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

Rocca Claudio
Matricola 1020395
claudio.rocca@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 compiute durante la giornata, contraddistinte in **camminata**, **spostamenti in macchina** e **attività sedentarie**. L'applicazione possiede differenti funzionalità, tra cui la registrazione delle attività compiute e la visualizzazione dei dati di altri utenti presenti sulla piattaforma. Tutte le funzionalità sviluppate saranno successivamente approfondite nella sezione **Feature**, tuttavia è possibile anticipare ciò che contraddistingue il sistema software ideato.

Ad un primo avvio l'utente visualizza la schermata per la registrazione in cui può effettuare il *Sign In* in caso di un nuovo utente, o il *Login* in caso di utente già registrato. Proseguendo, viene mostrata la schermata Home dell'app, composta da una *Navigation Bottom Bar*, che garantisce l'accesso diretto alle funzionalità sviluppate.
Tramite la Navigation Bar l'utente ha la possibilità di **registrare** nuove attività, **visualizzare** il proprio storico, osservare le proprie **aree geografiche di interesse**, visualizzare i **grafici** relativi alle proprie attività registrate ed, infine, condividere e visualizzare le attività condivise da altri utenti nella sezione **amici**.

Di seguito vengono brevemente descritte alcune scelte implementative e strumenti utilizzati durante lo sviluppo dell'app: per il versionamento del codice è stato utilizzato **Git**, con repository accessibile su **GitHub**, mentre per lo sviluppo dell'applicativo è stato utilizzato il linguaggio di programmazione **Kotlin**, per la realizzazione di un'applicazione **nativa Android**. Il progetto si compone di tre punti cardine per il corretto funzionamento, suddivisi come segue:

- Design pattern architetturale **MVVM**, utilizzato per separare i componenti **Model**, ovvero il *contenitore di dati*, e la sezione *attiva*, la **View** mostrata direttamente all'utente. In questo modo si garantisce la separazione tra i componenti, riducendo l'accoppiamento e le *dipendenze funzionali*, che potrebbero causare danni ai componenti dell'applicazione anche in seguito a piccole modifiche
- **Room**, utilizzato per garantire la persistenza dei dati all'interno di un *Database Relazionale*
- **Amazon S3**, utilizzato per salvare in cloud i dati degli utenti attraverso un *database dump*, consentendo la condivisione dei dati salvati sul *database locale* e permettendo di visualizzare i dati condivisi dagli amici

