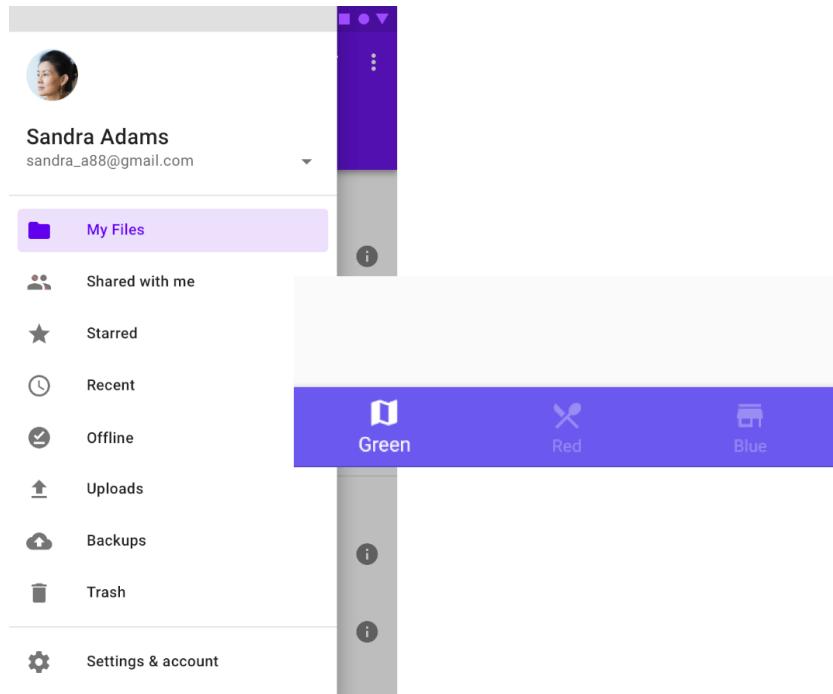


Figura 13: Navigation Drawer a sinistra, Bottom Navigation a destra.

2.5.2 Tutorial iniziale dell'app

A seguito della creazione e configurazione del progetto, abbiamo iniziato quindi con lo sviluppo dei Fragment e delle Activity, a partire dalla parte iniziale e più semplice: il **tutorial**. In questa parte, visualizziamo un tutorial, opzionale e saltabile dall'utente, che visualizza gli **aspetti peculiari della nostra app**, come le sue funzionalità.

Per quanto riguarda l'Activity, è stato sufficiente modificare il file XML di Layout per inserirci un `FragmentContainerView`, ovverosia un container che si occupa appositamente di contenere dei Fragment, che permette la comunicazione tra Fragment e Activity che la contiene.

Il lavoro più importante è stato svolto nei quattro fragment Welcome, Guida1, Guida2 e Guida3: questi Fragment contengono immagini e *Button* per consentire la **navigazione tra Fragment**.

In particolar modo, nel primo fragment, Welcome, vi sono i Button "Salta" e "Inizia tour", che servono rispettivamente a saltare e a iniziare il tutorial dell'app; nell'ultimo Fragment del tutorial, Guida3, il tasto "Avanti" viene sostituito con "Inizia", che serve per terminare il tutorial e iniziare la selezione del comprensorio.

2.5.3 Selezione del comprensorio

In questa Activity, verrà fornito l'**elenco di tutti i comprensori disponibili**, da cui si potrà selezionare quello in cui si desidera ottenere la mappa e fare tracciamenti. L'Activity offre anche la possibilità di **ricercare i comprensori per nome**, tramite una pratica barra di ricerca posta nella **AppBar**, in alto (indicata con il tasto della lente di ingrandimento).

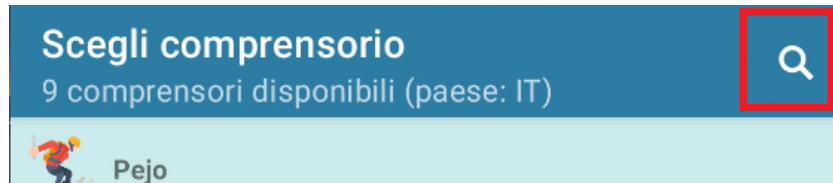


Figura 14: AppBar e pulsante di ricerca comprensorio.

Una volta selezionato il comprensorio di interesse, verranno **scaricati i dettagli** del comprensorio stesso (ad esempio, numero di piste e di impianti) e la lista di piste da cui è composto. L'app, se il download non va a buon fine o rileva che il comprensorio selezionato è chiuso o incompleto di informazioni, avvisa l'utente e lo invita a selezionare un'altro comprensorio.

Una volta selezionato il comprensorio, si verrà indirizzati alla MapActivity, dove è contenuta la mappa da visualizzare e, quindi, tutte le **funzionalità principali** fornite dall'applicazione.

Questa schermata sarà raggiungibile in un **prossimo momento** anche direttamente dalla MapActivity, nel caso si intenda cambiare comprensorio di interesse.

2.5.4 Area principale dell'app

MapActivity è l'Activity principale dell'app: contiene tutte le **funzionalità principali** offerte dalla nostra applicazione.

Nella AppBar, in alto, è presente un **Navigation Drawer** (noto anche come *Hamburger Menu*), che contiene le **funzioni secondarie** dell'Activity. In particolar modo, da questo menù è possibile accedere alle seguenti funzioni:

- **Regola zoom**: nel caso non siano visibili in mappa tutte le piste del comprensorio, è possibile regolare lo zoom della stessa tramite questa funzione. All'utente verrà chiesto se non vede alcune piste o ne vede troppe; nel primo caso, lo zoom verrà ridotto, nel secondo caso verrà aumentato. Il nuovo valore di zoom verrà salvato sul database e memorizzato permanentemente.
- **Cambia comprensorio**: permette di riaprire la schermata *Selezione comprensorio*, per cambiare comprensorio di interesse.

- **Ulteriori informazioni:** apre la activity AboutUsActivity, che contiene informazioni sugli sviluppatori e la versione dell'app.

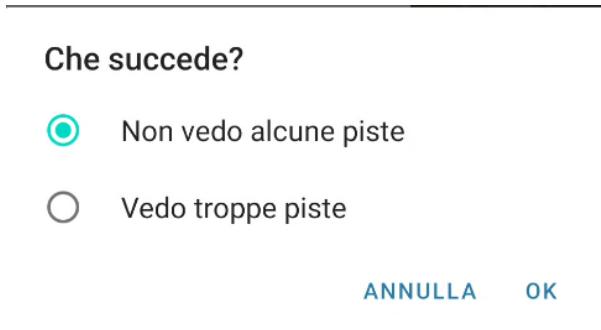


Figura 15: Finestra di dialogo per la regolazione dello zoom della mappa.

Le funzioni principali vengono offerte attraverso l'uso di **tre Fragment**, che vengono visualizzati in questa stessa Activity in un FragmentContainerView:

- **Mappa:** contiene la mappa del comprensorio comprensiva del tracciato delle piste (tecnicamente noto come *Overlay* delle piste).
- **Informazioni sul comprensorio** (info piste): contiene le informazioni principali del comprensorio selezionato e l'elenco delle sue piste.
- **Cronologia tracciamenti:** contiene la lista dei tracciamenti che sono stati fatti lungo le piste dei comprensori, raggruppati per comprensorio.

L'utente può **navigare tra questi tre fragment** tramite l'uso di una Bottom Navigation Bar, che permette, appunto, di scambiare il Fragment da visualizzare sul FragmentContainerView. La Bottom Navigation Bar infatti contiene tre pulsanti: Mappa, Info piste e Cronologia, che permettono di attivare i rispettivi Fragment.

