Physical Tracker

Laboratorio di applicazioni mobili

Gianluca Tondo
0001031952
gianluca.tondo@studio.unibo.it

Introduzione

Obbiettivo dell'applicazione

L'applicazione "Physical Tracker" è progettata per registrare le attività fisiche dell'utente in base alla modalità di movimento. Consente all'utente di avviare e interrompere manualmente il monitoraggio di diverse attività, tra cui camminare, correre, stare seduti, guidare, dormire e andare in bicicletta. Ogni attività viene registrata in termini di tempo trascorso e, per le attività che prevedono il movimento, come la camminata e la corsa, l'app raccoglie anche dati specifici, come il numero di passi effettuati.

Tramite l'interfaccia interattiva dell'applicazione, l'utente può visualizzare lo storico delle attività, con la possibilità di consultare i dati sotto forma di elenco o attraverso grafici riepilogativi. Inoltre, l'app offre una funzionalità avanzata in background, ovvero il monitoraggio della presenza dell'utente in specifiche aree geografiche tramite geofencing.

Tecnologie utilizzate

L'applicazione è stata sviluppata per dispositivi Android utilizzando l'IDE Android Studio e il linguaggio Kotlin. L'SDK utilizzata è la **SDK 34**, con una versione minima supportata di **SDK 24** e una versione target di **SDK 34**.

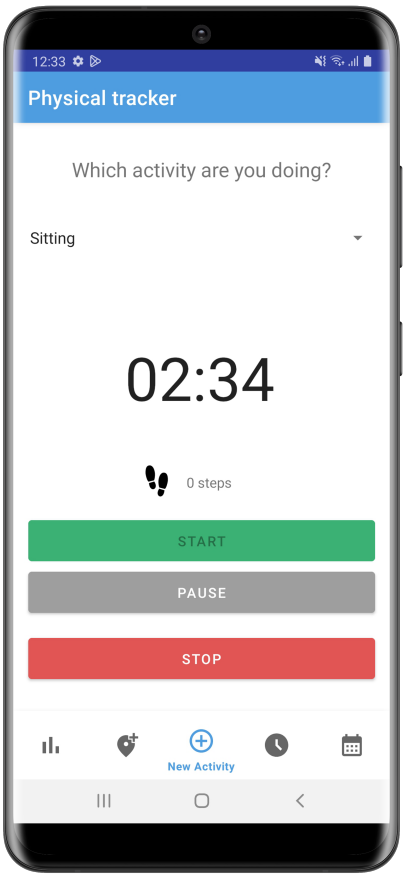
Per implementare i grafici è stata utilizzata la libreria **MPAndroidChart**, mentre per il Geofencing e le mappe è stata usata l'API **Google Maps** tramite il pacchetto **play-services-location**.

L'applicazione è stata testata su un **Samsung Galaxy A6+** con **Android 10** (API 29).

Descrizione Generale dell'Applicazione

Funzionalità principali

Le diverse funzionalità dell'applicazione sono accessibili all'utente tramite le schermate selezionabili dal **menù di navigazione** in basso.



Tra le varie funzionalità dell'applicazione, la principale è la registrazione delle attività fisiche, gestita attraverso la schermata **New Activity**, che viene selezionata automaticamente all'avvio dell'app. In questa schermata, l'utente può scegliere l'attività che sta svolgendo tramite uno spinner, che offre le seguenti opzioni: **Walking**, **Running**, **Sitting**, **Driving**, **Sleeping** e **Bicycle**.

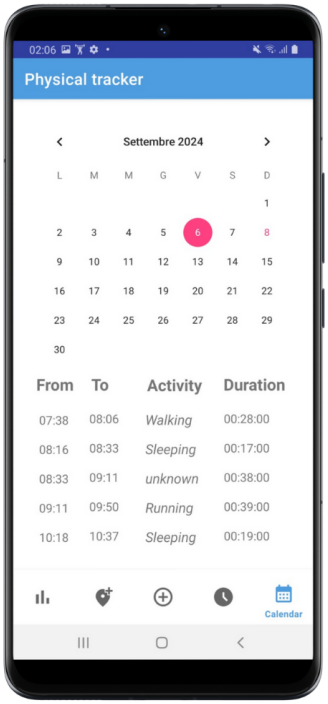
Una volta selezionata l'attività, l'utente può avviare il cronometro premendo il pulsante **Start**. Dopo l'avvio, l'utente è libero di navigare verso altre schermate, uscire dall'app o bloccare il telefono, mentre l'attività continua a essere monitorata in background. Quando l'attività è conclusa, premendo il pulsante **Stop**, l'utente può salvare i dati relativi all'attività svolta.

