Alma Mater Studiorum - Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica per il Management

Personal Physical Tracker

Canghiari Matteo
Matricola 1032059
matteo.canghiari@studio.unibo.it

Laboratorio di Applicazioni Mobili
Anno Accademico 2023/2024

Introduzione

Personal Physical Tracker è un'applicazione nativa Android, che permette di registrare le attività motorie quotidiane che possano essere compiute durante la giornata, contraddistinte in **camminata**, **spostamenti tramite veicoli**, **attività sedentarie** e **corsa**. L'obiettivo dell'applicazione consiste nello sviluppo di un sistema software in grado di poter memorizzare dati riferiti alle attività compiute, favorendo un'interfaccia grafica semplice e di facile utilizzo, da cui sia possibile osservare attivamente le informazioni manipolate e rappresentate. L'insieme delle funzionalità saranno successivamente approfondite nella sezione **Feature**, tuttavia è possibile anticipare il contenuto che contraddistingue il progetto proposto.

Ad un primo avvio l'utente visualizza la schermata per la registrazione, in cui può effettuare il *Sign In* qualora dovesse essere un nuovo iscritto, oppure il *Login* in caso sia già registrato; infatti, è richiesta totale univocità pur di garantire la persistenza dei dati. Proseguendo, viene mostrato il layout principale dell'applicazione, responsabile dell'avvicendamento dei molteplici *Fragment* ospitati al suo interno, composto da una *Navigation Bottom Bar* e da un *Drawer*. Mediante la *Navigation Bar*, l'utente possiede la facoltà di **registrare manualmente** nuove attività, accedere al **pannello di controllo**, visualizzare i propri **progressi**, osservare le **geofence di interesse**, **pianificare** le proprie attività ed, infine, visualizzare le attività condivise da altri iscritti nella sezione **Group**.

Di seguito sono brevemente descritte alcune scelte implementative e strumenti utilizzati durante lo sviluppo: per il versionamento del codice è stato utilizzato **Git**, con repository accessibile su **GitHub**, mentre lo sviluppo dell'applicativo è stato condotto mediante il linguaggio di programmazione **Kotlin**. Il progetto si compone di quattro punti cardine per il corretto funzionamento, suddivisi in:

- Design pattern architetturale **MVVM**, utilizzato per separare il **Model**, ovvero il contenitore di dati, e la sezione attiva, la **View**. In questo modo si garantisce la separazione dei componenti, riducendo l'accoppiamento e le dipendenze funzionali, che potrebbero causare massicci problemi in seguito a piccole modifiche

- **Jetpack Compose**, un toolkit moderno utilizzato per velocizzare l'implementazione della *User Interface* in determinate circostanze; dato il suo approccio dichiarativo e di facile intuizione ha permesso di incrementare e migliorare l'interattività dell'applicazione per specifiche componenti

- **Room**, utilizzato per garantire la persistenza dei dati all'interno di un *Database Relazionale Locale*

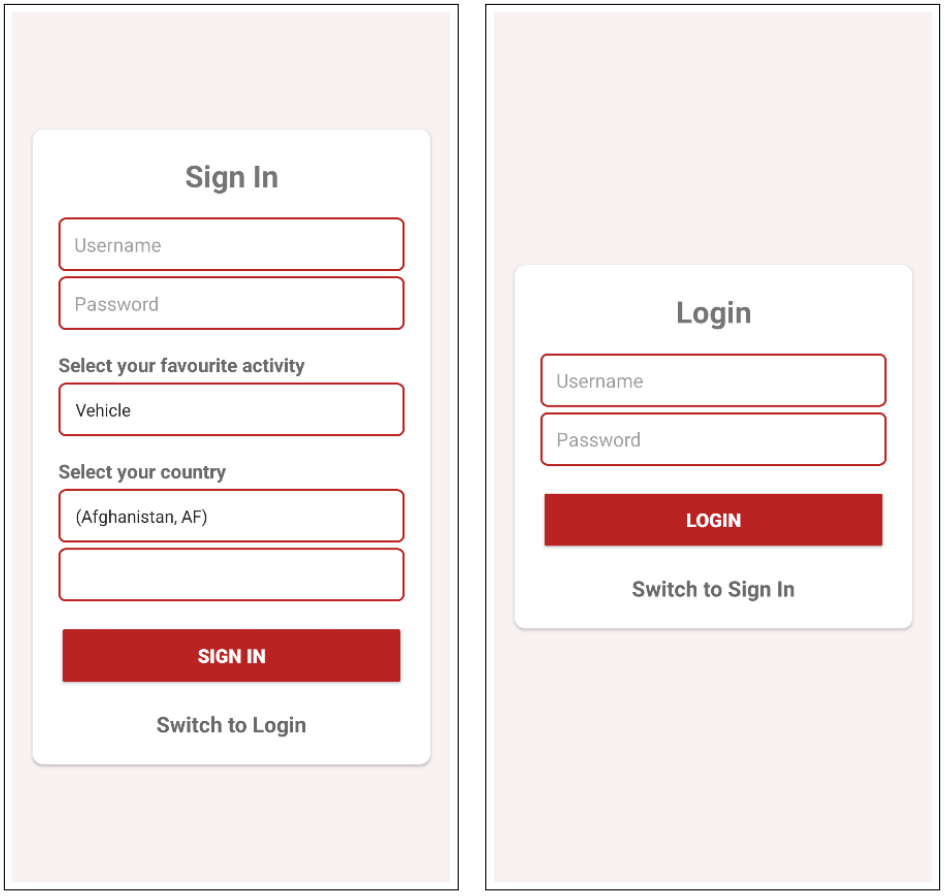
- **Supabase**, utilizzato per garantire un'ulteriore livello di persistenza dei dati, affinchè il *Database Locale* mantenga al suo interno un unico utente come riferimento, delegando al *Real Time Database* il compito di salvaguardare l'insieme di tutti i dati degli utenti iscritti