All'avvio dell'applicazione viene sempre visualizzato il Fragment Mappa, che fa appunto da punto di partenza dell'utente.

Il **Fragment Mappa** contiene la **mappa del comprensorio**, assieme al **tracciato delle sue piste**, che viene disegnato dall'app stessa durante il caricamento iniziale.

La mappa viene implementata usando **OsmDroid**, una libreria di mappe alternativa a Google Maps, che fa uso del provider **OpenStreetMap**. È stato preferito l'uso di OpenStreetMap a

Google Maps in quanto più facile da implementare (non è richiesto alcun API Key, né nessun tipo di autorizzazione all’uso) e perchè più chiara nel visualizzare gli elementi di un comprensorio sciistico, come piste, impianti di risalita e punti di interesse.

Per l’ottenimento del tracciato delle piste, vengono fatte le seguenti operazioni:

1. Attraverso la API di Nominatim, viene ottenuta l’**area in cui il comprensorio giace**;
2. Usando la API di Overpass, viene analizzata quest’area e vengono ricercati e ottenuti tutti gli **elementi di tipo piste**, ovvero gli elementi che OpenStreetMap identifica come delle piste da sci.
3. Una volta ottenuto ciò, per ogni pista ottenuta, l’app ottiene tutte le **coordinate che la compongono** e va a disegnare una *Polyline* che le unisce. Inoltre, in base alla difficoltà della pista, alla polyline viene data un colore.
4. Una volta ottenute tutte le Polyline, queste vengono **visualizzate sulla mappa** e trattate come Overlay.

Assieme alla mappa, viene visualizzato anche un marcitore (Marker), che indica la **posizione corrente dell’utente** nella mappa. La posizione viene ottenuta tramite GPS, e si aggiorna automaticamente non appena viene rilevato uno spostamento lungo la mappa.

Questo Fragment contiene anche due **Floating Action Buttons (FAB)**, che servono per mettere in risalto le azioni principali di un’Activity/Fragment:

1. **Geolocalizzazione manuale.** Se non è ancora stata trovata la posizione GPS va a richiederla; una volta ottenuta (o se lo era già stata), centra la mappa sulla posizione dell’utente.
2. **Inizio Tracciamento.** Cliccando su questo, verrà aperta la RouteTrackingActivity, dove sarà possibile gestire l’attività di tracciamento di una pista.

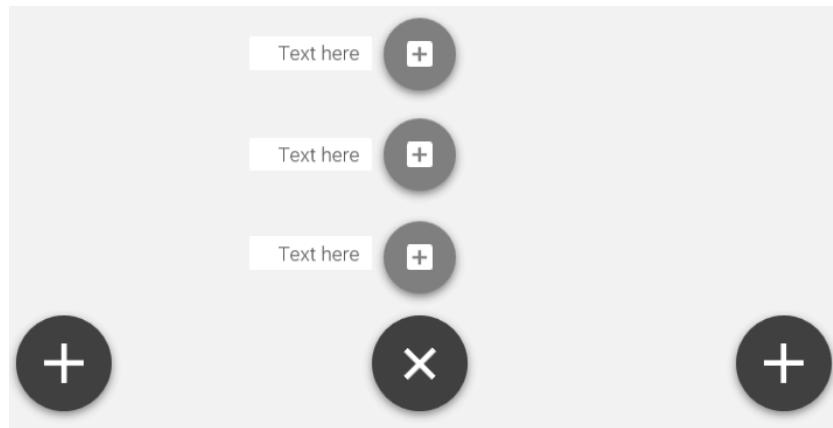


Figura 16: Floating Action Button

Nel **Fragment Informazioni sul comprensorio**, vi sono tutte le informazioni sul comprensorio selezionato.

In una prima sezione sono presenti le **informazioni generali** sul comprensorio, quali: nome, stato (aperto/chiuso), numero di piste, altitudine minima e massima e link al sito web.

Vi sono anche degli **indicatori grafici** che si attivano quando il comprensorio comprende piste notturne e/o snowparks.

Nella seconda sezione, invece, viene visualizzata la **lista di piste** del comprensorio: per ognuna di queste, si hanno il nome e la difficoltà, con indicazione colorata.

È interessante discutere la sua implementazione: tramite il file XML del layout sono state implementate le varie `TextView` e i vari `Button`, i cui contenuti (colore e testo) sono stati poi gestiti in Kotlin. Nella figura sottostante viene mostrato il codice che è stato usato per consentire queste operazioni.

```

    numPiste.text = "${skiArea.getNumPiste()}"
    impiantiRisalita.text = "${skiArea.getNumImpianti()}"
    max.text = "${skiArea.getMaxAlt()}"
    min.text = "${skiArea.getMinAlt()}"
    sito.text = "${skiArea.getWebSite()}"
```

//Personalizzo la scritta che indica se quel comprensorio è aperto o chiuso

```

    val stato = view.findViewById<TextView>(R.id.stato)
    val aperto = skiArea.isOperativo()

    if (aperto) {
        stato.text=APERTO
        context?.let { ContextCompat.getColor(it, R.color.green) }
            ?.let { stato.setBackgroundColor(it) }
    } else {
        stato.text=CHIUSO
        context?.let { ContextCompat.getColor(it, R.color.red) }
            ?.let { stato.setBackgroundColor(it) }
    }
```

//gestisco le scritte snowpark e piste notturne

```

    val snowpark = view.findViewById<TextView>(R.id.snowpark)
    val night = view.findViewById<textView>(R.id.piste_notturne)

    val showsnow = skiArea.getSnowPart()
    val shownight = skiArea.getNight()

    if (showsnow){
        snowpark.visibility = View.VISIBLE
    } else {
        snowpark.visibility = View.GONE
    }

    if (shownight){
        night.visibility = View.VISIBLE
    } else {
        night.visibility = View.GONE
    }
}

```

Figura 17: Parte del codice usato per la gestione dell’interfaccia grafica del Fragment Info comprensorio.

Un’altra tecnologia che è opportuno riportare è l’uso della **RecyclerView**. In questo Fragment, sono state utilizzate per riportare l’elenco di piste disponibili per il comprensorio selezionato.

Le RecyclerView sono una classe di Android che permettono di visualizzare **grandi insiemi di dati** in modo efficiente e veloce. Essa è una **ViewGroup** che contiene le viste corrispondenti ai dati, ma rispetto a quest’ultime, le viste non vengono ricreate continuamente, ma **memorizzate e "riciclate"** secondo un sistema sofisticato; in questo modo, liste molto grandi vengono gestite con un impatto minimo sulle prestazioni. Difatto una RecyclerView una vista, quindi la si aggiunge al layout come si farebbe con qualsiasi altro elemento dell’interfaccia utente.

In generale, le RecyclerView sono utilizzate per la creazione di liste e griglie di elementi.

Per costruire una RecyclerView è necessario definire un **Adapter**, che è responsabile della creazione delle viste che rappresentano i dati e della loro gestione.

L’Adapter di una RecyclerView ha tre funzioni principali:

- `onCreateViewHolder ()`: crea nuove istanze di ViewHolder quando il Layout Manager ne richiede una.
- `onBindViewHolder ()`: popola ogni ViewHolder con i dati corretti.

- `getItemCount ()`: restituisce il numero totale di elementi nella collezione di dati gestita dall'Adapter.