La schermata include anche un pulsante per **sospendere** e **riprendere** l'attività in caso di interruzioni o imprevisti. I pulsanti sono abilitati o disabilitati in base allo stato del cronometro, ad esempio: il pulsante **Start** non è disponibile mentre il cronometro è in funzione, e il pulsante **Stop** è disattivato quando il cronometro non è avviato.

Inoltre, se l'attività selezionata è **Walking** o **Running**, il **contatore di passi** presente al di sotto del cronometro viene aggiornato automaticamente durante il monitoraggio dell'attività.

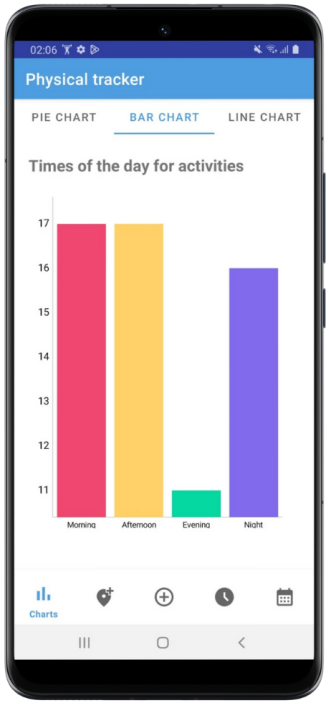
Nella schermata **History**, l'utente può visualizzare tutte le attività salvate, ordinate cronologicamente dalla più recente alla più vecchia. È possibile applicare **filtri** per visualizzare solo le attività di una determinata tipologia selezionandola tramite lo spinner posizionato in alto a sinistra. Inoltre, è possibile **ordinare** i risultati in base alla durata in ordine decrescente, facilitando l'individuazione delle attività più lunghe, utilizzando il pulsante presente in alto a destra.

Nella schermata **Calendar**, l'utente può visualizzare le attività svolte in una giornata specifica, selezionata tramite il **calendario** posizionato in alto. Le attività sono visualizzate in ordine cronologico, a partire dall'inizio della giornata.

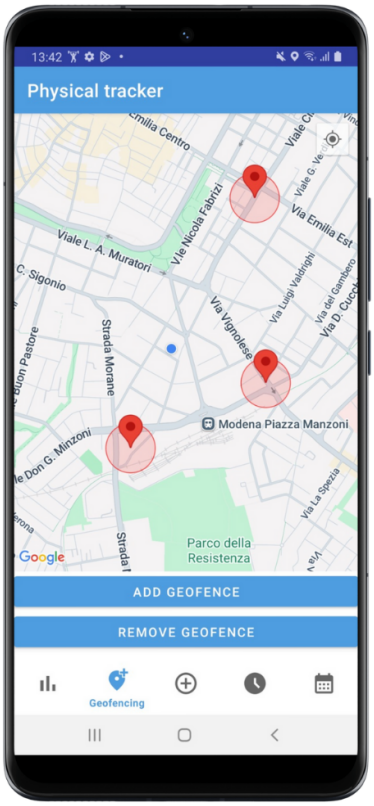


Nella schermata **Charts**, l'utente può visualizzare le statistiche sulle attività svolte attraverso tre diversi tipi di grafici:

1. Un **grafico a torta** che rappresenta le attività in base alla loro **durata totale**.
2. Un **grafico a barre** che mostra il numero di attività svolte in quattro diversi periodi della giornata: **morning**, **afternoon**, **evening** e **night**.
3. Un **grafico a linee** che illustra il numero di **passi fatti** durante i giorni della settimana.



Nella schermata dedicata al **Geofence**, l'utente può inserire un **marker** sulla mappa e, tramite il pulsante **"Add Geofence"**, è possibile memorizzare un'area di interesse a cui viene associato un nome inserito dall'utente. Quando l'utente entra o esce da quest'area, riceverà una notifica che lo avvisa della transizione. È possibile rimuovere un Geofence cliccando sul marker corrispondente e premendo il pulsante **"Remove Geofence"**.



L'applicazione include anche una funzionalità che invia notifiche di **promemoria periodiche**. Se l'utente non sta registrando alcuna attività, riceverà un **reminder ogni ora** che lo invita a iniziare il monitoraggio di un'attività, aiutando così a mantenere traccia costante delle attività fisiche.

Dettagli di implementazione

Memorizzazione dei dati tramite Room e gestione delle attività mancanti

Per la memorizzazione delle attività e dei geofence, l'applicazione utilizza **Room**, una libreria di persistenza che permette di gestire un database locale SQLite in modo semplice ed efficiente.