- **Amazon S3**, utilizzato per salvare in cloud i dati degli utenti tramite un **dump** del Database, consentendo la condivisione dell'informazioni memorizzate

Features

Registrazione

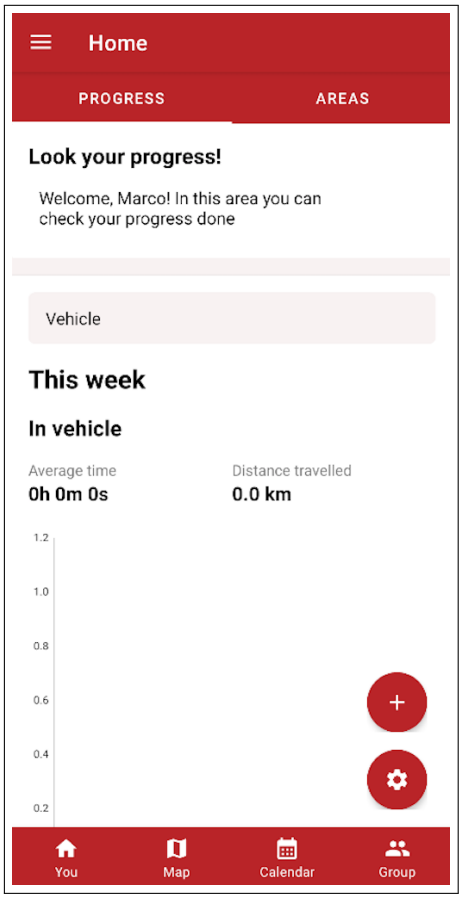
All'avvio dell'applicazione è richiesta la registrazione. Come già ribadito, è necessario che le credenziali siano univoche, per garantire la persistenza dei dati. Accettate e registrate quest'ultime, è richiesto il permesso di **Post Notification**, garantendo l'opportunità di inviare delle notifiche al dispositivo; in caso dovesse essere negato, è concesso ugualmente l'accesso. Tale feature richiede una *connessione ad Internet* stabile, per controllare l'unicità delle credenziali all'interno di *Supabase*.