Nella figura sottostante viene mostrato un esempio di RecyclerView e degli elementi che la compongono, separati in modo chiaro:

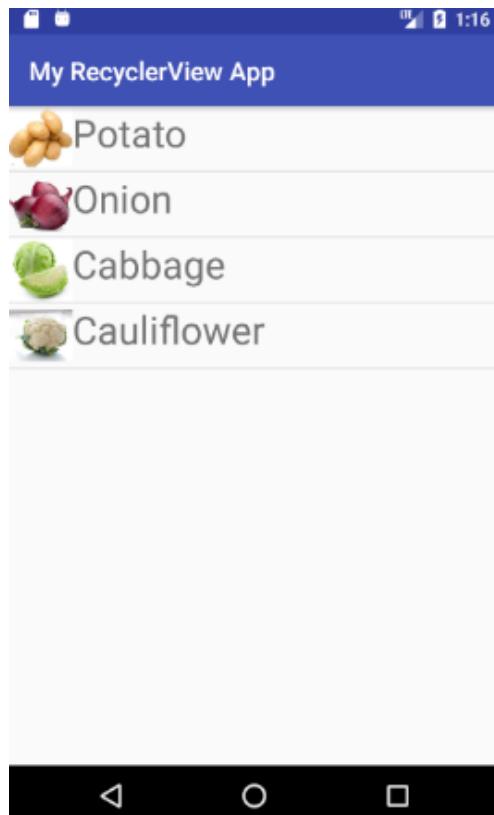


Figura 18: RecyclerView per mostrare la lista di Piste

Nel **Fragment Cronologia**, è contenuto l'**elenco di tutti i percorsi** che sono stati registrati. Inizialmente vuoto, viene aggiornato automaticamente mano a mano che vengono svolti nuovi tracciamenti.

Non occorre fare altro che **scegliere di quale comprensorio** si desidera vedere i percorsi tracciati e verrà restituito tutto l'elenco.

Anche per questo Fragment è stata usata una RecyclerView per mostrare l'elenco di tracciamenti, assieme ai loro dettagli: data e ora, velocità media, dislivello tra punto di partenza e punto di termine, durata in minuti e secondi e lunghezza del tragitto.

È stato inoltre usato uno **Spinner** per effettuare il filtraggio dei percorsi tracciati per comprensorio. Lo Spinner è un tipo di Widget che visualizza un **elenco di opzioni** tra cui l'utente può scegliere: all'interno di questo, vengono inseriti solamente i comprensori in cui l'utente ha effettuato almeno un tracciamento in passato.

Quando la lista della cronologia è vuota, viene visualizzato un messaggio che informa l'utente di ciò.

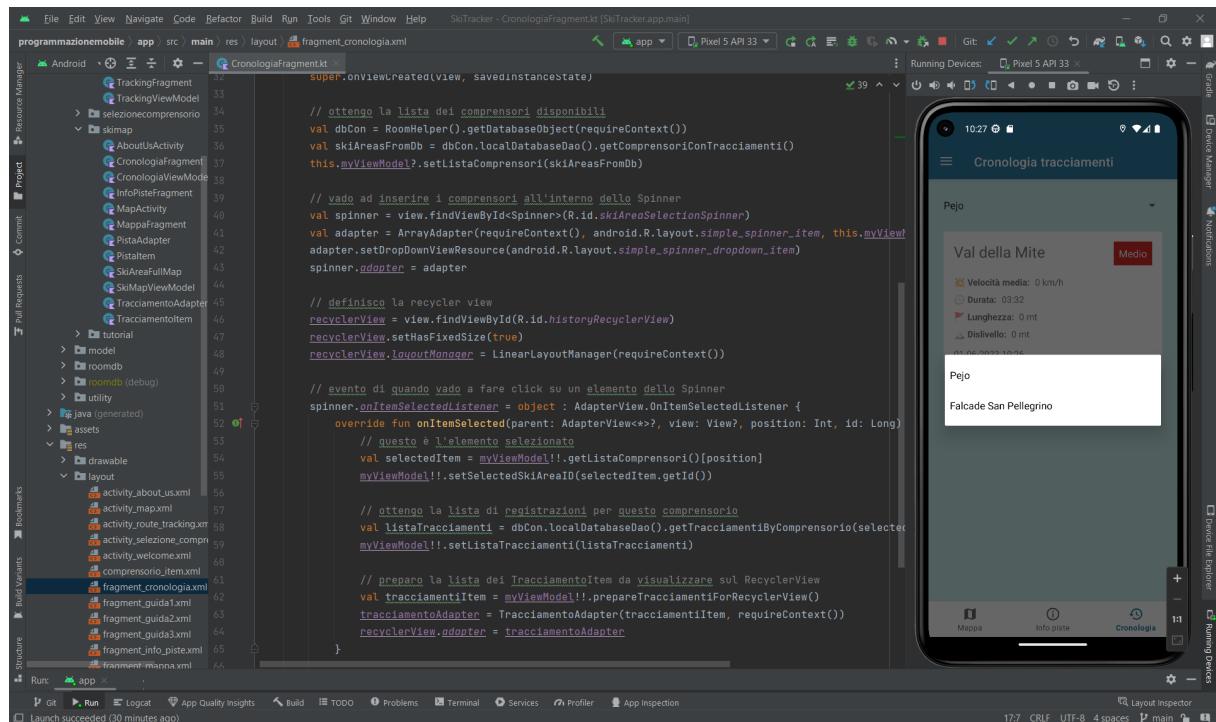


Figura 19: Cronologia dei percorsi

2.5.5 Tracciamento del percorso

La nostra app offre anche la possibilità di **tracciare un percorso**: quando l'utente percorre una pista da sci, l'app calcola **statistiche della sua prestazione** sciistica, come velocità, dislivello e durata del percorso, per poi memorizzarle e visualizzarle sul Fragment Cronologia.

L'intera funzionalità di questa parte di app viene concentrata sull'Activity `RouteTrackingActivity`; le funzionalità di selezione della pista da tracciare e il tracciamento stesso vengono **suddivise in due Fragment** appositi: `RouteSelectionFragment` e `TrackingFragment`. Questi Fragment, analogamente a quanto accade già in altre realtà dell'app, vengono contenuti in un `FragmentContainerView`.

Una volta aperta l'Activity, viene aperto il Fragment `RouteSelectionFragment`, che si occupa di visualizzare (tramite l'uso della `RecyclerView`) la **lista di piste** del comprensorio che è stato selezionato. L'utente, tramite questo Fragment **seleziona la pista** che intende tracciare: una volta fatto, viene avviato il tracciamento, e quindi si passa al Fragment `TrackingFragment`.

L'**attività stessa di tracciamento** del percorso si svolge all'interno del Fragment `TrackingFragment`. Inizialmente, viene **recuperata la posizione** GPS dell'utente, per ottenere la posizione iniziale del tracciamento. Il conteggio del tempo verrà avviato una volta che è stata ottenuta la posizione iniziale.

Una volta iniziato il tracciamento, gli **spostamenti vengono individuati** e gestiti da un `LocationListener`: quando questi rileva un cambiamento della posizione, ne intercetta le coordinate e individua la velocità con cui è stata cambiata la posizione (approssimata). Successivamente, viene calcolata la distanza e il dislivello tra i vari punti costituenti la rotta del tracciamento.

Il **conteggio del tempo** di tracciamento viene invece affidato al `Chronometer`, un componente di Android che implementa un sistema di conteggio, di tipo *count-down* o *count-up*.

L'utente può annullare il tracciamento in qualsiasi momento oppure interromperlo e salvare il percorso fatto. Una volta salvato il percorso, questo sarà disponibile sul Fragment Cronologia.