L'implementazione di Room si basa su tre componenti principali:

- **Entità**, che rappresentano i dati da memorizzare.
- **DAO (Data Access Object)**, che forniscono i metodi per eseguire operazioni sul database tramite query.
- **Database**, che funge da punto di accesso principale per l'archiviazione e il recupero dei dati.

Nel caso dell''applicazione, vengono incluse due principali entità:

1. **ActivityEntity** (definita in `ActivityEntity.kt`):
   Questa entità rappresenta le attività fisiche dell'utente. Ogni attività è caratterizzata da:
   - Un **id** (chiave primaria autogenerata).
   - Il **tipo** di attività (camminare, correre, sedersi, ecc.).
   - La **durata** dell'attività.
   - Il **numero di passi**, che viene registrato solo per le attività di tipo "Walking" e "Running" (null per le altre attività).
   - Il **tempo di inizio** e **tempo di fine** dell'attività.
   - La **data di inizio**.

2. **GeofenceEntity** (definita in `GeofenceEntity.kt`):
   Questa entità rappresenta i geofence impostati dall'utente. Ogni geofence è caratterizzato da:
   - Un **id** (chiave primaria autogenerata).
   - Un **geofenceId**, che identifica univocamente il geofence.
   - Le **coordinate di latitudine** e **longitudine**.
   - Il **raggio** dell'area di interesse.
   - Il **tipo di transizione** da monitorare (sia entrata e che uscita).
   - Un **nome** dato dall'utente per identificare l'area.

Ogni entità ha il proprio DAO che gestisce le operazioni di inserimento, aggiornamento e recupero dei dati dal database:

- **ActivityDao** (definito in `ActivityDao.kt`):
  Contiene i metodi per inserire nuove attività, recuperare tutte le attività o filtrare le attività per tipologia e data.

- **GeofenceDao** (definito in `GeofenceDao.kt`):
  Fornisce i metodi per aggiungere nuovi geofence, eliminare quelli esistenti e recuperare tutti i geofence memorizzati.

Il database viene definito nella classe `AppDatabase.kt` ed è il punto di accesso principale per ottenere le istanze dei DAO.

```kotlin
fun getDatabase(context: Context): AppDatabase {
    val tempInstance = INSTANCE

    //Use of Singleton
    if (tempInstance != null) {
        return tempInstance
    }
    synchronized( lock: this) {
        val instance = Room.databaseBuilder(
            context.applicationContext,
            AppDatabase::class.java,
             name: "app_database"
        ).fallbackToDestructiveMigration().build()
        INSTANCE = instance
        return instance
    }
}
```

Questa classe segue il pattern **Singleton** per garantire che venga creata una sola istanza del database durante l'intero ciclo di vita dell'applicazione.

La funzione `getDatabase()` verifica se esiste già un'istanza del database e, in caso contrario, la crea all'interno di un blocco sincronizzato per evitare problemi di concorrenza tra thread.

In combinazione con Room è stato implementato il pattern **MVVM** (Model-View-ViewModel) per garantire una separazione chiara delle responsabilità tra la logica di business e l'interfaccia utente.

Per entrambe le entità, è stato creato un **repository** che funge da intermediario tra i DAO e i ViewModel. Il repository ha il compito di fornire un'astrazione per la gestione dei dati, centralizzando le operazioni di accesso e aggiornamento al database.

All'interno del `ActivityRepository.kt`, è stata implementata una logica specifica per gestire l'inserimento automatico di un'attività denominata **"unknown"** quando non viene registrata nessuna attività per un certo periodo di tempo. Questa funzionalità viene attivata sia all'avvio dell'applicazione sia ogni volta che viene inserita una nuova attività.

La funzione scorre tutte le attività presenti nel database e, se rileva che il **gap** tra l'orario di fine di un'attività e l'orario di inizio della successiva è superiore a 30 minuti, inserisce una nuova attività "unknown". Questo assicura che ci sia sempre una continuità temporale nel registro delle attività dell'utente, anche nei periodi in cui non viene svolta alcuna attività specifica.

```kotlin
suspend fun insertUnknownActivities() {
    val activities = activityDao.readAllDataSortedByStartTime()

    for (i in 0 ≤ until < activities.size - 1) {
        val currentActivity = activities[i]
        val nextActivity = activities[i + 1]

        val endTime = currentActivity.endTime ?: continue
        val startTimeNext = nextActivity.startTime


        // Gap between two activities
        val gap = startTimeNext - endTime

        // Insert Unkonwn if gap > 30mins
        if (gap > TimeUnit.MINUTES.toMillis( duration: 30)) {
            val unknownActivity = ActivityEntity(
                type = "unknown",
                duration = gap,
                startTime = endTime,
                endTime = startTimeNext,
                date = endTime
            )
            activityDao.addActivity(unknownActivity)
        }
    }

}
```

Anche i **ViewModel** sono stati creati per entrambe le entità e hanno il compito di fornire i dati alla UI, esponendo i flussi di dati tramite **LiveData**, in modo che l'interfaccia utente possa osservare i cambiamenti. Usando **Dispatchers.IO**, le operazioni di interazione con il database vengono spostate su un thread in background, mantenendo il thread principale libero di gestire le interazioni con l'utente e aggiornare l'interfaccia grafica in modo fluido.

