Report Progetto LAM 2024
Fitness Tracker

Giuseppe Spathis M. 0001043077 giuseppe.spathis@studio.unibo.it
Cono Cirone M. 0001029785 cono.cirone@studio.unibo.it
Luglio 2024

Indice

Panoramica app

`FitnessTracker` è un’applicazione Android, sviluppata per il corso "Laboratorio di applicazioni mobili" AA 2023/2024 dell'Università di Bologna. L'app consente agli utenti di tracciare varie attività giornaliere e di monitorare il tempo trascorso in luoghi di interesse. Le principali funzionalità sono:

- **Tracciamento delle Attività:** Consente di monitorare attività quotidiane come passeggiate, corse, guida, nonchè periodi di inattività.
- **Geofencing:** Permette di aggiungere luoghi di interesse per monitorare, in maniera automatica, il tempo trascorso in essi. L'app invia notifiche quando l'utente entra o esce da questi posti.
- **Visualizzazione delle Statistiche:**
    - **Calendario:** Mostra le attività giornaliere e i luoghi visitati, con dettagli per ogni attività.
    - **Grafici a Torta:** Fornisce una visualizzazione sommaria delle attività e dei luoghi frequentati, filtrabili per periodo di tempo.
    - **Grafici di Andamento:** Mostra il variare di parametri come il numero di passi, la distanza percorsa, il tempo trascorso seduto e la velocità media di guida nell’ultima settimana.
- **Condivisione dei Dati:** Consente di condividere dati con altri utenti nelle vicinanze tramite Bluetooth, con notifiche che avvisano della presenza di altri utenti vicini.
- **Sistema di Login e Registrazione:** Gestisce l'autenticazione degli utenti per la sicurezza e la personalizzazione dei dati.
- **Notifiche Periodiche:** Ricorda all'utente di tracciare nuove attività e di visualizzare i grafici dei dati recenti.

Dettagli implementativi

Scelte progettuali

Nel nostro progetto Android, abbiamo adottato il design pattern Model-View-Controller (MVC) ed elaborato scelte tecnologiche specifiche per garantire efficienza e facilità d'uso.

Abbiamo scelto il pattern MVC in quanto offre una chiara separazione tra logica di business, presentazione e gestione delle interazioni. Il file `Model` gestisce le interazioni con i database, i layout delle activities fungono da View, e le diverse activities agiscono come Controller, gestendo l'interazione dell'utente e aggiornando il Model e la View di conseguenza.

Per la gestione dei dati, ci serviamo di due tipi di database: **Room** è utilizzato per la persistenza locale, salvando le attività degli utenti e le informazioni sulle geofences. **Firebase** gestisce il sistema di registrazione e login degli utenti tramite Firebase Authentication, e offre sincronizzazione automatica tramite il Realtime Database.

Abbiamo scelto di supportare un minimo di SDK Android 8.0 (Oreo), che copre il 95.4% dei dispositivi. Questa scelta è stata fatta per garantire il pieno supporto a tutte le funzionalità recenti di Android, migliorando l'esperienza dell'utente e la compatibilità con le librerie moderne.

Struttura Database

Il database locale di Room è strutturato in quattro tabelle: `Attività`, `OthersActivity`, `GeoFence`, e `timeGeofence`.

- **Tabella `Attività`:** Memorizza le attività degli utenti con campi per ID, userId, startTime, endTime, date, activityType, stepCount, distance, pace, avgSpeed e maxSpeed.
- **Tabella `OthersActivity`:** Archivia le attività di altri utenti con campi simili ad `Attività` e aggiunge un campo per il nome utente (`username`).
- **Tabella `GeoFence`:** Memorizza i dati delle geofence con campi per ID, latitudine, longitudine, raggio e nome del luogo.
- **Tabella `timeGeofence`:** Traccia il tempo di entrata e uscita dalle geofence con campi per ID, userId, latitudine, longitudine, raggio, enterTime, exitTime, date e placeName.

DAO (Data Access Object)

L'interfaccia `ActivityDao` definisce metodi per interagire con le tabelle, inclusi metodi per inserire, aggiornare, eliminare e recuperare dati. Alcuni esempi sono `insertActivity`, `getAttivitàByDate`, `insertGeofence`, e `getAllTimeGeofences`.