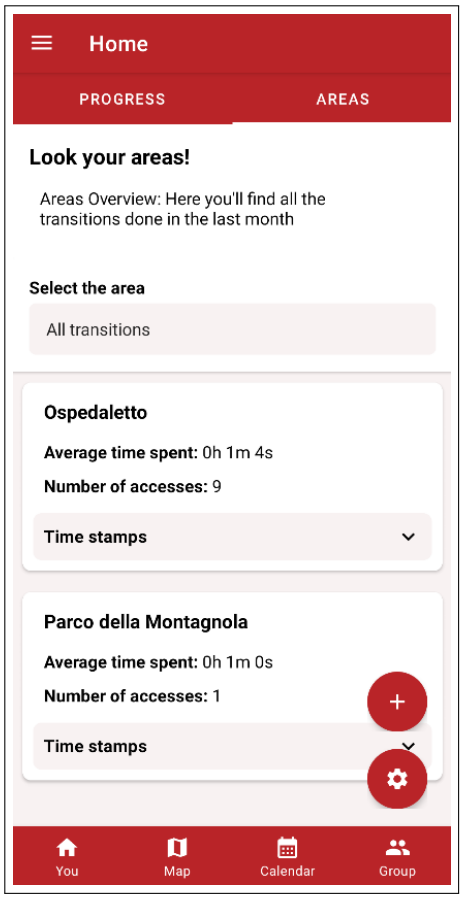
(a) Login Activity (b) Sign In Activity

Dashboard

Successivamente all'autenticazione, il primo layout presentato riguarda la sezione personale dell'utente. Mediante un *Tab Layout*, è possibile osservare i *progressi* ottenuti durante la registrazione delle attività motorie e le *transizioni* verificatesi durante lo stazionamento all'interno delle *Geofence*. La *Progress Page* possiede una *View* focalizzata sulla descrizione delle attività compiute attraverso l'ausilio di **grafici**, mentre la *Geofence Page* evidenzia le **transizioni** avvenute durante un determinato arco temporale.



(c) Page Progress

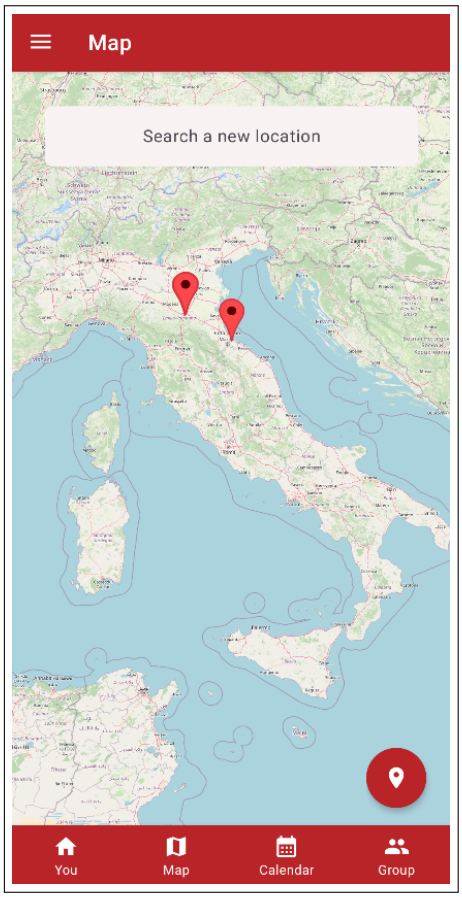


(d) Page Geofence

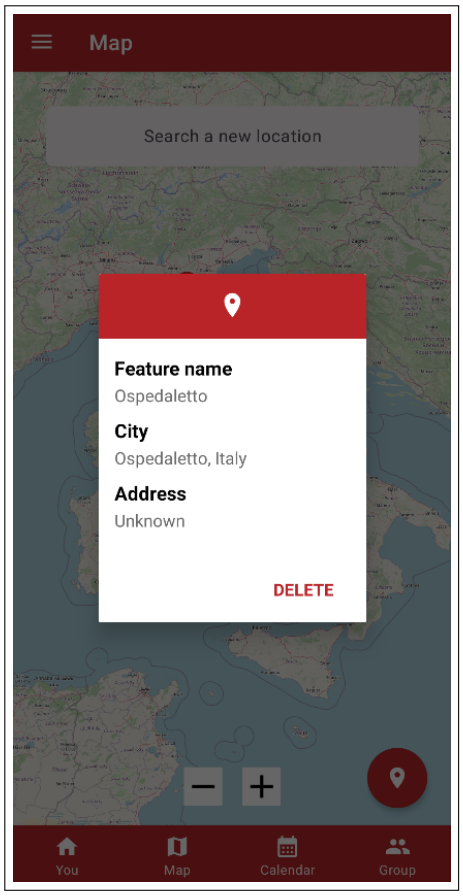
Inoltre, entrambi i *Fragment* ospitano due *Floating Button*; rispettivamente, il primo consente di *registrare manualmente* nuove attività, mentre il secondo permette l'accesso al *pannello di controllo*, in cui l'utente può selezionare le proprie preferenze, abilitando oppure disabilitando le funzionalità disponibili.