Registrazione tramite cronometro e monitoraggio dei passi

La logica per la registrazione di una nuova attività è gestita all'interno del `RecordFragment.kt`, che rappresenta l'interfaccia dove l'utente può avviare, mettere in pausa e fermare il cronometro per tracciare il tempo di una specifica attività fisica.

Il componente centrale utilizzato per la registrazione del tempo è il **Chronometer** di Android, che tiene traccia del tempo trascorso dall'inizio dell'attività selezionata dall'utente. Il cronometro può essere avviato, messo in pausa e fermato, e ciascuno di questi eventi viene gestito in apposite funzioni che interagiscono con le variabili memorizzate nel **ViewModel**. Queste variabili includono:

- **chronometerBase**: la base del cronometro, che rappresenta il momento in cui è stato avviato o ripreso.
- **pauseOffset**: il tempo durante il quale il cronometro è stato in pausa.

Nelle varie funzioni, queste variabili vengono utilizzate per calcolare correttamente il tempo corrente del cronometro, basandosi sulle differenze di tempo tra il momento attuale e il momento della pausa, in modo da mantenere la precisione nel conteggio del tempo.

La variabile **isChronometerRunning** svolge un ruolo importante nel determinare lo stato del cronometro:

- Viene utilizzata nel pulsante **Pausa/Riprendi** per decidere quale azione eseguire (se mettere in pausa o riprendere il cronometro).
- Viene verificata nel metodo **onViewCreated**, per assicurarsi che, nel caso in cui l'utente abbia cambiato schermata mentre il cronometro era in esecuzione, lo stato del cronometro possa essere ripristinato correttamente senza perdita di informazioni.

Inoltre, lo stato del cronometro viene salvato nelle **SharedPreferences** per renderlo accessibile al `NotificationWorker.kt`, poiché quest'ultimo non può accedere direttamente al **ViewModel**. Questo approccio permette di evitare che vengano inviate notifiche periodiche quando il cronometro è attivo, in modo da non disturbare l'utente mentre sta registrando un'attività.

```
private fun saveChronometerState(isRunning: Boolean) {
    val sharedPreferences = requireContext().getSharedPreferences( name: "activity_tracker_prefs", Context.MODE_PRIVATE)
    with(sharedPreferences.edit()) { this: SharedPreferences.Editor!
        putBoolean("is_chronometer_running", isRunning)
        apply()
    }
}
```

Il conteggio dei passi viene gestito solo per le attività di “Walking” e “Running”, utilizzando il sensore di conteggio passi di Android, noto come **Step Counter Sensor** (`Sensor.TYPE_STEP_COUNTER`). Questo sensore viene inizializzato tramite il **SensorManager** nel metodo *onViewCreated*.

Quando l'utente seleziona una delle attività che richiedono il conteggio dei passi, il sensore viene attivato registrando un **SensorEventListener**, che ascolta i cambiamenti nel conteggio dei passi e aggiorna il valore in tempo reale.

Ogni volta che il valore del contatore di passi cambia, il metodo **onSensorChanged** viene chiamato e aggiorna i passi attuali. Alla prima lettura, il sensore fornisce il numero totale di passi fatti dal momento in cui il dispositivo è stato acceso. Per ottenere il numero di passi relativi all'attività in corso, il valore iniziale viene memorizzato e viene sottratto dal valore corrente per calcolare i passi effettivi dell'attività selezionata.

```
private val sensorEventListener = object : SensorEventListener
    @SuppressLint("SetTextI18n")
    override fun onSensorChanged(event: SensorEvent) {
        if (event.sensor.type == Sensor.TYPE_STEP_COUNTER) {
            if (steps == 0) {
                steps = event.values[0].toInt()
            }
            currentSteps = event.values[0].toInt() - steps
            tvStepsCounter.text = "$currentSteps steps"
        }
    }
```

Visualizzazione dei dati su Calendar e History con filtri

L'utente può visualizzare le attività salvate nel database tramite le schermate History e Calendar. Nel **HistoryFragment**, l'utente ha la possibilità di visualizzare la lista delle attività in ordine cronologico. Questa lista viene presentata all'interno di un **RecyclerView**, che utilizza un adattatore personalizzato, **ListAdapter**, per visualizzare ogni attività come una riga della lista.