Il layout del `TrackingFragment` è organizzato in due parti: nella prima, vi è un `TableLayout` dove sono contenuti: il nome e la difficoltà selezionata, la velocità istantanea, il tempo trascorso e la distanza percorsa; nella seconda, vi è invece la mappa, dove è contenuto il marcatore della posizione GPS e i pulsanti Annulla e Termina.

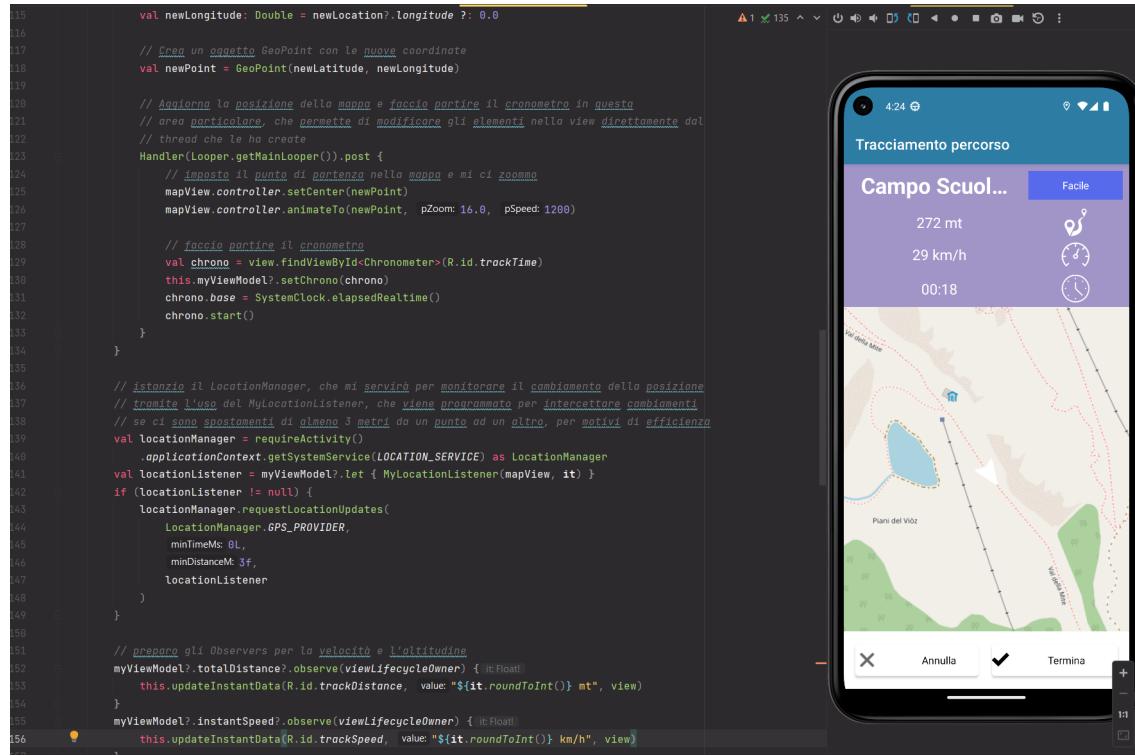


Figura 20: Parte di codice per il tracciamento di un percorso, assieme ad un esempio.

2.5.6 Problematiche note e conclusioni

Durante lo sviluppo del progetto in Kotlin, si sono verificate alcune problematiche. Verso la fine del mese di maggio, la API di SkiMap, utilizzata per l’ottenimento della lista di comprensori, è stata disattivata: non è stato quindi più possibile usufruirne delle funzionalità, compromettendo così una applicazione fino al momento completa e funzionante. Si è quindi deciso di passare ad una soluzione alternativa: è stato creato un database fisso contenente alcuni comprensori già preparati allo scopo (nove). In questo modo, il progetto è stato comunque completato e consegnato, pur mancando dell’uso dell’API che avrebbe reso dinamico ed automatico l’aggiornamento della lista di comprensori. Questo database è presente nei file di progetto nella cartella `assets/app.db`.

Inoltre, il progetto non presenta alcune funzionalità che erano state previste: non sono presenti l’integrazione con Firebase per l’aggiornamento dello stato delle piste e i dettagli di un tracciamento in Cronologia; questo a causa del ritiro del terzo componente del progetto, e quin-

di per l'ottimizzazione dei tempi che è stata imposta di conseguenza.

Ci sono state problematiche anche nella gestione della programmazione asincrona in Kotlin: le chiamate API verranno quindi effettuate in modalità sincrona, bloccando, nel peggio dei casi, l'app per qualche secondo. Ovviamente, gli eventi eccezionali, come il timeout della richiesta, sono stati gestiti correttamente.

2.6 Testing delle funzionalità

La parte di Testing dell'applicazione è una attività finale che viene effettuata per **verificare che non ci siano mai errori** in una funzione implementata. Per i nostri scopi, sono state implementate quattro funzioni di test: due di queste sono **Instrumented Test**, le altre due sono **Unit Test**.

Le prime appartengono ad una categoria di testing utili a simulare l'**interazione con l'applicazione**, come se la stessimo usando nella vita reale. Per una corretta esecuzione, si consiglia di **usare un telefono fisico**, collegato via USB al PC, e con il Debug USB abilitato; l'emulatore è risultato inadeguato all'esecuzione di questa tipologia di test.

In questo caso, si è deciso di sviluppare un test per verificare la corretta interazione con il Bottom Navigation Bar (riportato nella figura sottostante) ed uno per verificare il corretto funzionamento del Floating Action Button che apre l'Activity di tracciamento.

```

    @Test
    fun testFragmentChange() { // Testa il cambio di Fragment cliccando sulla Bottom Navigation Bar
        val fragmentManager: FragmentManager = getActivity()?.supportFragmentManager
        ?: throw IllegalStateException("FragmentManager is null")

        val currentFragment = getCurrentFragment(fragmentManager)
        val newFragment = InfoPisteFragment()

        // Esegui la transazione del Fragment
        fragmentManager.beginTransaction()
            .replace(R.id.mapaFragment, newFragment)
            .addToBackStack(null)
            .commit()
    }

    private fun getActivity(): FragmentActivity? {
        return activityTestRule.activity
    }
}

```

Test Results
> Task :app:createDebugAndroidTestApkListning is UP-TO-DATE
Connected to process 8818 on device 'oppo-cph2247-a9bd61a1'.
Tests
 > Test Results 2s 1/1
 > MyTest 2s 1/1
 > testFragmentChange 2s ✓

> Task :app:connectedDebugAndroidTest
Starting 1 tests on CPH2247 - 13

BUILD SUCCESSFUL in 13s
67 actionable tasks: 1 executed, 66 up-to-date
Build Analyzer results available

Figura 21: Instrumented Test

Riguardo alla seconda categoria, essi appartengono alla più nota categoria di Testing. Gli **Unit Test** vengono usati per verificare il **corretto funzionamento di alcune funzioni e/o di classi**.

A differenza di quelli strumentali, essi non richiedono l'uso di un telefono o un emulatore, bensì vengono **eseguiti direttamente su un terminale**. In questo progetto sono stati implementati due test che verificano il corretto funzionamento del Database. Le due funzioni inserite sono visibili nella figura sottostante.

The screenshot shows the Android Studio IDE with the file `ExampleUnitTest.kt` open. The code is written in Kotlin and contains unit tests for a database operation. The code includes annotations for test setup and teardown, and assertions for test results.

```
1 ExampleUnitTest.kt
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
```

The code defines two test functions: `testPistaInComprensorio()` and `testTutorialCompleto()`. It uses Room's in-memory database builder to set up a database instance and then performs assertions on the results of the database queries.