Il database Firebase contiene una tabella principale per memorizzare le informazioni degli utenti:

- **Tabella `User`**: Memorizza le informazioni degli utenti con campi per ID, email, username, gender, lastLatitude, lastLongitude e lastUpdated.

Activities

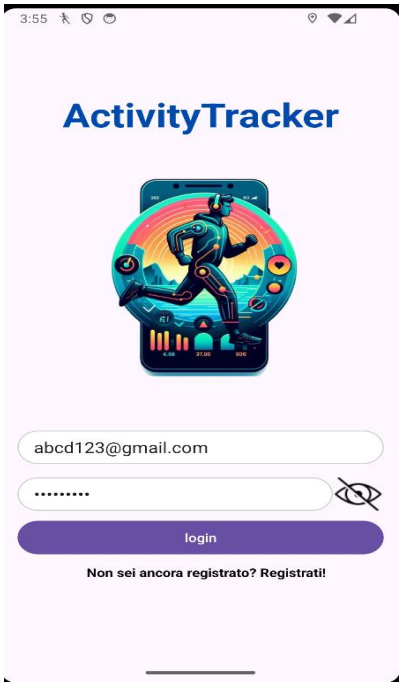
MainActivity

La MainActivity è il punto d'ingresso principale dell'applicazione, responsabile di garantire che tutte le autorizzazioni necessarie siano concesse dall'utente. Se le autorizzazioni sono concesse, l'app avvia i servizi di localizzazione e di monitoraggio degli utenti nelle vicinanze. Inoltre, la MainActivity inizializza `Firebase` e naviga verso la `LoginActivity` per l'autenticazione dell'utente.

Metodi principali

1. **checkPermissions:** Questo metodo verifica se le autorizzazioni necessarie sono già concesse. Ritorna false se non tutte le autorizzazioni sono state concesse, true altrimenti.
2. **startLocationService:** Questo metodo avvia il servizio `LocationUpdatesService` come foreground service. Ciò permette all'app di continuare a ricevere aggiornamenti di posizione anche quando è in background.
3. **startCheckNearbyUsersWorker:** Questo metodo configura un PeriodicWorkRequest per eseguire il worker CheckNearbyUsersWorker ogni 15 minuti. Utilizza *WorkManager* per gestire l'esecuzione periodica del worker, garantendo che l'app controlli periodicamente la presenza di utenti nelle vicinanze.

LoginActivity



La LoginActivity gestisce l'autenticazione degli utenti. L'activity permette agli utenti di accedere utilizzando le loro credenziali e di navigare verso la HomeActivity se l'accesso ha successo. Inoltre, consente agli utenti di passare alla schermata di registrazione nel caso in cui gli stessi non siano ancora registrati.

Metodi principali

1. **onCreate:** Configura l'interfaccia utente e gestisce il flusso logico principale. Controlla se l'utente è già autenticato; in caso affermativo, recupera i dati dell'utente e naviga verso la HomeActivity. Se l'utente non è autenticato, imposta i listener per i vari elementi dell'interfaccia utente, come i pulsanti di accesso e di registrazione.
2. **signInAndFetchUserData:** Tenta di autenticare l'utente con le credenziali fornite. Se l'autenticazione ha successo, recupera i dati dell'utente dal modello `Model` e li salva in un oggetto LoggedUser. Gestisce anche gli errori di autenticazione, mostrando messaggi appropriati all'utente in caso di fallimento.
3. **handleSignInError:** Gestisce gli errori che possono verificarsi durante il tentativo di accesso, mostrando messaggi specifici basati sul tipo di errore (e.g., email non valida, password errata, utente non trovato).

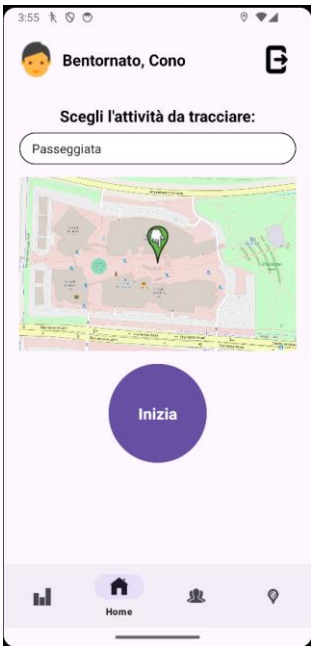
RegistrationActivity

La RegistrationActivity gestisce la registrazione degli utenti, includendo la raccolta di informazioni come email, username, password e genere. Verifica anche la validità delle credenziali inserite e salva i dati dell'utente nel database.