In questa schermata, è possibile filtrare le attività in base al tipo e ordinarle per durata in ordine decrescente. Questi filtri e ordinamenti sono gestiti utilizzando i metodi del **ViewModel**, combinati con gli **Observer** per monitorare i dati in tempo reale. Le diverse query eseguite tramite l'**ActivityDao** consentono di applicare i filtri e gli ordinamenti richiesti, aggiornando automaticamente la lista visualizzata ogni volta che i dati cambiano.

Nel **CalendarFragment**, l'utente ha la possibilità di visualizzare le attività registrate in una specifica giornata, selezionata tramite il **CalendarView** di Android. La lista delle attività del giorno scelto viene visualizzata utilizzando una **RecyclerView** in combinazione con un adattatore personalizzato, il **CalendarAdapter**.

L'utente può interagire con il CalendarView per selezionare una data specifica. Quando la data cambia, viene eseguita una query che recupera dal database le attività registrate in quel giorno. Questo avviene attraverso il **ViewModel** e l'**ActivityDao**, che forniscono le attività filtrate in tempo reale utilizzando le query predefinite per ottenere le attività comprese tra l'inizio e la fine della giornata selezionata.

Ogni volta che viene selezionata una data nel CalendarView, la funzione *fetchActivitiesForDate()* viene chiamata per ottenere le attività relative a quel giorno. Questa funzione utilizza due metodi (*getStartOfDayInMillis()* e *getEndOfDayInMillis()*) per calcolare i millisecondi di inizio e fine giornata. Questi valori vengono poi passati alla query *getActivitiesByDate()* definita nel ActivityDao, che restituisce le attività nel range di tempo selezionato.

Visualizzazione dei grafici

La visualizzazione dei grafici nel **ChartsFragment** è gestita tramite una combinazione di **TabLayout** e **ViewPager2**. L'utente può navigare tra tre grafici (torta, barre, linee) selezionando la relativa scheda o scorrendo orizzontalmente.

Il ViewPager2 contiene tre fragment separati, uno per ciascun grafico, gestiti dall'**adapter** (`ChartsPagerAdapter.kt`). Il **TabLayoutMediator** collega il TabLayout al ViewPager2, aggiornando automaticamente le schede in base alla pagina visualizzata.
Questo approccio consente di passare facilmente tra i grafici e di gestire ogni grafico in un fragment distinto.

Il **grafico a torta** rappresenta la durata totale di ogni attività registrata dall'utente ed è implementato nel `PieChartFragment.kt`.

La funzione ***setUpPieChart()*** raccoglie i dati delle attività salvate nel database, forniti dal **ViewModel** tramite il metodo *readAllData*. Questi dati vengono osservati in tempo reale, permettendo al grafico di aggiornarsi automaticamente in caso di modifiche.

Per ciascuna attività, la funzione somma la durata totale di ogni tipo di attività utilizzando una mappa, in cui la chiave rappresenta il tipo di attività (ad esempio, "Walking", "Running") e il valore è la durata totale di quella specifica attività. Questa aggregazione consente di ottenere una distribuzione complessiva del tempo dedicato a ciascun tipo di attività.

Una volta popolata la mappa, i dati vengono convertiti in oggetti **PieEntry**, ognuno dei quali rappresenta una sezione del grafico a torta, corrispondente a un'attività e alla sua durata. Infine, il grafico viene personalizzato modificando attributi come la visualizzazione della leggenda, le dimensioni del testo e il formato dei valori.

```
34      private fun setUpPieChart(activities: List<ActivityEntity>) {
35          val entries = mutableListOf<PieEntry>()
36          val activityDurationMap = mutableMapOf<String, Long>()
37
38          activities.forEach { activity ->
39              val currentDuration = activityDurationMap.getOrDefault(activity.type,  defaultValue: 0L)
40              activityDurationMap[activity.type] = currentDuration + activity.duration
41          }
42
43          activityDurationMap.forEach { (type, totalDuration) ->
44              entries.add(PieEntry(totalDuration.toFloat(), type))
45          }
```

Il **grafico a barre** nel `BarChartFragment.kt` visualizza il numero di attività svolte in diversi periodi della giornata: mattina, pomeriggio, sera e notte.