Figura 22: Unit Test

3 Sviluppo con Flutter

La seconda modalità di implementazione, e quella a cui si è dedicato meno tempo, prevede la realizzazione della stessa applicazione, seppur con meno funzionalità, utilizzando il **framework Flutter**. Un framework è un ambiente di sviluppo che fornisce una certa rigidità schematica nel modo di progettare software. Essi esistono in diversi contesti ed ognuno utilizza un suo approccio.

Nello specifico, Flutter è un **framework open-source** creato da Google e pensato per sviluppare applicazioni mobile installabili su **diversi sistemi operativi** (dette *cross-platform*), caratterizzati da un'**interfaccia grafica in comune**, basata sulla libreria grafica **Material 3**. Lo stesso codice in Flutter può essere usato anche per realizzare siti web o applicazioni desktop. Adopera, come linguaggi di programmazione, C, C++ e **Dart**, quest'ultimo è quello che verrà usato nello sviluppo del progetto in Flutter.

Particolarità di Flutter è che, contrariamente a tanti framework per la creazione di app cross-platform, ad esempio **Cordova**, è che il codice sorgente dell'app viene **compilato nativamente nel sistema di destinazione**: ad esempio, è possibile ottenere, dallo stesso codice Dart, un'app sia per Android che per iOS, tutto questo facendo sì che la compilazione del codice Dart venga eseguita in modo nativo, attraverso i sistemi di compilazione previsti dai vari sistemi operativi (**Gradle** nel caso di Android, ad esempio). In questo modo, abbiamo un'app il cui codice può essere facilmente integrato in altri sistemi operativi, e che presenta una grafica quasi identica per ogni piattaforma.

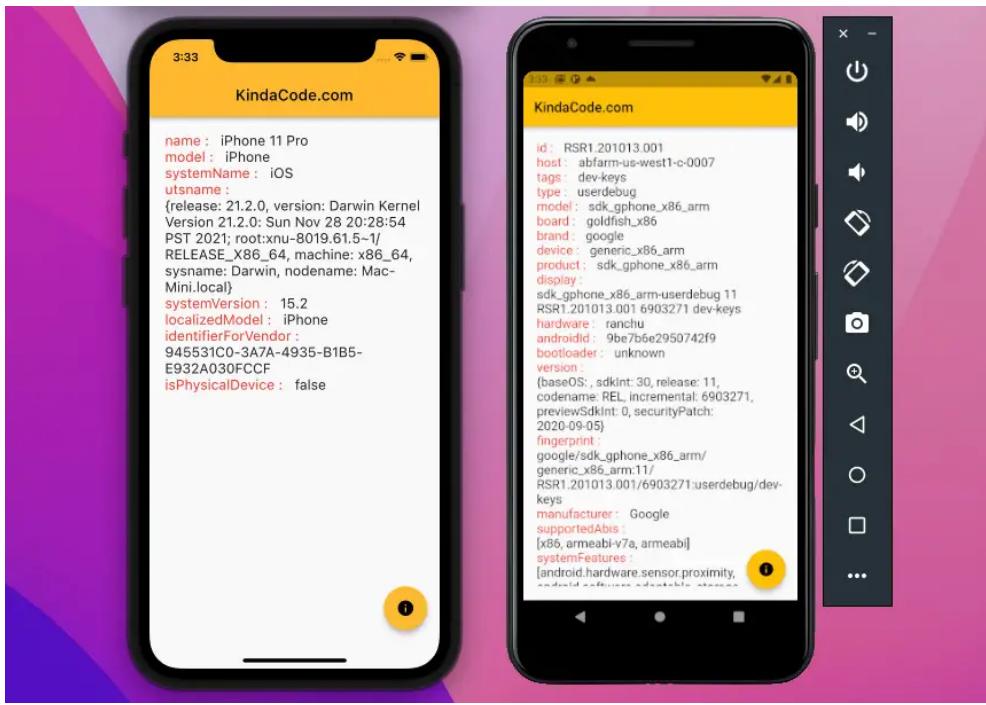


Figura 23: Stessa app in Flutter, eseguita su iOS (a sinistra) e su Android (a destra) (crediti foto: KindaCode.com)

3.1 Installazione di Flutter

Per sviluppare app Flutter possono essere usati vari IDE ed editor di testi: ad esempio, la documentazione di Flutter consiglia l'uso di [Visual Studio Code](#), un editor di testo personalizzabile con le estensioni, oppure direttamente lo stesso [Android Studio](#), a cui viene **aggiunto un plugin** per integrare le funzionalità di gestione di codice in Flutter e Dart.

Nel nostro caso, abbiamo preferito rimanere sull'IDE Android Studio, e per prepararlo allo sviluppo in Flutter abbiamo svolto i seguenti passaggi:

- **Installare Flutter e l'SDK di Dart.** L'installazione di Flutter, spiegata in [questa guida](#), consiste in sostanza nel scaricare la versione di Flutter adatta al proprio sistema operativo e scaricare i programmi e le librerie necessarie al suo funzionamento. L'SDK di Dart viene incluso nel download di Flutter.
- **Verificare l'installazione di Flutter e Dart.** Attraverso l'utilità `flutter doctor` abbiamo verificato che Flutter fosse perfettamente installato e configurato sui nostri sistemi, e abbiamo risolto le eventuali mancanze di librerie/software segnalateci.

- **Installare il Plugin su Android Studio.** Una volta aperto l'IDE, si accede al menù File → Settings → Plugins. Si dovrà, in seguito, ricercare il plugin Flutter e poi installarlo. Una volta fatto ciò, sarà possibile creare e aprire progetti Flutter correttamente.

Completati i seguenti passaggi, si potrà verificare che l'installazione è andata a buon fine, andando a creare un nuovo progetto in Flutter selezionando File → New. Tra le opzioni, comparirà anche l'opzione Flutter Project, come si può vedere nella figura sottostante.

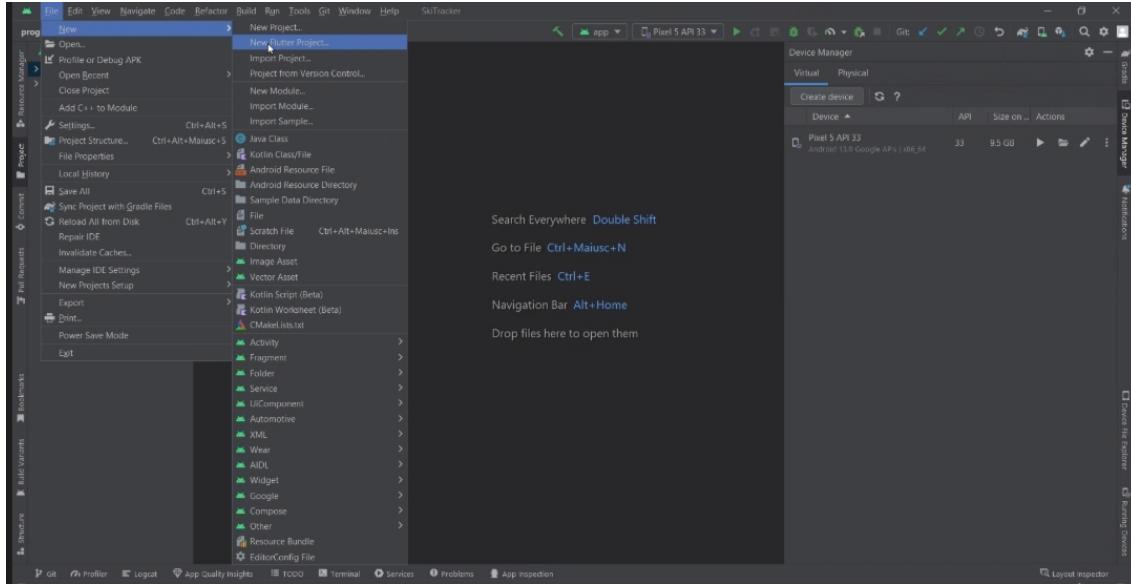


Figura 24: Opzione per creare un nuovo progetto Flutter su Android Studio.