Mappa

La *View* contiene una **mappa** all'interno di un *MapView*. L'utente può inserire, visualizzare ed eliminare le *aree geografiche di interesse*; una zona di interesse è l'area compresa nel raggio di {150, 200, 250} metri, rispetto alle coordinate geografiche che definiscono il centro della circonferenza, varcata la soglia di una di esse il dispositivo riceve una notifica personalizzata. Premendo su uno dei molteplici *marker* a disposizione, è mostrato un *Dialog* contenente le informazioni principali della località, tra cui, il nome, la provincia e l'indirizzo.



(e) Map Fragment

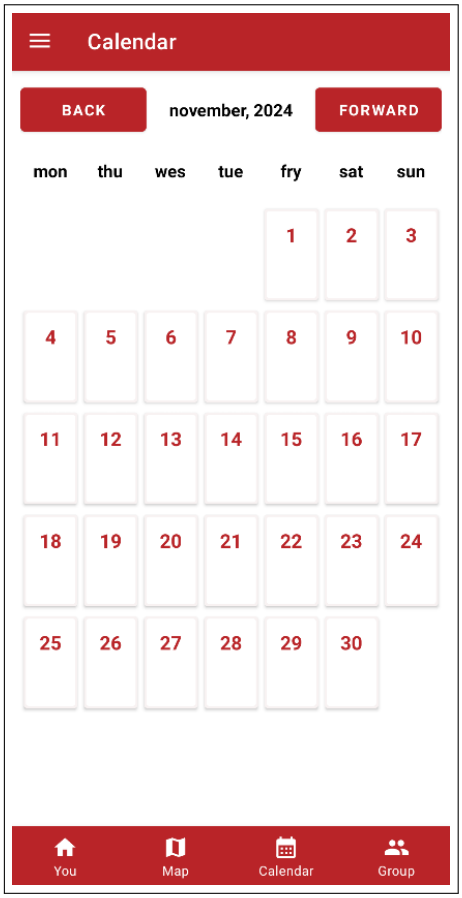


(f) Marker Dialog

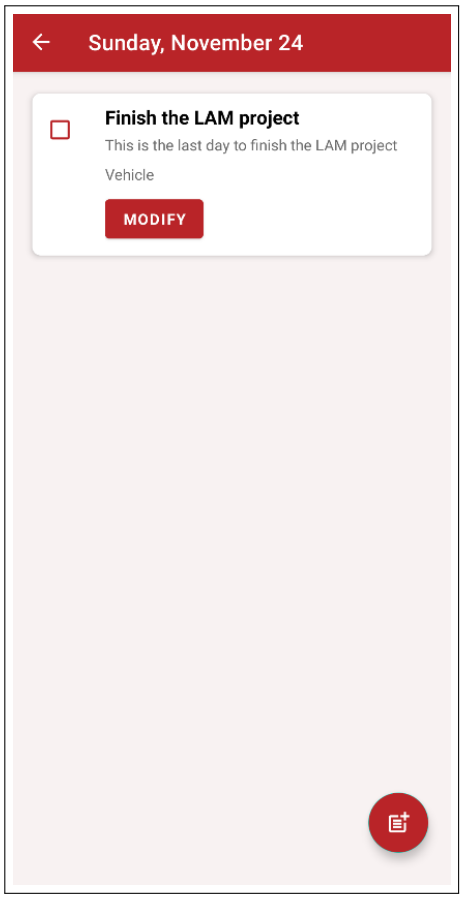
Concludendo, è presente un ulteriore *Floating Button*, visualizzabile solamente all'interno di questo *Fragment*, in grado di marcare la posizione corrente dell'utente; per riusciure nell'intento è necessario che sia stato autorizzato il permesso di **Access Fine Location**.

Calendario

L'utente utilizzando il **calendario** può organizzare le proprie attività. Cliccando su una delle celle, sono mostrati tutti i *Memo* correnti alla data selezionata. Ogni *Memo* possiede un titolo, una descrizione e una tipologia di attività associata. L'utilizzatore premendo sul *Floating Button* possiede l'opportunità di inserire nuovi promemoria; cliccando sul bottone riportato nello stesso componente, ha la possibilità di variarne il contenuto. Contrariamente, qualora siano conclusi alcuni promemoria, digitando sulle checkbox e raggiungendo l'*Activity* precedente, saranno automaticamente eliminati dal *Database Locale*. Quest'ultima azione scaturisce l'avvio di un determinato *Worker*, denominato *DeleteMemoWorker*. Tale componente, data la presenza di una *connessione ad Internet* stabile, consente di eliminare i *Memo* selezionati all'interno di *Supabase*.