La funzione ***setUpBarChart()*** raccoglie i dati delle attività salvate nel database e li aggrega in base al periodo del giorno in cui sono iniziate.
Per suddividere le attività, viene utilizzata la funzione ***getPeriodOfDay()***, che converte l'ora di inizio di ogni attività in uno dei quattro periodi: **Morning** (mattina), **Afternoon** (pomeriggio), **Evening** (sera) e **Night** (notte).

Una volta popolata la mappa con il conteggio delle attività per ciascun periodo, i dati vengono convertiti in oggetti **BarEntry**, ognuno dei quali rappresenta una barra nel grafico. L'asse X è personalizzato per visualizzare i periodi della giornata (ad esempio, "Morning", "Afternoon"), mentre l'asse Y mostra il numero totale di attività.

```
89      private fun getPeriodOfDay(startTime: Long): String {
90          val calendar = Calendar.getInstance()
91          calendar.timeInMillis = startTime
92
93          val hourOfDay = calendar.get(Calendar.HOUR_OF_DAY)
94          return when (hourOfDay) {
95              in 5 ≤ .. ≤ 11 -> "Morning"
96              in 12 ≤ .. ≤ 17 -> "Afternoon"
97              in 18 ≤ .. ≤ 21 -> "Evening"
98              else -> "Night"
99          }
100     }
```

Il `LineChartFragment.kt` implementa un **grafico a linee** che mostra il numero di passi giornalieri compiuti durante la settimana corrente. La funzione ***setUpLineStepsChart()*** raccoglie i dati delle attività salvate nel database e filtra solo le attività di tipo “Walking” e “Running”, poiché sono le uniche che registrano i passi.

I passi totali per ciascun giorno della settimana (da lunedì a domenica) vengono memorizzati in una mappa, **stepsByDay**, e la settimana corrente viene calcolata utilizzando la classe Calendar. Successivamente, le attività vengono filtrate per includere solo quelle svolte durante la settimana e l'anno corrente. Per ogni attività valida, i passi vengono sommati al totale del rispettivo giorno della settimana all'interno della mappa.

Una volta raccolti i dati, questi vengono convertiti in oggetti **Entry**, che rappresentano i punti da tracciare nel grafico a linee. Il grafico viene quindi personalizzato: la linea è colorata e ha uno spessore di 2px, mentre ogni punto del grafico è indicato da un cerchio colorato. L'asse X è configurato per mostrare i giorni della settimana (ad esempio, "Sun", "Mon"), mentre l'asse Y visualizza il numero totale di passi.

```
54          activities.filter { it.type == "Walking" || it.type == "Running" }.forEach { activity ->
55              calendar.timeInMillis = activity.date
56              val activityWeek = calendar.get(Calendar.WEEK_OF_YEAR)
57              val activityYear = calendar.get(Calendar.YEAR)
58              if (activityWeek == currentWeek && activityYear == currentYear) {
59                  val dayOfWeek = calendar.get(Calendar.DAY_OF_WEEK)
60                  stepsByDay[dayOfWeek] = stepsByDay.getOrDefault(dayOfWeek, defaultValue: 0f) + (activity.steps?.toFloat() ?: 0f)
61              }
62          }
```

Permessi dell'Applicazione

L'applicazione richiede l'accesso a diversi permessi per garantire il corretto funzionamento delle sue funzionalità:

- **Activity Recognition**: utilizzato per accedere ai sensori del dispositivo, come il contapassi, per monitorare le attività fisiche dell'utente.
- **Post Notifications**: richiesto per inviare notifiche all'utente, come i promemoria periodici per ricordare di registrare le attività.
- **Access Background Location, Coarse Location e Fine Location**: necessari per il funzionamento del geofencing, che rileva la posizione del dispositivo.

Al primo avvio dell'applicazione, viene eseguita la funzione ***requestPermission()***, che verifica quali permessi sono già stati concessi e quali devono essere richiesti attraverso funzioni dedicate. Se un permesso non è stato ancora accettato, viene aggiunto a un array, che successivamente viene utilizzato per inviare la richiesta all'utente tramite la funzione ***ActivityCompat.requestPermissions***().

```
90      private fun requestPermission(){
91          var permissionTORequest = mutableListOf<String>()
92          if(!hasActivityRecognitionPermission()){
93              permissionTORequest.add(Manifest.permission.ACTIVITY_RECOGNITION)
94          }
95
96          if(!hasNotificationPermission()){
97              permissionTORequest.add(Manifest.permission.POST_NOTIFICATIONS)
98          }
99
100         if(!hasAccessBackgroundLocation()){
101             permissionTORequest.add(Manifest.permission.ACCESS_BACKGROUND_LOCATION)
102         }
103
104         if(!hasAccessCoarseLocation()){
105             permissionTORequest.add(Manifest.permission.ACCESS_COARSE_LOCATION)
106         }
107
108         if(!hasAccessFineLocation()){
109             permissionTORequest.add(Manifest.permission.ACCESS_FINE_LOCATION)
110         }
111
112         if(permissionTORequest.isNotEmpty()){
113             ActivityCompat.requestPermissions( activity: this, permissionTORequest.toTypedArray(),  requestCode: 0)
114         }
115     }
```