3.2 Requisiti funzionali e non funzionali

L'applicazione realizzata in Flutter presenta delle funzionalità in meno rispetto a quella implementata in Kotlin. Di seguito verranno riportati i requisiti funzionali e non funzionali, come fatto in precedenza per l'app in Kotlin.

Riguardo ai requisiti funzionali, sono stati individuati i seguenti:

1. **Selezione del comprensorio:** il cliente deve poter selezionare il comprensorio di cui intende ottenere le piste e la mappa.
2. **Visualizzazione della mappa del comprensorio scelto:** l'app visualizzerà una mappa che comprenderà tutte le piste del comprensorio scelto dal cliente.

3. **Visualizzazione delle informazioni del comprensorio:** dalla nostra app sarà possibile vedere tutte le informazioni del comprensorio sciistico selezionato dal cliente, ad esempio: numero di piste, numero impianti di risalita, massima e minima altitudine sul livello del mare, etc.
4. **Regolazione dello zoom della mappa:** per via di una limitazione delle API impiegate dall'app, in casi particolari il cliente potrebbe imbattersi nella situazione spiacevole in cui nella mappa non vengono visualizzate tutte le piste del comprensorio; è possibile risolvere questo problema regolando lo zoom della mappa ottenuta con le API, funzione che l'app renderà disponibile al cliente nel caso debba esserci necessità.
5. **Geolocalizzazione dell'utente sulla mappa:** l'app consentirà la visualizzazione della posizione in tempo reale (ottenuta tramite GPS) del cliente lungo il comprensorio.

Riguardo, invece, i requisiti non funzionali, sono stati individuati i seguenti:

1. **Scrittura del codice in linguaggio Dart:** il codice dell'applicazione verrà scritto in linguaggio Dart, e si fa uso del framework Flutter.
2. **Uso di API esterne:** l'app, per ottenere varie informazioni farà uso di due [API \(Application Programming Interface\)](#):
 - (a) **Nominatim:** API che usa la tecnologia di [OpenStreetMap](#) per la geocodifica della posizione del comprensorio: fornisce l'area in cui il comprensorio è compreso.
 - (b) **Overpass API:** API basata sulle tecnologie di OpenStreetMap, che ottiene l'area del comprensorio e la elabora, aggiungendole i punti di interesse, come rifugi, impianti di risalita e le piste.
3. **Salvataggio dati dell'app su un database interno:** i dati necessari al funzionamento dell'app (come il comprensorio selezionato) verranno salvati su un database interno all'app, implementato con [SQLite](#).

3.3 Architettura dell'app

Come si discuterà ampiamente in seguito, in Flutter vi è un concetto fondamentale che lo rende diverso dalla programmazione in Kotlin: ogni elemento grafico è un **Widget**. Lo sono anche tutti i Layout che verranno successivamente descritti ed illustrati.

A livello di Pattern, Flutter non implementa l'approccio Model - View - ViewModel (MV-VM) tipico di Android, ma usa un'**architettura più libera** e meno rigida.

Viene usato quindi un paradigma simile ad un **Model - View** (MV), dove la View, rappresentata dai **Widget stateful** (widget che presentano dati variabili) e **stateless** (che non presentano dati variabili) contiene anche le logiche che permettono di interagire con i dati dell'app, custoditi e gestiti dai Model.

Per quanto riguarda invece la **disposizione dei Widget** e dei componenti che compongono un'interfaccia grafica, Flutter presenta invece un'architettura molto più rigida, riassunta col seguente schema:

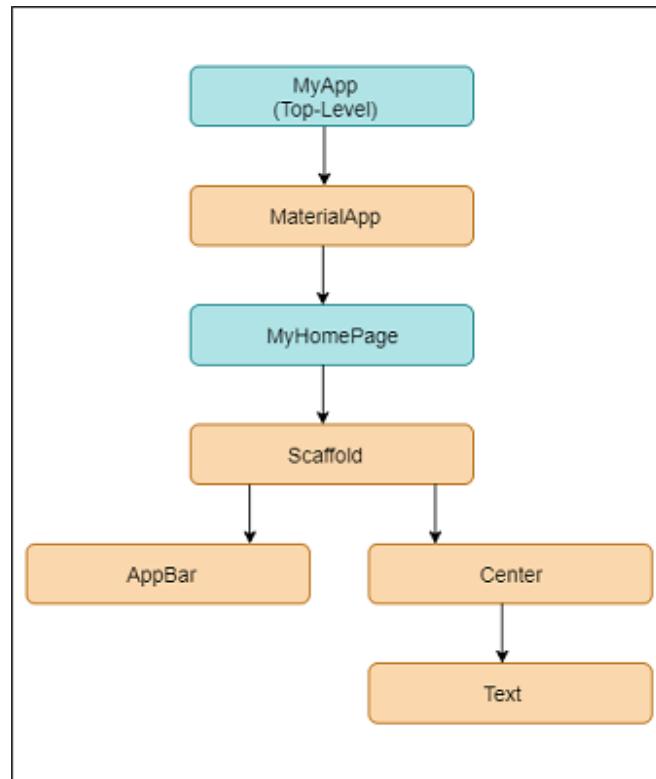


Figura 25: Architettura della GUI in Flutter.

Al vertice, vi è l'oggetto **MyApp**, che rappresenta l'applicazione in se. Subito sotto vi è la **MaterialApp**, ovvero la libreria grafica impiegata per la GUI dell'app.

Successivamente vengono i **vari Widget** di cui è composta l'app, siano questi stateless o statefull. Gli elementi grafici che compongono ognuno di questi sono a loro volta dei Widget, e sono inseriti all'interno di un contenitore, denominato **Scaffold**.

Nell'esempio riportato precedentemente, vi è un unico widget, `MyHomePage`, il cui Scaffold contiene l'AppBar (la barra superiore dell'app) e un contenitore Center, che a sua volta contiene un Text, che viene allineato al centro.

3.4 Database dell'app

Lo schema Entity - Relationship del database dell'app in Flutter è riportato nella figura sottostante. Le differenze con la versione in Kotlin sono pressoché nulle, se non per l'assenza delle entità relative al Tracciamento. Questo semplicemente perchè il tracciamento del percorso è una delle funzionalità che, nell'app implementata in Flutter, non si è riproposta.