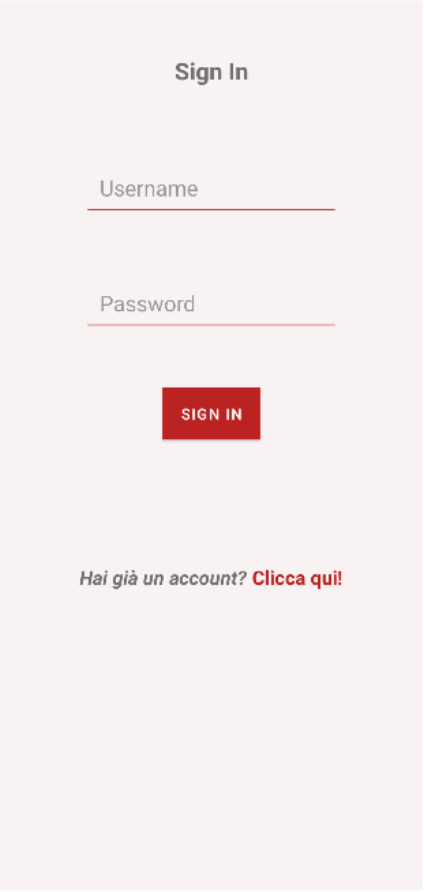
Feature

Registrazione

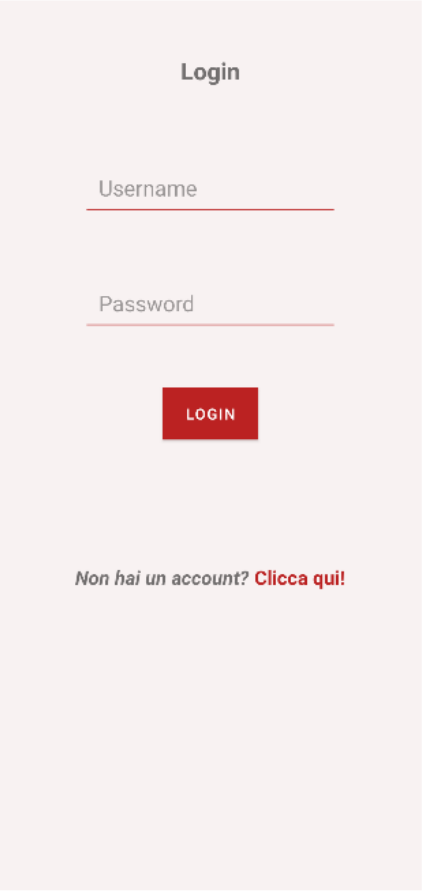
All'avvio dell'app è richiesta la registrazione delle credenziali. E' necessario registrarsi con un username non ancora presente, per garantire l' univocità dei dati. Una volta registrato, all'utente sarà richiesto di concedere tutti i permessi necessari per il funzionamento dell'applicazione. L'applicazione utilizza la *connessione Internet* per controllare se un altro utente si è già registrato in passato con lo stesso username, è pertanto necessario avere una connessione dati per registrarsi all'app. Tra i permessi richiesti sono presenti:

- **Access fine location**, autorizzazione di accesso alla posizione corrente del dispositivo, necessaria per monitorare le aree geografiche varcate
- **Access background location**, permesso necessario per accedere alla posizione del dispositivo anche quando l'applicazione è in secondo piano
- **Post notifications**, consente di inviare notifiche al dispositivo

Qualora siano negati i permessi richiesti non sarà consentito l'utilizzo dell'applicazione; ad un nuovo avvio è mostrato un *Alert Dialog* in cui è richiesto all'utente di abilitare le autorizzazioni necessarie tramite *Impostazioni*.



(a) Sign In fragment



(b) Login fragment

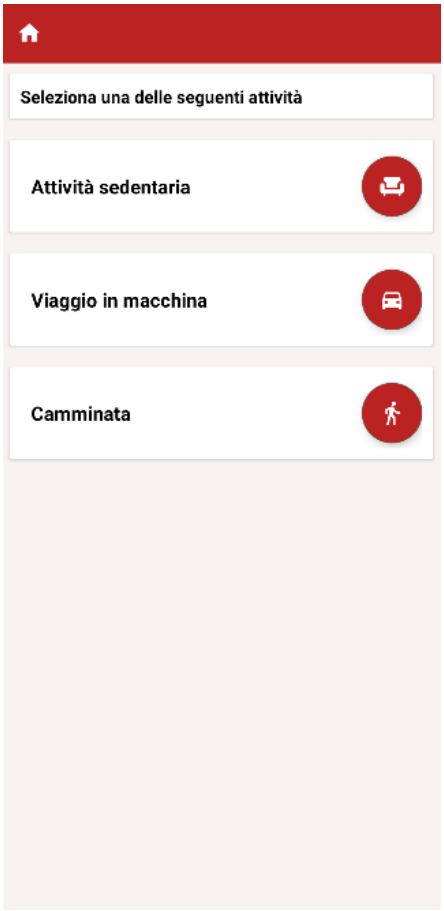
Registrazione attività

Proseguendo, successivamente all'autenticazione, è possibile registrare le attività compiute durante la giornata cliccando sul *floating button* presente nella schermata **Home**, **Calendario** e **Grafici**. L'utente può registrare le seguenti attività:

- **Attività sedentaria**, in cui è definito il tempo trascorso
- **Camminata**, in cui sono memorizzati i passi fatti e il dislivello percorso durante la registrazione
- **Spostamenti in macchina**, durante i quali viene registrata la distanza percorsa

Tutti i dati relativi alle attività dell'utente vengono salvati in un database locale e rielaborati per consentirne la visualizzazione grafica. Per consentire ciò è necessario che l'utente, abbia in precedenza autorizzato il permesso **Activity recognition**. Quest'ultimo garantisce di identificare propriamente l'avvio oppure l'interruzione di un'attività.