(g) Calendar Fragment

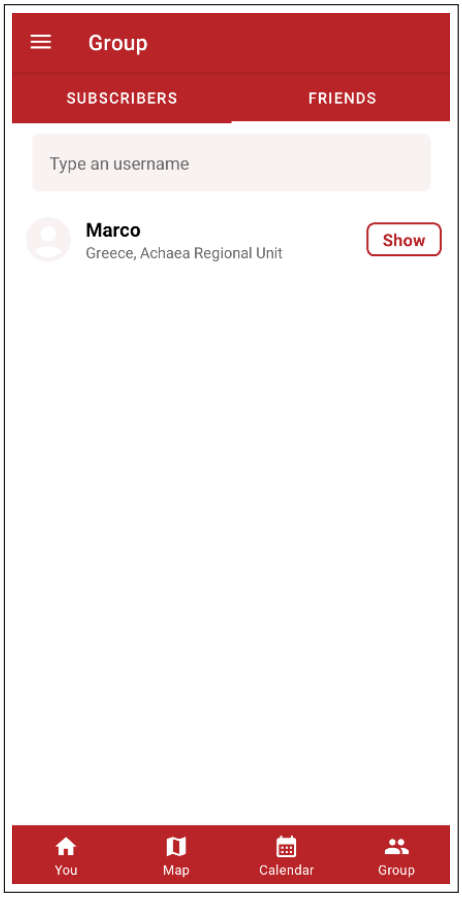


(h) Daily Memo Activity

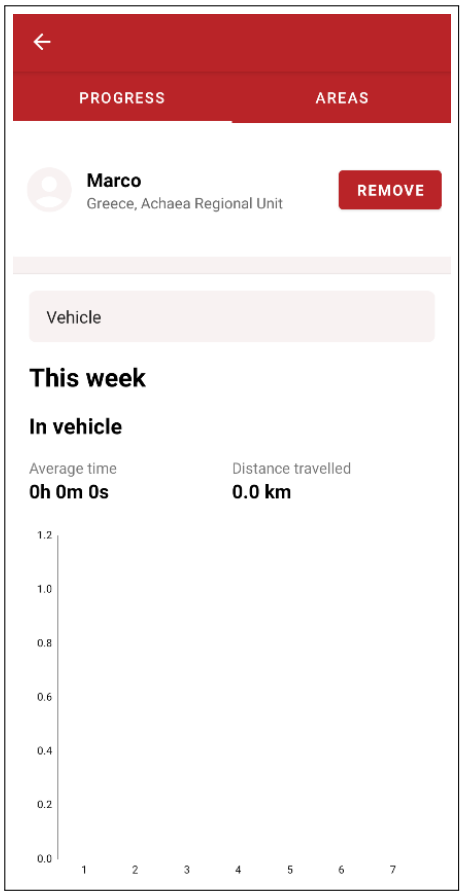
Nota bene: in assenza di Internet il *Worker* narrato viene momentaneamente sospeso, in attesa di una connessione, Wi-Fi oppure cellulare.

Gruppi

Tramite l'applicativo è possibile condividere le proprie attività e transizioni con altri *iscritti*, e visualizzare le stesse da loro compiute. Affinchè i dati di un ulteriore utente siano osservabili, occorre che abbia abilitato il *servizio di sincronizzazione*. Tale funzionalità è stata implementata mediante un *Worker*, denominato *SyncBucketWorker*. Si tratta di un **Worker periodico**, in cui definito un certo arco temporale di attesa, provvede a sincronizzare le informazioni memorizzate localmente rispetto al *dump* contenuto nel **Bucket** in cloud, agendo in totale autonomia.