La funzione ***onRequestPermissionsResult()*** gestisce il risultato della richiesta dei permessi. Ogni volta che un permesso viene concesso o negato, l'app registra l'esito e, in caso di concessione, mostra un log o un messaggio di conferma.

Alcuni permessi richiedono versioni specifiche dell'API di Android, come Activity Recognition e Background Location disponibile a partire da **API 29 (Android 10)** e Post Notifications richiesto solo per dispositivi con **API 33 (Android 13)**.

Notifiche dall'Applicazione

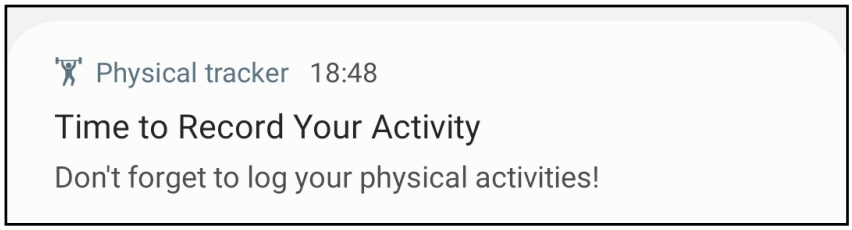
L'applicazione invia notifiche periodiche per ricordare all'utente di registrare una nuova attività. Questo sistema viene inizializzato nella MainActivity, dove viene chiamato il metodo ***createNotificationChannel()*** al momento della creazione dell'app. Questo metodo crea un **Notification Channel**, che permette di categorizzare le notifiche e definire proprietà come la priorità e il comportamento.

Le notifiche vengono programmate per essere inviate ogni ora tramite **WorkManager**, una libreria che gestisce operazioni in background anche quando l'app è chiusa o il dispositivo viene riavviato. Nella MainActivity, viene utilizzato il **PeriodicWorkRequestBuilder** per creare una richiesta che esegue il task ogni ora. Questo task corrisponde alla funzione ***doWork()*** della classe **NotificationWorker**.

```
132    private fun createNotificationChannel(context: Context) {
133        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
134            val channelId = "activity_reminder_channel"
135            val channelName = "Activity Reminder"
136            val importance = NotificationManager.IMPORTANCE_DEFAULT
137            val channel = NotificationChannel(channelId, channelName, importance).apply { this: NotificationChannel
138                description = "Channel for activity reminders"
139            }
140
141            val notificationManager: NotificationManager =
142                context.getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
143            notificationManager.createNotificationChannel(channel)
144        }
145    }
146
147    private fun schedulePeriodicNotification() {
148        val workRequest = PeriodicWorkRequestBuilder<NotificationWorker>( repeatInterval: 1, TimeUnit.HOURS)
149            .build()
150
151        WorkManager.getInstance( context: this).enqueue(workRequest)
152    }
```

La classe **NotificationWorker**, che estende la classe Worker di WorkManager, contiene la logica per controllare lo stato del cronometro (se l'utente sta registrando o meno un'attività). La notifica viene inviata solo se il cronometro non è attivo e questo controllo avviene leggendo lo stato del cronometro dalle **SharedPreferences**.

Se il cronometro non è in funzione, la funzione **doWork()** si occupa di inviare una notifica all'utente personalizzandola con un testo ed un'icona adatta.



Geofencing

Il geofencing è una funzionalità che consente di definire aree geografiche (geofence) e monitorare quando un dispositivo entra o esce da queste aree. Nell'applicazione, il geofencing è implementato all'interno del GeofenceFragment e tramite il GeofenceBroadcastReceiver.

Il **GeofenceFragment** rappresenta l'interfaccia in cui l'utente può interagire con una mappa, selezionare una posizione e aggiungere o rimuovere geofence. La mappa viene inizializzata tramite **SupportMapFragment**, e nel metodo **onMapReady()**, che viene chiamato quando la mappa è pronta, viene verificata la disponibilità dei permessi di localizzazione. Questi permessi sono necessari affinché la mappa possa mostrare la posizione attuale dell'utente e consentire l'uso delle funzioni di geolocalizzazione.