Nota bene: nel progetto proposto la funzionalità legata al **detect** dell'attività in corso non consiste in un'**operazione in background**.

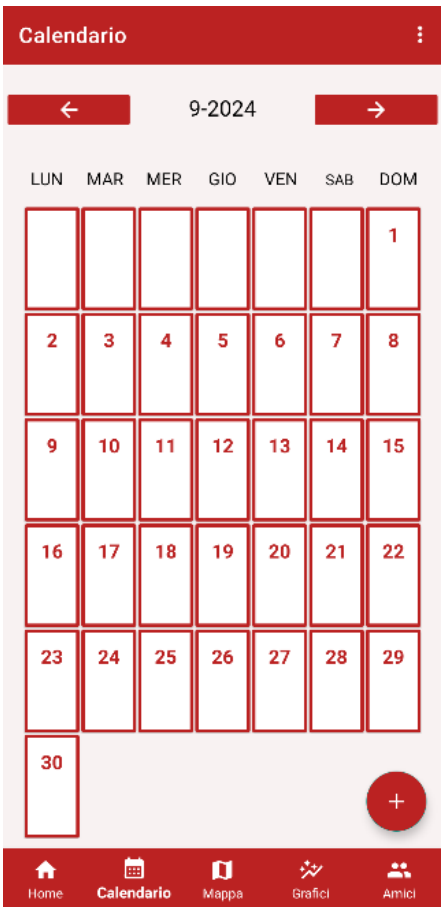


(c) Registrazione nuove attività

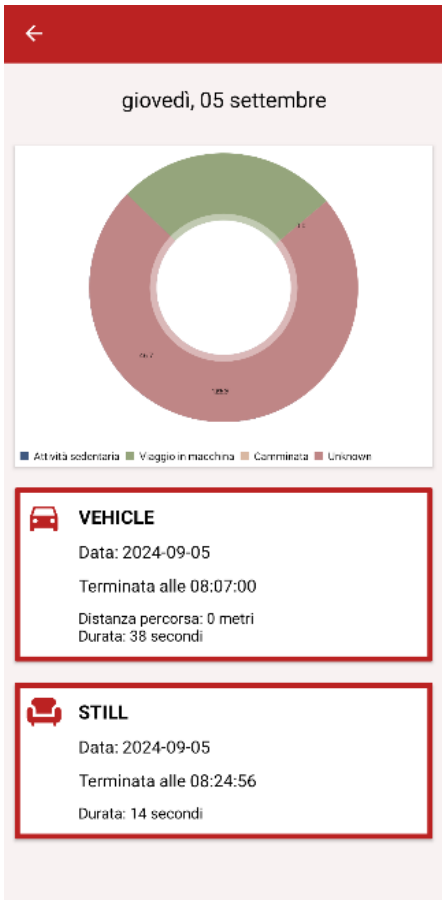
(d) Avvio/Interruzione registrazione attività

Calendario

L'utente può visualizzare nel **calendario** tutte le attività registrate. Cliccando su una delle celle che compongono il *calendario*, è possibile visualizzare le attività compiute nella giornata selezionata e le relative informazioni, oltre ad un *grafico a torta* che riporta il tempo speso per i vari tipi di attività durante l'arco delle 24 ore, specificando la porzione di tempo di cui non si hanno dati a disposizione.