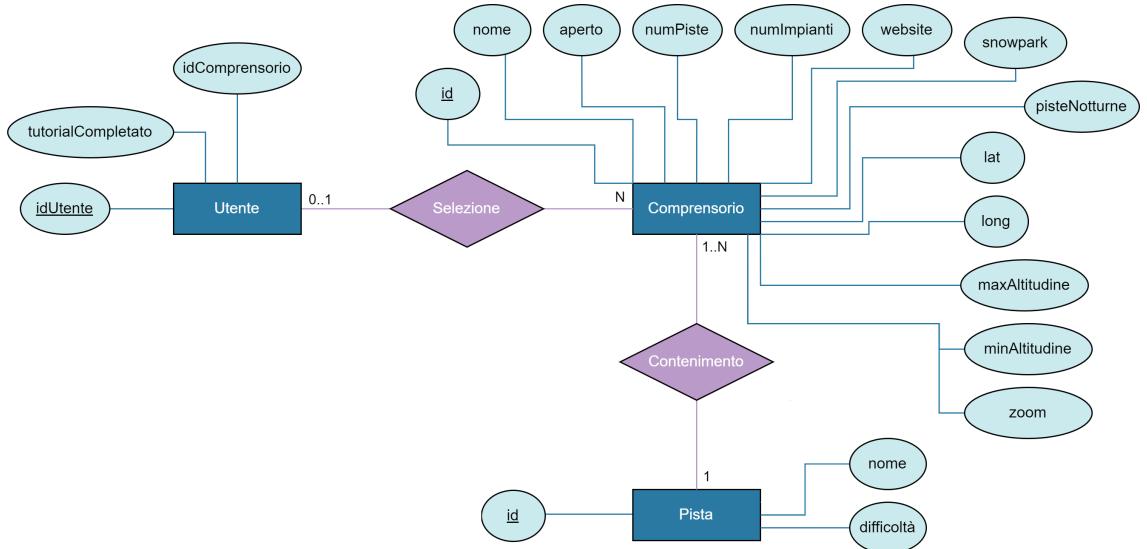


Figura 26: Database del progetto (Flutter).

3.5 Mockup della User Interface

Di seguito vengono illustrati i Mockup dell'interfaccia grafica dell'app in Flutter. Si precisa come i Mockup siano solamente un'anteprima della GUI dell'app, e pertanto potrebbero non rispecchiare il prodotto finale.



Figura 27: Mockup delle varie interfacce: Mappa del comprensorio, informazioni sul comprensorio, scelta di un nuovo comprensorio e informazioni sull'app.

3.6 Sviluppo dell'app

Anche per lo sviluppo dell'app in Flutter, come detto in precedenza, si è scelto di usare Android Studio come IDE.

Una grande ed importante differenza tra la programmazione di applicazioni mobile in Kotlin e in Flutter è che la seconda modalità non prevede l'uso di Activity e Fragment per la suddivisione delle varie interfacce utente. La filosofia di programmazione è diversa, in quanto in Dart e Flutter ogni componente grafico è un Widget.

Il Widget principale è lo Scaffold, che occupa tutto lo spazio disponibile dello schermo, e che fa da scaffale contenitore di altri widget. Una volta definito, al suo interno vengono inseriti e definiti tutti i componenti da visualizzare.

3.6.1 Definizione dell'area principale

A partire dall'area principale dell'app, definita in main.dart, i componenti principali sono i seguenti:

- **AppBar.** Essa rappresenta la barra in alto, contenente il nome dell'applicazione.
- **Body.** Il corpo dell'applicazione, contiene un riferimento alla classe selezionata nella Bottom Navigation Bar. La selezione impostata di default è quella della pagina riportante la Mappa.
- **Bottom Navigation Bar.** Presenta i riferimenti alle tre aree realizzate per il progetto. Per passare da un'area all'altra, utilizza un indice per distinguere le tre classi Mappa, InfoPiste

e AboutUs. Cliccando su una delle opzioni, verrà individuata la pagina interessata, che andrà a sostituire quella pre-esistente.

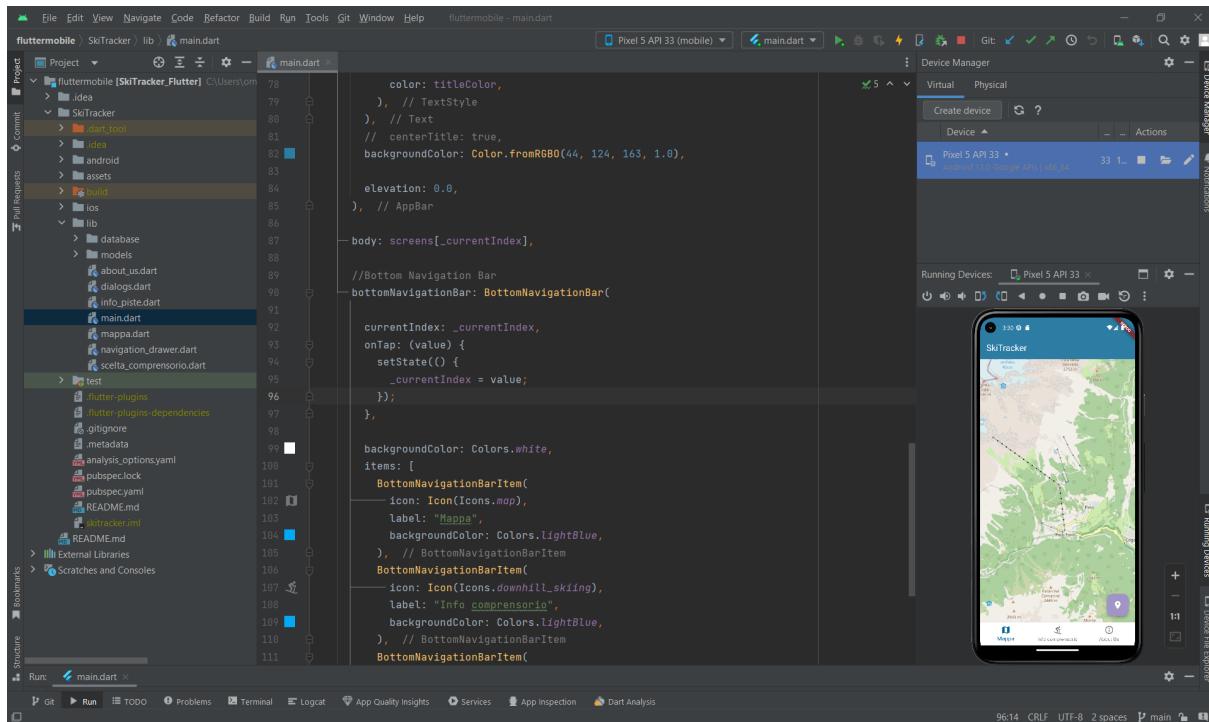


Figura 28: Home page dell'applicazione avviata di default e relativa visuale personalizzata nella Bottom Navigation Bar.

Le tre classi, che identificano le tre aree dell'app (Mappa, Informazioni sul comprensorio e About Us) appartengono a tre file .dart distinti, che vanno ovviamente importati in testa al documento main.dart. Dettagliamo ora questi file.

3.6.2 Mappa del comprensorio

Contenuto nel file mappa.dart, visualizza la **mappa del comprensorio** selezionato, fungendo quindi da **pagina principale**. Se non è stato selezionato alcun comprensorio, si limita a visualizzare un messaggio che invita l'utente a selezionarne uno dall'area Informazioni comprensorio.

Nello stesso file sono definiti anche i due **Floating Action Button**, uno che consente di aggiornare la posizione dell'utente sulla mappa, ed un secondo che permette di regolare lo zoom, in

caso di problemi nella visualizzazione di alcune piste.

Per l'ottenimento della mappa si fa uso della libreria `flutter_map`, mentre la gestione del tracciato delle piste è molto simile alla versione in Kotlin: l'area del comprensorio viene **geocodificata** tramite l'API di Nominatim e gli **elementi di tipo pista** vengono ottenuti tramite Overpass: l'app si preoccupa poi di **interpretare l'elenco di piste** nella mappa, tramite `Polyline` colorate in base alla difficoltà della pista che indicano. In Flutter, tuttavia, a causa di limitazioni dovute all'uso delle librerie principali per la geolocalizzazione, il marcatore della posizione, che viene inserito sulla mappa una volta individuata la posizione dell'utente, non si aggiorna dinamicamente: va aggiornato attraverso l'uso del Floating Action Button *Aggiorna posizione GPS*.