(i) Group Fragment



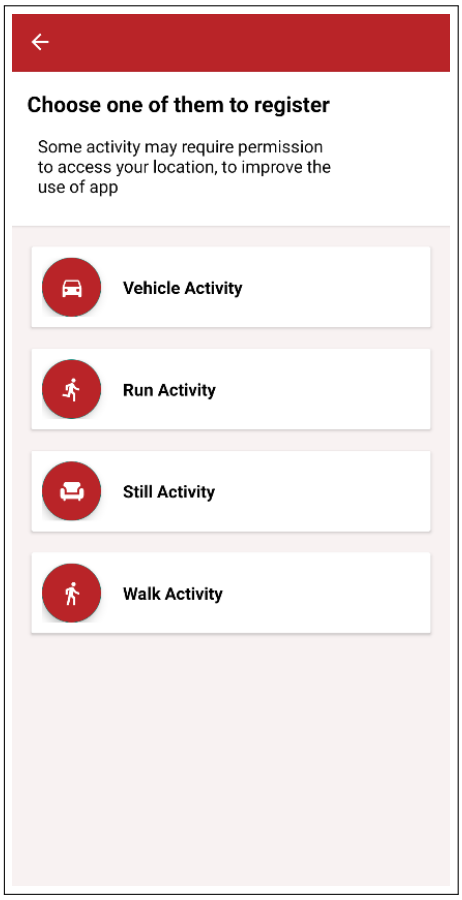
(j) Page Progress User

Nota bene: come accade per il *DeleteMemoWorker*, in assenza di connessione il *Worker* descritto viene sospeso.

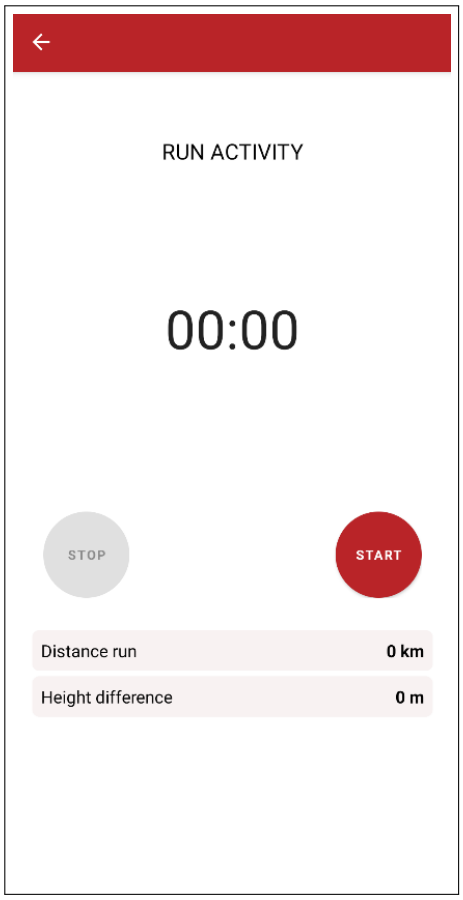
Registrazione attività

Come da immagine sottostante, l'utente può registrare quattro tipologie di attività, a cui sono associati alcuni sensori, suddivise in:

- **Vehicle Activity**, attività che ricadono nell'utilizzo di un qualche veicolo, in cui è memorizzata la distanza percorsa
- **Run Activity**, attività di corsa, in cui è rilevata la distanza sostenuta e il dislivello affrontato
- **Still Activity**, attività sedentaria, in questo specifico caso non è attuato alcun sensore
- **Walk Activity**, dove è memorizzato il numero di passi e il dislivello affrontato durante la camminata



(k) Manual Registration Activity



(l) Run Activity

Tutti i dati estrapolati vengono catalogati e salvati all'interno del *Database Locale*, per poi essere successivamente rielaborati per consentirne la corretta visualizzazione grafica. Alcuni dei sensori citati, richiedono che l'utente abbia preventivamente accettato il permesso di **Access Fine Location**, necessario per specificare la distanza percorsa durante uno spostamento tramite veicolo oppure durante una corsa. Inoltre, sono stati implementati ulteriori due sensori, rispettivamente il **Step Counter Sensor**, che, come da denominazione, permette di circoscrivere il numero di passi, e il **Sensore di Pressione**, adeguato per indicare il dislivello affrontato durante la rilevazione dell'attività.