(e) Calendario



(f) Visualizzazione attività

Grafici

La feature *grafici* permette di visualizzare in maniera intuitiva i dati registrati dall'applicazione nel mese selezionato. La *View* è composta da tre grafici, cosi suddivisi:

- **Grafico a barre**, contenente i **passi fatti** dall'utente per ogni giorno che componga il mese registrati durante attività di *camminata*
- **Grafico a barre**, utilizzato per visualizzare la **distanza percorsa** dall'utente ogni giorno durante le attività di *spostamento con veicolo*

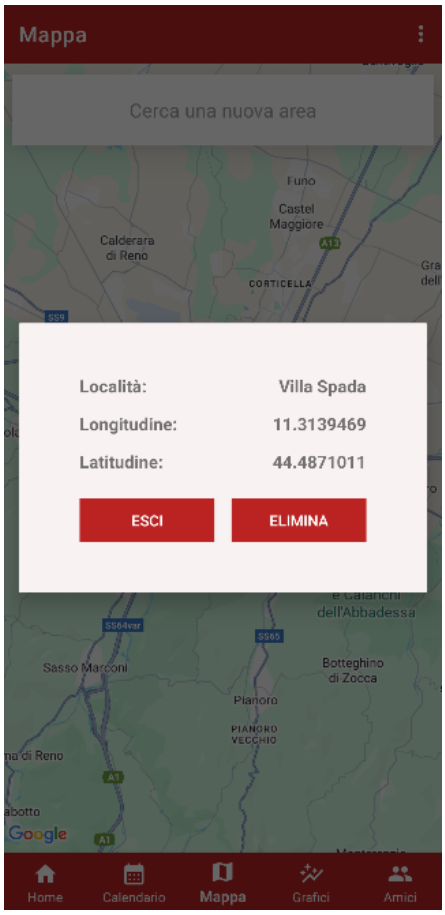
- **Grafico a torta**, che riporta in percentuale la suddivisione per **tipo di attività** registrate dall'utente

Mappa

La *View* contiene una **mappa** all'interno di un *FragmentContainerView*. L'utente può inserire, visualizzare ed eliminare le *aree geografiche di interesse*, registrate all'interno del *database locale*. Una zona di interesse è l'area compresa nel raggio di [150m, 250m] rispetto alle coordinate geografiche selezionate, varcata la soglia di una di esse l'utente riceve una notifica.

Nota bene: il funzionamento della feature è garantito solamente in presenza di una **connessione dati stabile**.

(g) Mappa



(h) Dialog dei marker sulla mappa

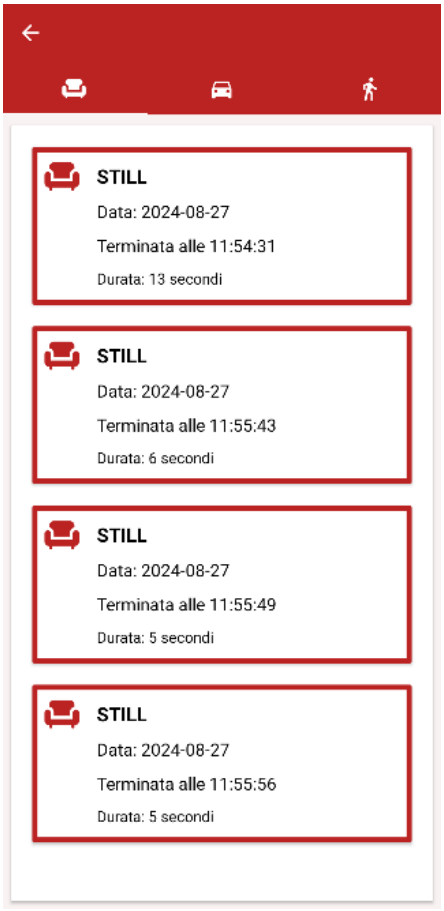
Amici

Tramite l'applicativo è possibile condividere le proprie attività con altri **utenti** utilizzatori e visualizzare le stesse da loro compiute. Un utente può cercare lo *username* di un amico per poi aggiungerlo alle proprie **amicizie** e visualizzare le attività da esso memorizzate. Precedentemente alla condivisione dei dati, viene mostrato un *Alert Dialog* che richiede

l'autorizzazione per il salvataggio dei dati in cloud, se il permesso dovesse essere negato sarà comunque possibile visualizzare i dati degli amici. Inoltre, l'utente può sincronizzare i dati salvati in *cloud* in qualsiasi momento, cliccando sul *floating button* presente nella schermata.