Metodi principali

1. **registerUser:** Registra un nuovo nuovo utente nel database Firebase utilizzando FirebaseAuth; successivamente chiama la funzione `Model.saveUserData` per salvare anche gli altri dati nel db.
2. **validateInput:** Valida l'email, la password e controlla che l'email e lo username non siano già in uso; per fare ciò richiama le funzioni `Model.emailExists` e Model.usernameExists. Mostra messaggi di errore appropriati se i dati non sono validi.

HomeActivity



La HomeActivity è responsabile della visualizzazione della mappa e della gestione delle attività principali dell'utente, come il reindirizzamento al tracciamento dell'attività fisica e il logout. Utilizza anche i servizi di localizzazione per ottenere la posizione corrente dell'utente e aggiornarla sulla mappa. La mappa è stata realizzata utilizzando il servizio **osmdroid**, che permette di visualizzare e interagire con le mappe OpenStreetMap all'interno dell'applicazione.

Metodi principali

1. **checkLocationPermission:** Controlla se i permessi di localizzazione sono concessi; in caso contrario, li richiede. Se i permessi sono concessi, chiama la funzione `getLastKnownLocation`.
2. **getLastKnownLocation:** Gestisce la posizione corrente dell'utente. Utilizza `FusedLocationProviderClient` per ottenere l'ultima posizione conosciuta e aggiorna la mappa con un marker nella posizione corrente.
3. **handleTrackButtonClick:** Gestisce il click del pulsante di tracciamento, avviando l'attività corrispondente in base all'opzione selezionata nello spinner (Passeggiata, Corsa, Guidare, Stare fermo). In base all'opzione scelta:

- **Passeggiata:** Mostra un dialog (gestito dalla funzione `showStepGoalDialog`) per impostare l'obiettivo in termini di passi.
- **Corsa:** Mostra un dialog (gestito dalla funzione `showDistanceGoalDialog`) per impostare l'obiettivo in termini di distanza.
- **Guidare:** Mostra un dialog (gestito dalla funzione `showSpeedLimitDialog`) per impostare il limite di velocità.
- **Stare fermo:** Avvia direttamente l'attività `SitActivity`.

Una volta impostato l'obiettivo tramite il dialog, l'utente viene reindirizzato all'attività corrispondente.

4. **handleLogout:** Gestisce la disconnessione dell'utente, resettando i dati dell'utente loggato e indirizzando alla `LoginActivity`.

WalkActivity

Si occupa di tracciare l'attività “passeggiata” utilizzando i sensori del dispositivo per contare i passi. L'interfaccia utente mostra le informazioni sull'attività in tempo reale, inclusi il tempo trascorso, il numero di passi e la distanza percorsa, con un indicatore di progresso che si riempie man mano che l'utente si avvicina al proprio obiettivo.

Utilizzo dei Sensori

Per tenere traccia del numero di passi effettuati dall'utente, viene utilizzato il sensore `TYPE_STEP_COUNTER`, che tiene traccia del numero totale di passi effettuati dall'ultima accensione del dispositivo. Nella classe `WalkActivity`, il sensore di contapassi viene gestito nel seguente modo:

1. **Inizializzazione**:

```
sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
stepCounterSensor = sensorManager.getDefaultSensor(Sensor.TYPE_STEP_COUNTER) as
Sensor
```

2. **Registrazione del listener:** Questo consente all'Activity di ricevere gli aggiornamenti dal sensore quando cambiano i dati (ovvero quando l'utente fa un passo)

```
sensorManager.registerListener(this, stepCounterSensor, SensorManager.SENSOR_DEL
AY_NORMAL)
```

3. **Gestione dati:** Dato che il sensore restituisce il numero totale di passi da quando l'utente accende il telefono è importante prendere in considerazione solo quelli effettuati da quando è stata avviata l'activity.

```
val totalSteps = event.values[0].toInt()
if(initialStepCount == 0) {
   initialStepCount = totalSteps
 }
stepCount = totalSteps - initialStepCount
```