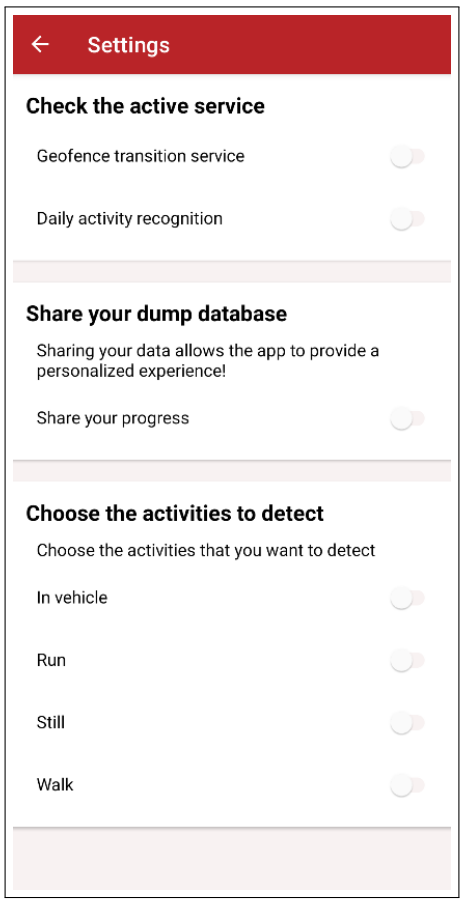
Pannello di controllo

Il **pannello di controllo** costituisce un riferimento univoco in cui l'utente può abilitare oppure disabilitare i servizi disponibili. Al tempo stesso, qualora alcuni di essi non siano mai stati attivati, saranno richiesti i permessi necessari pur di avviare i componenti desiderati. Pertanto, all'interno di tale *Activity* sono richiesti:

- **Activity Recognition**, permesso necessario per abilitare il riconoscimento automatico delle attività compiute dall'utente
- **Access Fine Location**, come già narrato, autorizzazione fondamentale per risalire alla geolocalizzazione corrente del dispositivo

- **Access Background Location**, accesso alla geolocalizzazione del dispositivo anche qualora l'applicazione non sia in uso

Rispettivamente, il primo permesso è necessario per avviare il *service di Activity Recognition*, mentre i restanti sono a loro volta necessari per abilitare il *service di Geofencing*.



(m) Settings Activity

Nota bene: il layout riporta nella sezione finale un breve catalogo di tutte le attività che il servizio è in grado di gestire e manipolare; in assenza di *Toggle* selezionati non sarà possibile avviare il riconoscimento automatico delle attività, anche qualora il *service* sia stato attivato.

Scelte implementative

Nella sezione seguente sono riportate le considerazioni adeguate durante il proseguimento del progetto. L'obiettivo del paragrafo consiste nella definizione e stesura degli aspetti cruciali che abbiano maggiormente inciso durante l'implementazione.

Model-View-ViewModel

Come già accennato dall'introduzione, uno dei punti cardine su cui si basa il progetto è il design pattern architetturale **Model-View-ViewModel**. Questo approccio separa la *View*, denominata entità attiva, dal *Model*, contenitore statico di dati, permettendone lo scambio. Il *ViewModel* è incaricato di mantenere aggiornate le strutture dati che possiedono informazioni del *Model*. Ciò avviene per garantire che tutti i soggetti il cui comportamento può variare in base al contenuto dei dati del *Model*, possano essere notificati dinnanzi ad una qualsiasi variazione. Comportamento reso possibile mediante l'impiego di **LiveData**, a loro volta basati sul concetto di *Observables*.

```
class NetworkViewModel: ViewModel() {

    private val _currentNetwork = MutableLiveData<Boolean>()
    val currentNetwork: LiveData<Boolean> get() = _currentNetwork

    init {
        _currentNetwork.postValue(MyNetwork.isConnected)
    }

    fun setNetwork(enabled: Boolean) {
        if (_currentNetwork.value != enabled) {
            _currentNetwork.postValue(enabled)
        }
    }
}
```

Lo snippet di codice precedente, definisce il metodo implementato per aggiornare tutte le componenti dell'applicazione che siano interessate allo stato corrente della connessione. All'interno delle classi è sviluppata una funzione *Observer* dedita ad acquisire i potenziali cambiamenti del *LiveData* circoscritto, abilitanto oppure disabilitando le feature che richiedano una connessione persistente.

Jetpack Compose

Jetpack Compose è un framework utilizzato per l'implementazione della *User Interface* di un'applicazione Android. Rappresenta uno strumento in grado di accelerare lo sviluppo dell'interfaccia grafica, concentrando in un solo componente, *Activity* o *Fragment*, tutto il codice necessario per la sua realizzazione.

Ogni funzione che si accinge a definire qualsiasi *View*, deve riportare la *keyword @Composable*.

```
@Composable
private fun DefineFloatingButton(description: String, icon: ImageVector, onClick:
    () -> Unit) {
    FloatingActionButton(
        onClick = {
            onClick()
```

```
        },
        containerColor = colorResource(id = R.color.uni_red),
        elevation = FloatingActionButtonDefaults.elevation(4.dp),
        shape = CircleShape
    ) {
         Icon(
            imageVector = icon,
            contentDescription = description,
            tint = Color.White
        )
    }
}
```

Il progetto proposto adotta un approccio *ibrido*, ossia impiega i layout realizzati mediante file con estensione *XML* assieme a specifici tag *ComposeView*, articolando i due paradigmi. La scelta di una composizione simile, ha facilitato lo sviluppo di tutti i componenti grafici attesi da un aspetto comportamentale, ovviando a potenziali dipendenze che avrebbero potuto causare differenti problematiche dinnanzi a modifiche necessarie.