Nota bene: il funzionamento della feature è garantito solamente in presenza di una **connessione dati stabile**.

(i) Amici fragment



(j) Visualizzazione dati amico

Scelte implementative

RecyclerView

Questo tipo di *View* è stata utilizzata per la rappresentazione delle attività dell'utente, essa è infatti ottimizzata per gestire grandi quantità di dati pur garantendo buone prestazioni nella velocità di caricamento.

Negli screenshot riportati, la **RecyclerView** è stata utilizzata nelle schermate Home, Calendario e Amici, riutilizzando il codice per caricare i dati all'interno delle card. In diverse occasioni sono state implementate funzionalità direttamente nei singoli ViewHolder, operazione resa possibile grazie al *binding* dei parametri, semplificato dall'implementazione degli *Adapter*. Questo approccio permette una gestione più efficiente e flessibile dei dati nelle varie sezioni dell'app.

ViewModel

Come già accennato nell'introduzione, uno dei punti cardine su cui si basa il progetto è il design pattern architetturale **Model-View-ViewModel**.
Questo approccio separa la *View*, definita **entità attiva**, poichè al suo interno è sviluppata la *business logic*, e il Model, **entità passiva**, contenitore statico di dati.

Il **ViewModel** è incaricato di mantenere aggiornate le strutture dati che acquisiscono informazioni dal *Model*. Ciò avviene per garantire che tutti i soggetti il cui comportamento può variare al cambiamento dei dati contenuti nel *Model*, possano essere notificati dinnanzi ad una qualsiasi modifica. Questo comportamento è stato ottenuto attraverso l'impiego di **LiveData**, i quali si basano sul concetto di *Observables*.

```
class NetworkViewModel: ViewModel() {
    private val _currentNetwork = MutableLiveData<Boolean>()
    val currentNetwork: LiveData<Boolean> get() = _currentNetwork

    init {
        _currentNetwork.value = MyNetwork.isConnected
    }

    fun setNetwork(enabled: Boolean) {
        viewModelScope.launch(Dispatchers.Main) {
            if (_currentNetwork.value != enabled) {
                _currentNetwork.value = enabled
            }
        }
    }
}
```

La sezione di codice riportata definisce il metodo utilizzato per mantenere aggiornati i *Fragment* interessati allo stato corrente della *connessione*. All'interno delle classi è implementata una funzione incaricata di osservare cambiamenti del *LiveData currentNetwork*, abilitando oppure disabilitando le funzionalità che richiedono una connessione.

Room

Per garantire la persistenza dei dati è stata utilizzata la libreria **Room**. *Room* permette di gestire con semplicità un *Database Relazionale*, consentendo la gestione automatica di

aggiornamento del database, causato per esempio dall'introduzione di una nuova tabella o dalle modifiche apportate alla struttura di una esistente.