Indicatore di Progresso

L'elemento UI che visualizza quanto l'utente si avvicina al proprio obiettivo di passi è un `ProgressBar` personalizzato. La `ProgressBar` viene aggiornata in tempo reale man mano che l'utente fa dei passi.

```
<ProgressBar
    android:id="@+id/progressBar"
    android:layout_width="130dp"
    android:layout_height="187dp"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_gravity="center"
    android:layout_marginTop="16dp"
    android:background="@drawable/circular"
    android:padding="10dp"
    android:progressDrawable="@drawable/custom_progress" />
```

Metodi principali

1. **onStopButtonclicked:** Quando l'utente clicca sul pulsante "Stop", l'attività viene salvata nel database tramite la funzione `saveActivity` definita nel `Model`. L'attività viene salvata nel db solo se la sua durata è superiore a un minuto. Se è troppo breve, viene mostrato un popup informativo.

RunActivity

Si occupa di tracciare l'attività 'Corsa'. Il funzionamento è identico a quello di `WalkActivity`; a differenza di quest'ultima, però, dà all'utente anche un'informazione sul passo a cui sta andando al momento.

SpeedActivity

E' progettata per tracciare la velocità a cui si muove l'utente utilizzando i sensori del dispositivo. Fornisce informazioni in tempo reale sulla velocità attuale, la velocità media e la velocità massima registrata. Inoltre, invia una notifica all'utente se la velocità supera il limite massimo impostato.

Utilizzo del Sensore

Utilizza il sensore di accelerazione lineare (`TYPE_LINEAR_ACCELERATION`) per ottenere i dati necessari a calcolare la velocità dell'utente. Viene gestito nel seguente modo:

1. **Inizializzazione:**

```
sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
speedSensor = sensorManager.getDefaultSensor(Sensor.TYPE_LINEAR_ACCELERATION)!!
```

2. **Gestione dati:**

Il codice calcola la velocità lineare a partire dai valori di accelerazione sui tre assi (X, Y, Z) usando il teorema di Pitagora. Somma i quadrati delle accelerazioni, ne calcola la radice quadrata, e poi converte il risultato da m/s² a km/h moltiplicando per 3.6.

```
val linearAcceleration = Math.sqrt(
            (event.values[0] * event.values[0]
                    + event.values[1] * event.values[1]
                    + event.values[2] * event.values[2]).toDouble()
        )
speed = linearAcceleration * 3.6 // Converte m/s^2 in km/h
```

Metodi principali

1. **onSensorChanged:** Viene chiamato ogni volta che il sensore registra un nuovo valore. Se la velocità corrente supera la velocità massima impostata, viene inviata una notifica all'utente.
2. **sendNotification:** Quando la velocità supera la soglia massima, il metodo `sendNotification` elabora e invia una notifica. La notifica viene creata utilizzando `NotificationCompat.Builder`. Il canale di notifica è configurato per assicurarsi che le notifiche siano visibili, con priorità alta per attirare l'attenzione dell'utente.

SitActivity

È progettata per monitorare il tempo totale che l'utente trascorre seduto e per inviare notifiche di promemoria ogni mezz'ora, ricordandogli di fare attività. Ogni qualvolta l'utente si alza può resettare il timer, che ripartirà da 0.

Metodi principali

1. **updateTimeRunnable**: Runnable che viene eseguito ogni secondo per aggiornare i tempi e verificare se è necessaria una notifica. Se il tempo dall'ultima volta che l'utente si è alzato supera i 30 minuti, viene inviata una notifica.

GeoFenceActivity

Consente agli utenti di interagire con le geofences su una mappa. Le principali funzionalità includono la possibilità di cercare posizioni tramite OpenStreetMap (OSM), aggiungere nuove geofences, visualizzare quelle esistenti e rimuoverle. La mappa utilizzata è interattiva e basata sulla libreria osmdroid.