Room

Per la gestione del *Model* è stato utilizzato **Room**. La libreria ha permesso di creare e di manipolare con estrema semplicità un *Database Relazionale*, memorizzando al suo interno tutte le informazioni estrapolate durante l'utilizzo dell'applicazione da parte dell'utente.

Dichiarata un'istanza della base di dati, è necessario definire le *Entity*, ossia le tabelle dedotte durante la stesura del *Modello E-R*, e i *DAO*, contenenti tutte le funzioni desiderate.

AWS

Per consentire agli utenti di condividere i propri progressi è stato utilizzato *AWS*. La scelta è ricaduta su di esso per le conoscenze pregresse e l'intuitiva implementazione garantita dalla SDK di Kotlin. Abilitato il servizio in questione, presente all'interno del *pannello di controllo*, è programmato un **Worker periodico**, incaricato di effettuare un *dump* della basi di dati in formato *JSON*, successivamente caricato all'interno di un *bucket* mediante una richiesta *AWS-S3Client*. In questo modo è possibile garantire totale autosufficienza della feature, poichè provvede autonomamente a memorizzare i record del database mediante l'ausilio del servizio esterno. Qualora l'utente decidesse di non condividere più i propri progressi, potrà semplicemente disabilitare la funzionalità dal *pannello di controllo*.

Background operations

Le operazioni in background sviluppate all'interno del progetto si articolano in tre sezioni differenti, così suddivise:

- **Activity Recognition**, deduzione automatica dell'attività fisica conseguita dall'utente. Dopo aver precedentemente stabilito quali attività siano di suo interesse, l'applicazione mediante un *Broadcast Receiver* e un *Service*, dedito alla memorizzazione dei dati estrapolati, provvede a notificare l'utilizzatore
- **Geofencing**, recinti virtuali attuati per contrassegnare località di interesse. A partire dalle coordinate geografiche, è realizzata un'area circolare avente un raggio prestabilito; qualora l'utente dovesse varcare la soglia di una di queste circonferenze, l'applicativo memorizza l'evento all'interno del Database Locale ed invia una notifica al dispositivo

- **Connectivity**, *Broadcast Receiver* registrato per acquisire tutti i cambiamenti incisivi di connessione che interessino il dispositivo

A livello implementativo, tutte le *background operations* riportate sono caratterizzate da un'architettura piuttosto simile. Ad ognuno di esse è stato associato un *Broadcast Receiver*, responsabile di intercettare gli intenti inviati dal sistema e di risvegliare i *Service* circoscritti. Per attivare i servizi, sono stati implementati dei *Worker*, incaricati di effettuare alcune azioni di *pre-processing* prima di richiamare la funzione *startService()*. Successivamente i *Service*, ottenuti i dati necessari, provvedono a salvare le informazioni dedotte all'interno della base di dati.

Nota a margine per quanto riguarda l'operazione *Connectivity*. Il servizio citato, oltre ad aggiornare lo stato corrente della connessione, permette di variare la dimensione delle *Geofence*. Ciò avviene mediante l'impiego del *Geofence Service*; una volta risvegliato, si accerta della tipologia di connessione corrente, e varia di conseguenza il perimetro dell'aree di interesse, in ottica di risparmio energetico e per garantire maggiore accuratezza.

```
class ActivityPreprocessingWorker(private val context: Context, workerParams:
    WorkerParameters): Worker(context, workerParams) {

    override fun doWork(): Result {
        val type = inputData.getInt("TYPE", 4)
        val transition = inputData.getInt("TRANSITION", -1)

        val serviceIntent = Intent(context,
            ActivityRecognitionService::class.java).apply {
            val items = if (transition == 0) {
                Pair(
                    ActivityRecognitionService.Actions.INSERT.toString(),
                    "ARRIVAL_TIME"
                )
            } else {
                Pair(
                    ActivityRecognitionService.Actions.UPDATE.toString(),
                    "EXIT_TIME"
                )
            }

            action = items.first
            putExtra("ACTIVITY_TYPE", type)
            putExtra(items.second, System.currentTimeMillis())
        }

        context.startService(serviceIntent)

        return Result.success()
    }
}
```