L'implementazione di *Room* è piuttosto intuitiva. Dichiarata un'instanza del database, è necessario creare le differenti tabelle che caratterizzano il *Modello E-R* dedotto e i loro **DAO**, ossia classi in cui sono definite le principali interazioni possibili con il database.

```
@Database(
    entities = [UserEntity::class, UserStillActivityEntity::class,
        UserVehicleActivityEntity::class, UserWalkActivityEntity::class,
        GeofenceAreaEntry::class, FriendEntity::class],
    version = ROOM_DATABASE_VERSION
)
abstract class BrockDB: RoomDatabase() {
    abstract fun UserDao(): UserDao
    abstract fun UserStillActivityDao(): UserStillActivityDao
    abstract fun UserVehicleActivityDao(): UserVehicleActivityDao
    abstract fun UserWalkActivityDao(): UserWalkActivityDao
    abstract fun GeofenceAreaDao(): GeofenceAreaDao
    abstract fun FriendDao(): FriendDao

    companion object {
        @Volatile
        var INSTANCE: BrockDB? = null

        @Synchronized
        fun getInstance(context: Context): BrockDB {
            if (INSTANCE == null) {
                INSTANCE = Room.databaseBuilder(
                    context,
                    BrockDB::class.java,
                    "brock.db"
                ).fallbackToDestructiveMigration()
                    .build()
            }
            return INSTANCE as BrockDB
        }
    }
}
```

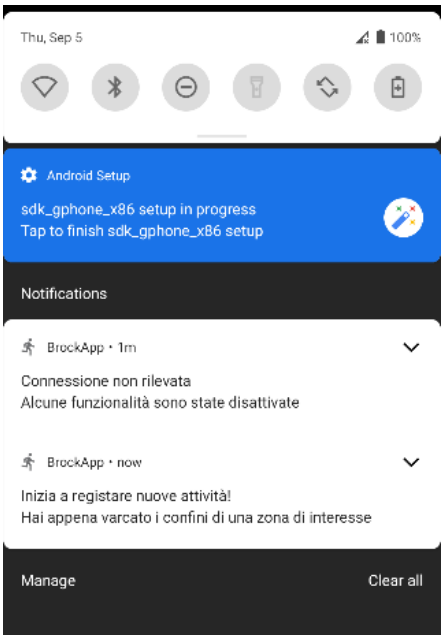
Come è possibile osservare dallo snippet di codice proposto, il database è composto da sei *tabelle*, ad ognuna delle quali è associato il proprio *DAO*. Inoltre, seguendo il *design pattern Singleton*, è realizzata una sola istanza del *database*, per garantire che tutte le operazioni di lettura e scrittura siano sincronizzate e che facciano riferimento agli stessi dati.

Background operations

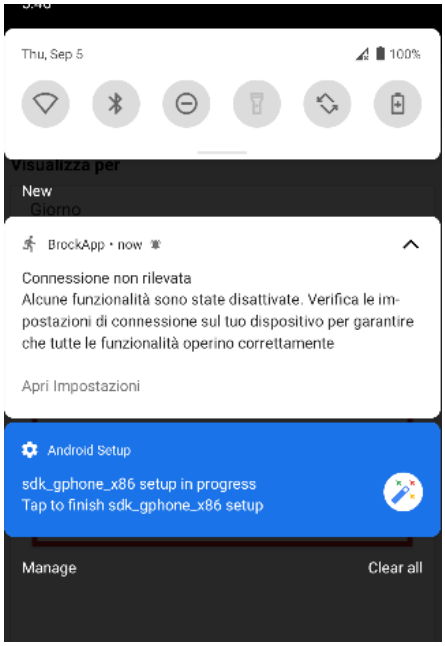
Le **operazioni in background** sviluppate all'interno del progetto si suddividono in due sezioni, cosi descritte:

- **Geofencing**, recinti virtuali utilizzati per contrassegnare località di interesse. A partire dalle coordinate geografiche scelte dall'utente viene registrata un'area circolare, rispetto ad un raggio prestabilito. Qualora l'utente dovesse varcare la soglia di una di queste are, l'applicativo invia una notifica al dispositivo
- **Connectivity**, utilizzato per accertarsi della presenza di una *connessione ad Internet* stabile, abilitando oppure disabilitando determinate funzionalità richiedenti una con-

nessione dati. Tale approccio è utilizzato soprattutto per ovviare a comportamenti indesiderati dell'applicazione



(k) Notifica proveniente da Geofencing



(l) Notifica proveniente da Connectivity

Geofencing
Come già accennato, l'operazione in questione controlla attivamente qualora l'utente varchi i confini di un'area di interesse. A questo scopo, è stato implementato un **Broadcast Receiver** capace di intercettare intenti lanciati dai componenti che registrano la posizione dell'utente. Un **Worker** viene poi incaricato di notificare il dispositivo dell'utente.