Metodi principali

1. **searchLocation:** Questo metodo utilizza i servizi di OSM per cercare una posizione basata sulla query dell'utente. Se la posizione viene trovata, richiama la funzione `moveToLocation` per aggiornare la mappa, centrandola sulle coordinate trovate; inoltre aggiunge un marker visibile.
2. **`addGeofenceAtLocation:`** Richiama il metodo `Model.insertGeofence` per salvare nel db la geofence con il nome scelto dall'utente.
3. **showDeleteGeofenceDialog:** Quando l'utente clicca su un marker, viene mostrato un dialogo di conferma che chiede se si desidera rimuovere la geofence selezionata. Se l'utente conferma la rimozione, viene richiamato il metodo `Model.deleteGeofence` per rimuovere la geofence dal db.
4. **viewGeofences:** Tramite il metodo Model.getAllGeofences vengono recuperate tutte le geofences dal db e la mappa viene aggiornata per mostrarle. Ogni geofence è rappresentata da un marker e da un cerchio che indica l'area di copertura.

StatsActivity

La schermata delle statistiche dell'applicazione offre una panoramica dettagliata delle attività svolte e dei luoghi visitati dall'utente. Viene utilizzata la libreria `MPAndroidChart` per visualizzare i dati in vari formati di grafici, inclusi bar charts, pie charts e un line charts.

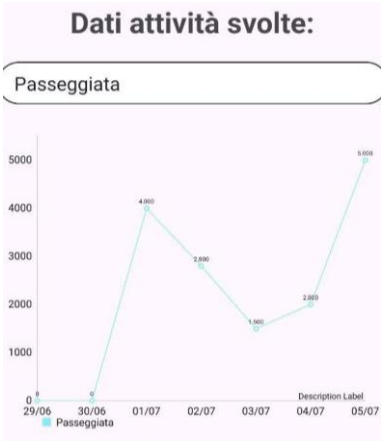
Pie charts

I grafici a torta permettono all'utente di visualizzare il tempo totale che ha trascorso nello svolgere un'attività o in un determinato luogo; l'utente può filtrare tali dati in base a giorno, settimana, mese e anno. Vi sono due grafici a torta: uno per le attività e uno per i luoghi. Per la realizzazione è stata utilizzata la classe `PieChart` di `MPAndroidChart`.

Metodi principali

1. **updatePieChartForPeriod:** Metodo che in base al periodo inserito e al tipo di pie chart da aggiornare richiama o la funzione Model.getActivitiesForPeriod o la funzione Model.getGeofencesForPeriod per ottenere i dati da inserire nel grafico .

Line chart



Il line chart mostra l'andamento delle attività nel tempo, offrendo una visione temporale delle tendenze. Viene utilizzata per crearlo la classe `LineChart` di `MPAndroidChart`.

Metodi principali

1. **prepareStepCountData, prepareDistanceData, prepareAvgSpeedData, prepareStationaryTimeData:** Sono progettate per preparare i dati di attività dell'utente per la visualizzazione in grafici. Iterano attraverso una lista di attività per aggregare i dati giornalieri (ad esempio, sommare i passi, le distanze, o calcolare la velocità media). Assicurano che ogni giorno della settimana precedente abbia un valore, anche se non sono presenti attività in quei giorni, impostando i valori a zero o valori predefiniti. Ordinano i dati per data, per garantire che i grafici mostrino una progressione temporale corretta.

Calendar View

In Stats Activity é presente inoltre un calendario con la funzionalità di visualizzare e filtrare le attività tracciate relativamente a ogni giorno.

Il calendario è stato realizzato utilizzando la libreria **ProlificInteractive/material-**

calendarview, che offre un'ampia gamma di strumenti e opzioni per la gestione e visualizzazione dei dati su un calendar view. Le principali caratteristiche implementate nel nostro calendario sono:

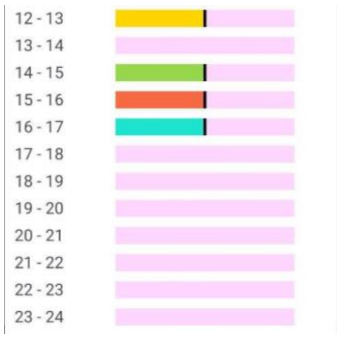
- **Visualizzazione delle Attività Giorno per Giorno**:
    - Cliccando su un giorno specifico del calendario, si apre un **modal** che mostra tutte le attività tracciate in quel giorno utilizzando dei **bar chart** che saranno descritti in seguito.
- **Filtraggio delle Attività**:
    - Il calendario è dotato di un sistema di filtro che consente di selezionare e visualizzare i giorni del calendario in base al tipo di attività tracciata in quel singolo giorno. Dal punto di vista implementativo viene usato uno spinner per filtrare in base al tipo di attività; visualmente il calendario viene modificato usando un **custom DayViewDecorator** per ogni giorno