3.6.3 Informazioni sul comprensorio

Contenuta nel file `info_piste.dart`, presenta **due macro sezioni**. La prima riporta tutte le **informazioni del comprensorio scelto**, quindi nome, stato, numero di piste, etc. La seconda è una `ListView` contenente tutte le **piste del comprensorio** ed i relativi livelli di difficoltà. Se non è stato selezionato ancora nessun comprensorio, l'app visualizza un messaggio, che invita l'utente a selezionare un comprensorio. Tutte queste informazioni vengono **caricate dal database locale** e visualizzate in questo widget, organizzate in delle `Row` (righe).

È inoltre presente, anche qui, un Floating Action Button, che viene usato per aprire il widget Selezione comprensorio, che permette di **selezionare o cambiare il comprensorio** di interesse. Una volta che l'utente avrà selezionato il comprensorio, verrà riaperta questa sezione, e verrà ricaricata con i dati del comprensorio appena selezionato.

3.6.4 Selezione del comprensorio

Accedendo alla pagina di scelta del comprensorio (contenuta nel file `scelta_comprensorio.dart`), la cui schermata è riportata a pagina successiva, sarà interessante notare come cambia tutto il corpo dell'applicazione.

La presente AppBar contiene un pulsante per tornare alla pagina di informazioni del comprensorio e, soprattutto, una **barra di ricerca** indicata con la scritta *cerca qui il tuo comprensorio*. Cliccandoci e scrivendoci, sarà possibile effettuare una **ricerca dei comprensori** in base al nome inserito e, se non vi è alcuna corrispondenza tra l'input dell'utente e l'elenco di comprensori nel database, verrà visualizzato un messaggio di errore.

La **lista di comprensori** viene caricata dal database e **visualizzata in un ListView**; una volta che è stato selezionato un comprensorio, analogamente all'app in Kotlin viene scaricato l'elenco di piste di quel comprensorio e si ritorna al widget Informazioni sul comprensorio, che verrà ricaricato con le informazioni del comprensorio appena selezionato.

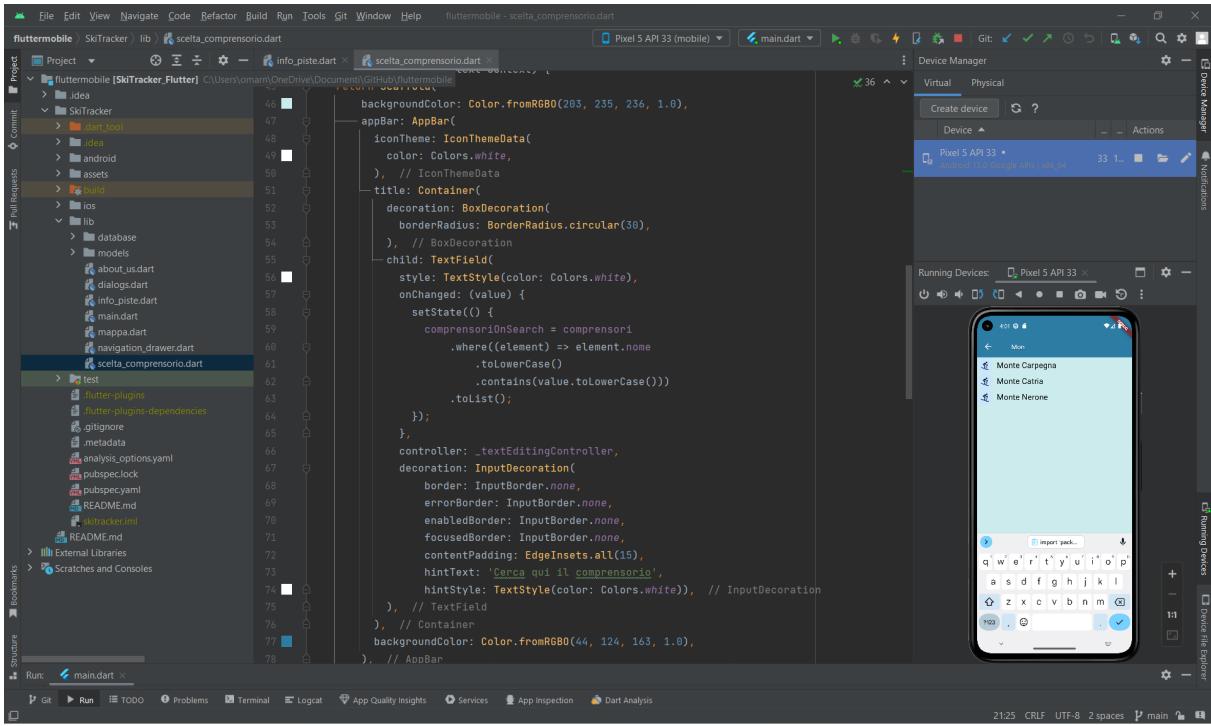


Figura 29: Pagina di scelta del compressorio ed esempio di uso della barra di ricerca.

3.6.5 About Us

Contenuto all'interno di `about_us.dart`, contiene semplicemente un Widget immagine (il logo dell'app) ed alcuni testuali che riportano informazioni sugli sviluppatori dell'app e sulla versione dell'app stessa.

4 Conclusioni

Il progetto è stato sviluppato usando due diversi approcci, quello nativo per lo sviluppo Android e quello *Cross-Platform*, che permette di realizzare applicazioni funzionanti in altri sistemi operativi e anche nello stesso PC, generando l'applicazione come app per Desktop o anche come servizio accessibile sul Web.

I due approcci sono molto diversi, ed in particolare hanno dei loro punti di forza e di debolezza. Ad esempio, l'applicazione sviluppata in Kotlin ha diversi vantaggi nella gestione dell'architettura e nella divisione dei ruoli, grazie al concetto di Activity e Fragment, mentre lo sviluppo in Flutter è stato più semplice in termini di UI, in quanto i vari componenti delle varie interfacce sono stati costruiti impiegando meno tempo e con più semplicità rispetto al corrispondente in Android Studio (basti solo confrontare la sezione "Informazioni sul comprensorio", in entrambe le versioni accessibile tramite Bottom Navigation Bar); inoltre per la gestione della programmazione asincrona, vi sono stati diversi approcci: in Flutter è quasi obbligatoria ma relativamente semplice da implementare e da gestire, mentre in Kotlin viene resa anche facoltativa, ma la sua eventuale implementazione risulta più complessa.

Non citata in precedenza, essa consente la gestione di più parti del progetto in parallelo. In sostanza, nella programmazione asincrona ogni istruzione o comando viene eseguito indipendentemente dal fatto che l'istruzione precedente sia conclusa. Ciò permette dei vantaggi sia in tempistiche che in funzionalità. Nel progetto sviluppato in Flutter, ad esempio, la incontriamo al momento della selezione del comprensorio. Nel mentre che viene scaricata la lista delle piste del comprensorio, si può osservare che nella parte alta dello schermo si attiva quella che, nel linguaggio comune, è chiamata "barra di caricamento".

In conclusione, entrambi gli approcci di sviluppo ed entrambi i progetti hanno permesso di studiare diversi metodi di implementazione di applicazioni per Smartphone, in particolar modo per Android, dandoci anche il modo di capire, nel pratico, come realizzare un qualcosa di concreto ed usufruibile da chiunque.