Connectivity
Contrariamente a quanto fatto per il Geofencing, in questo caso è stato utilizzato un **Service**. Nuovamente, è stato implementato un **Broadcast Receiver** affinchè possano essere gestiti cambiamenti della connessione. Una volta ricevuto l'intento viene richiamato il *Service*, in cui è controllato lo stato della connessione: a seconda della disponibilità della connessione vengono abilitate o disabilitate le feature che richiedono l'accesso ad Internet.
Infine, è avviato un **Worker** incaricato di notificare l'utente in caso di cambiamenti significativi della connessione.

Tuttavia, è bene affermare che il *Service* descritto è utilizzato anche per modificare la dimensione dell'aree di interesse. In base alla tipologia di *connessione ad Internet* presente, il servizio provvederà a modificare il raggio delle *aree di interesse*, in ottica di risparmio energetico e per garantire maggiore accuratezza anche in caso di connessione scarsa.

AWS Simple Storage Service

Per consentire agli utenti di condividere i propri dati è stato utilizzato *AWS Simple Storage Service*, la scelta è ricaduta su esso per le conoscenze pregresse del gruppo e per la facile

integrazione fornita dalla AWS SDK in Kotlin.
Al momento della condivisione dei dati dell'utente viene effettuato un dump del database in formato JSON: i dati vengono strutturati in questo formato tramite la popolare libreria Gson, per poi essere salvati su un file che viene caricato su un bucket tramite una richiesta formulata da un AwsS3Client come nel seguente codice:

```
fun uploadUserData() {
    viewModelScope.launch(Dispatchers.Default) {
        val walkActivities =
            db.UserWalkActivityDao().getWalkActivitiesByUserId(User.id)
        val vehicleActivities =
            db.UserVehicleActivityDao().getVehicleActivitiesByUserId(User.id)
        val stillActivities =
            db.UserStillActivityDao().getStillActivitiesByUserId(User.id)

        val userData = mapOf(
            "username" to User.username,
            "walkActivities" to walkActivities,
            "vehicleActivities" to vehicleActivities,
            "stillActivities" to stillActivities
        )

        val gson = Gson()
        val json = gson.toJson(userData)

        val file = File(context.filesDir, "user_data.json")
        file.writeText(json)

        withContext(Dispatchers.IO) {
            try {
                val request = PutObjectRequest(BUCKET_NAME,
                    "user/${User.username}.json", file)
                s3Client.putObject(request)
            } catch (e: Exception) {
                Log.e("S3Upload", "Failed to upload user data", e)
            }
        }
    }
}
```

Per cercare un amico l'utente digita lo username della persona desiderata nella barra di ricerca, viene dunque effettuata una richiesta di *getObject* al bucket, che restituisce i dati di tutti gli utenti che hanno come prefisso dello username la stringa inserita dall'utente nella barra di ricerca.

Possibili sviluppi futuri

In un ulteriore sviluppo dell'applicazione sarebbe possibile migliorare i seguenti aspetti:

- **Maggiore sicurezza** nella condivisione dei dati in cloud, per garantire l'accesso solo agli utenti autorizzati
- **Miglioramento interfaccia** ed esperienza utente, per garantire un facile accesso a tutte le funzionalità

- **Aumento dei servizi in background**, come la registrazione automatica delle attività, per eliminare la necessità dell'avvio e dell'interruzione di una attività da parte dell'utente.
- **Miglioramento gestione batteria**, con riduzione delle prestazioni dell'app in caso di batteria scarica
- Possibilità di **eliminare amici** appartenenti alle proprie amicizie