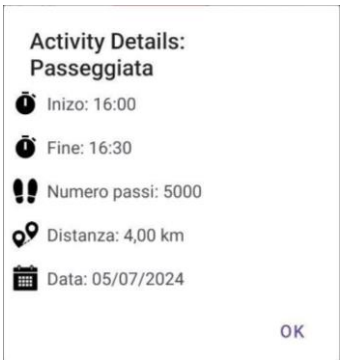
Bar chart

Il grafico a barre orizzontali è creato utilizzando un insieme di `HorizontalBarChart`, inseriti all'interno di un layout verticale. Questo grafico mostra il tempo dedicato a diverse attività per ciascuna ora del giorno. Ogni HorizontalBarChart rappresenta un'ora specifica e visualizza le attività svolte in quell'ora.

Metodi principali

1. **displayActivitiesForDate, displayGeofencesForDate:** Permettono di calcolare il tempo dedicato a ciascuna attività per ogni ora: creano con cadenza oraria un `HorizontalBarChart`, configurandolo con i dati aggregati, e impostano un listener per gestire i clic sulle barre e mostrare i dettagli delle attività in un popup.
2. **showActivityPopup**: Permette di visualizzare i dettagli dell'activity selezionata nel bar chart.
3. **showGeofencesInfoDialog:** Permette di visualizzare i dettagli sul luogo in cui si è stati.





Social Activity

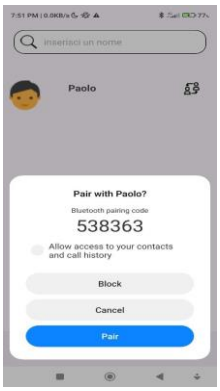
L'activity "Social" si occupa di mettere in comunicazione più utenti attraverso il Bluetooth, consentendo loro di inviare messaggi e condividere i dati delle attività tracciate.
Vengono usati i file ausiliari Social Handler per gestire tutto il sistema del bluetooth e myAdapter per gestire la lista di utenti vicino all'utente, rilevati col bluetooth.
Di seguito, una panoramica dei principali aspetti:

1. **Gestione del Bluetooth:**

- All’avvio dell’activity, vengono richiesti i permessi necessari per utilizzare il Bluetooth.
- I dispositivi devono essere accoppiati per poter comunicare tra loro.
- Un dispositivo che vuole fungere da server si rende “visibile agli altri” tramite l’apposito bottone. In questo modo, viene creato un **listener RFCOMM Bluetooth socket,** che attende connessioni in entrata.
- Un dispositivo client avvia la scansione dei dispositivi Bluetooth nelle vicinanze. Quando viene rilevato un dispositivo, un broadcast receiver verifica se l’utente è registrato su Firebase e, in caso positivo, lo aggiunge alla lista dell’adapter. Cliccando su ogni nome presente nella lista, si possono vedere le attività che quell’utente ha condiviso con te in precedenza.

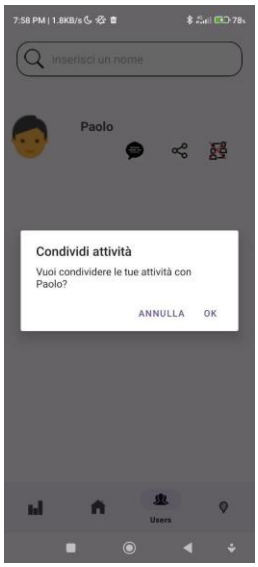
```
private val receiver = object : BroadcastReceiver() {
    @SuppressLint("MissingPermission")
    override fun onReceive(context: Context?, intent: Intent?) {
        when(intent?.action){
            BluetoothDevice.ACTION_FOUND -> {
                val device = if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU){
                    intent.getParcelableExtra(
                        BluetoothDevice.EXTRA_DEVICE,
                        BluetoothDevice::class.java
                    )
                }
                else {
                    intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE)
                }

                if(device?.name != null){
                    CoroutineScope(Dispatchers.IO).launch { this: CoroutineScope
                        val found = model.updateList(device)
                        if(found){
                            withContext(Dispatchers.Main) { this: CoroutineScope
                                SocialInterface.listUpdated(getPersonlist())
                            }
                        }
                    }
                }
            }
        }
```