Se la posizione corrente è disponibile, viene utilizzato il servizio **FusedLocationProviderClient** per ottenere l'ultima posizione nota del dispositivo. Una volta ottenuta, la mappa viene automaticamente centrata su questa posizione con un livello di zoom appropriato.

Dopo l'inizializzazione della mappa, i geofence memorizzati nel database vengono osservati attraverso il **GeofenceViewModel**. Ogni geofence esistente viene recuperato e visualizzato sulla mappa come marker e cerchi che rappresentano il raggio dell'area monitorata.

Un listener viene aggiunto alla mappa per consentire all'utente di selezionare una nuova posizione premendoci sopra. Quando l'utente seleziona un punto sulla mappa, viene aggiunto un marker e disegnato un cerchio che rappresenta il raggio del potenziale nuovo geofence. La posizione selezionata viene memorizzata nella variabile **selectedLatLng**, in attesa che l'utente prema il pulsante "Add Geofence", per avviare il processo di creazione e memorizzazione del geofence tramite il metodo ***addGeofence()***.

Il metodo *addGeofence()* costruisce un oggetto **Geofence** utilizzando la classe **Geofence.Builder()**. Vengono utilizzati questi parametri principali:

- **setRequestId()**: Imposta un ID univoco per il geofence.
- **setCircularRegion()**: Definisce l'area geografica da monitorare come una regione circolare, specificando la latitudine, la longitudine e il raggio.
- **setExpirationDuration()**: Imposta la durata di vita del geofence. In questo caso, `NEVER_EXPIRE` significa che il geofence non scadrà mai.
- **setTransitionTypes()**: Specifica i tipi di transizioni da monitorare. In questo caso vengono monitorate sia l'entrata che l'uscita dall'area geofence.

Dopo la creazione del geofence, viene costruita una richiesta **GeofencingRequest** con **GeofencingRequest.Builder()**, che stabilisce come il sistema deve gestire il geofence:

- **setInitialTrigger(GeofencingRequest.**`INITIAL_TRIGGER_ENTER`**)**: Imposta il trigger iniziale per rilevare immediatamente l'ingresso nell'area al momento della creazione del geofence.
- **addGeofence()**: Aggiunge il geofence appena creato alla richiesta.

Il geofence viene poi aggiunto al sistema tramite il **GeofencingClient**. Il metodo *addGeofences()* prende in input la richiesta del geofence e un **PendingIntent**, un meccanismo che consente ad altre applicazioni (o a servizi come Google Play Services) di eseguire azioni future per conto

dell'applicazione. In questo caso, il PendingIntent viene utilizzato per attivare il GeofenceBroadcastReceiver quando l'utente entra o esce dal geofence.

Il **GeofenceBroadcastReceiver** è una classe che estende la classe **BroadcastReceiver** di Android e viene utilizzata per ricevere e gestire le transizioni del geofence.

Il metodo principale di un BroadcastReceiver è **onReceive()**. Questo metodo viene chiamato quando il sistema invia un Intent al receiver. Nel caso del GeofenceBroadcastReceiver, l'intent proviene da una transizione del geofence.

Dopo aver estratto l'oggetto **GeofencingEvent** dall'intent ricevuto tramite il metodo **GeofencingEvent.fromIntent(intent)**, viene rilevata la transizione del geofence tramite la proprietà **geofenceTransition**. A seconda del tipo di transizione, viene chiamato il metodo ***showNotification()*** per creare un Notification Channel se non esiste e inviare una notifica all'utente, indicando se è entrato o uscito dall'area geofence.

```
28  override fun onReceive(context: Context, intent: Intent) {
29      val geofencingEvent = GeofencingEvent.fromIntent(intent)
30
31      if (geofencingEvent != null) {
32          if (geofencingEvent.hasError()) {
33              val errorMessage = geofencingEvent.errorCode
34              Log.e( tag: "GeofenceBroadcastReceiver",  msg: "Error code: $errorMessage")
35              return
36          }
37      }
38
39      val geofenceTransition = geofencingEvent?.geofenceTransition
40
41      if (geofenceTransition == Geofence.GEOFENCE_TRANSITION_ENTER) {
42          showNotification(context,  message: "Entered geofence area")
43          Log.i( tag: "GeofenceBroadcastReceiver",  msg: "Entered geofence area")
44      } else if (geofenceTransition == Geofence.GEOFENCE_TRANSITION_EXIT) {
45          showNotification(context,  message: "Exited geofence area")
46          Log.i( tag: "GeofenceBroadcastReceiver",  msg: "Exited geofence area")
47      } else {
48          Log.e( tag: "GeofenceBroadcastReceiver",  msg: "Unknown geofence transition")
49      }
50  }
```