2. **Accoppiamento e Comunicazione:**
    - Per l’accoppiamento, viene utilizzata la funzione **connect2device**. Questa funzione monitora gli eventi di accoppiamento di un particolare dispositivo Bluetooth usando sempre un broadcast receiver. Quando lo stato cambia in “accoppiato”, viene stabilita una connessione socket.



    - I dati vengono scambiati tramite gli stream di input e output della connessione Bluetooth.
    - Per distinguere tra messaggi e file JSON nell’output stream, viene concatenato il carattere “#” all’inizio dei file JSON inviati.

LATO CLIENT **LATO SERVER**



3. Gestione della Disconnessione:

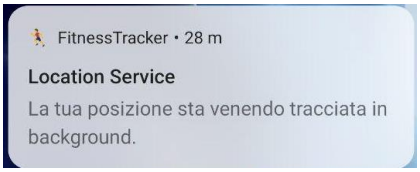
- La funzione **disconnectDevice** gestisce la situazione in cui si desidera rimuovere l'accoppiamento. Essa chiude tutti gli elementi di comunicazione, invocando la funzione close() su ogni singolo elemento.

```
fun disconnectDevice(activity: Activity, holder: MyAdapter.MyViewHolder, you_are_connected: MyAdapter.Ref<Boolean>, device: BluetoothDevice?){
    Thread {
        try {
            try {
                closeConnections()
            } catch (e: UninitializedPropertyAccessException) {
                //non fare niente se le variaibli non sono state inizializzate
            }
            unBondBluetoothDevice(device)
            activity.runOnUiThread {
                MotionToast.createColorToast(
                    activity,
                    activity.resources.getString(R.string.successo),
                    activity.resources.getString(R.string.disconnected),
                    MotionToastStyle.SUCCESS,
                    MotionToast.GRAVITY_BOTTOM,
                    MotionToast.LONG_DURATION,
                    ResourcesCompat.getFont(activity, www.sanju.motiontoast.R.font.helvetica_regular))

                holder.connect.visibility = View.VISIBLE
                holder.message.visibility = View.GONE
                holder.share.visibility = View.GONE
                holder.disconnect.visibility = View.GONE
                you_are_connected.value = false
            }
        }
        catch (e: IOException) {
                socketError(e, activity)
        }
    }.start()
}
```

Servizi in background

Sono stati implementati due tipi di servizi in background: `LocationUpdatesService` e `CheckNearbyUsersWorker`. Tali servizi sono fondamentali per garantire il tracciamento continuo della posizione dell'utente e per notificare all'utente quando si trova nelle vicinanze di altri utenti.

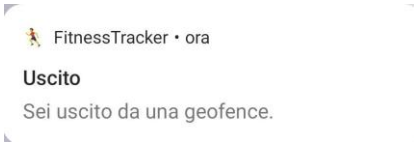
Il servizio `LocationUpdatesService` è responsabile del monitoraggio continuo della posizione dell'utente e della gestione delle geofence.



Caratteristiche principali:

- **Monitoraggio in Foreground:** Questo servizio è progettato per funzionare come un servizio in foreground, assicurando che il sistema operativo lo mantenga attivo anche quando l'applicazione non è visibile. Ciò è cruciale per l'aggiornamento continuo della posizione.
- **Creazione della Richiesta di Posizione:**
  Utilizza `FusedLocationProviderClient` per ottenere aggiornamenti della posizione con alta precisione (`PRIORITY_HIGH_ACCURACY`). La frequenza di aggiornamento è configurata per ottenere nuove posizioni ogni 10 secondi, permettendoci di sapere costantemente dove si trova l'utente, riuscendo inoltre a comprendere in maniere precisa quanto tempo trascorre in una determinata geofence.

- **Gestione delle Geofence:** Quando viene rilevata una posizione, il servizio verifica se l'utente è all'interno di una geofence predefinita. Se l'utente entra o esce da una geofence, vengono generate notifiche appropriate e vengono registrati gli eventi nel database locale.



- **Prevenzione di Servizi e Notifiche Duplicati**: Per evitare la creazione ripetuta dello stesso servizio, il metodo **startServiceIfNotRunning** controlla lo stato del servizio prima di avviarlo. Allo stesso modo, le notifiche vengono inviate solo quando si verifica effettivamente un cambiamento nello stato della geofence.

Abbiamo deciso di sviluppare un'implementazione personalizzata per il geofencing, anziché utilizzare il Geofencing Client di Google, principalmente a causa della latenza degli avvisi. Il servizio di geofencing di Google non esegue infatti continuamente query per la posizione, e ciò introduce una latenza significativa quando vengono inviati gli avvisi. Nello specifico, la latenza potrebbe essere intorno ai 2-3 minuti in condizioni normali. Inoltre, se il dispositivo è rimasto fermo per un periodo prolungato, la latenza può aumentare fino a 6 minuti. Questo comportamento non è accettabile per applicazioni che richiedono risposte tempestive ai cambiamenti di posizione, come la nostra. Con un sistema personalizzato, abbiamo avuto la libertà di ottimizzare la gestione delle posizioni e ridurre al minimo la latenza degli avvisi, garantendo che gli utenti ricevano notifiche in tempo reale. Inoltre, la scelta di non utilizzare il Geofencing Client di Google ci ha permesso di superare il limite di 100 geofences imposto dal servizio di Google, offrendo così una soluzione scalabile, che può gestire un numero illimitato di geofences senza compromettere le prestazioni o l'affidabilità del sistema.

Il worker `CheckNearbyUsersWorker` è implementato utilizzando `WorkManager` e viene eseguito periodicamente per controllare la presenza di utenti vicini. L'uso di `WorkManager` garantisce che il controllo venga eseguito in modo efficiente, rispettando le politiche di

gestione del risparmio energetico del sistema operativo; inoltre il `WorkManager` gestisce automaticamente la riprogrammazione dei lavori in caso di fallimenti temporanei, garantendo che il controllo degli utenti vicini avvenga regolarmente.

Caratteristiche principali:

- **Esecuzione Periodica:** Questo worker è configurato per eseguire controlli ogni 15 minuti, utilizzando `PeriodicWorkRequestBuilder`.
- **Verifica degli Utenti Vicini:** Recupera la posizione corrente dell'utente e controlla la presenza di altri utenti che sono stati attivi negli ultimi 10 minuti. Se trova utenti nelle vicinanze, invia una notifica.
- **Notifiche:** Utilizza `NotificationCompat` per inviare notifiche agli utenti, avvisandoli della presenza di altri utenti vicini.

I servizi in background sono richiamati e gestiti principalmente all'interno di `MainActivity` e `GeoFenceActivity`.

Notifiche periodiche

Le notifiche periodiche sono avvisi temporali programmati per attivarsi regolarmente a intervalli specifici. Per implementare tali notifiche, sono stati utilizzati dei worker simili a quelli descritti precedentemente.

Revisione Attività

L'idea alla base di questa funzionalità è di inviare una notifica alle 11 di sera per ricordare all'utente di rivedere le attività tracciate durante la giornata. Per implementare questa funzionalità, è stato utilizzato un worker che viene eseguito una volta al giorno, precisamente alle 11 di sera, e invia la notifica. Se l'utente clicca sulla notifica, viene aperta direttamente la StatsActivity, permettendogli di esaminare rapidamente le attività giornaliere.

Tracciamento Attività

L'obiettivo di questo servizio è di inviare una notifica all'utente se non ha aperto l'app per un periodo di 4 ore, invitandolo a tracciare le attività. L'implementazione avviene utilizzando SharedPreferences per salvare l'ultima volta che l'app è stata aperta, nel campo last_opened; quindi il valore di tale variabile viene aggiornato nella MainActivity.

Un worker, eseguito ogni 4 ore, controlla se il tempo corrente, meno il valore di last_opened, è maggiore di 4 ore. In caso positivo, viene inviata una notifica. Cliccando sulla notifica, l'utente viene indirizzato direttamente alla HomeActivity, dove può tracciare le nuove attività.