Relazione Progetto Sviluppo Applicazioni Mobili

Samuele Calugi
Corso A
579086

Luglio 2022

Contents

1 Introduzione

Il progetto VoxelGO consiste nello sviluppo di un'applicazione simile a Pokémon GO: l'utente ha la possibilità di muoversi all'interno del mondo reale visualizzando una mappa all'interno dell'applicazione. Vengono generati casualmente dei Marker sulla mappa che segnalano la posizione di un collezionabile, ovvero di un modello 3D.
L'utente ha quindi la possibilità di cliccare sul Marker, dopo che ha raggiunto una certa distanza dall'obiettivo e ottenere quindi il suo collezionabile. Quest'ultimo viene conservato all'interno di un database dentro l'applicazione.
Per poter mostrare i collezionabili, è stata implementata una WebView che renderizza una pagina web contentente al suo interno il modello 3D del collezionabile. Il collezionabile può essere visualizzato soltanto dopo che l'utente è riuscito a catturarlo.

Per riassumere, il progetto VoxelGO contiene:

- integrazione dei servizi di Google con le api di GoogleMaps.
- database interno all'applicazione per memorizzare i collezionabili e le loro catture.
- integrazione delle WebView con una RestAPI.

1.1 Linguaggio di programmazione e librerie utilizzate

L'applicazione è stata sviluppata in Java e resa disponibile su github. Le librerie sono state implementate nel progetto con Gradle e sono facilmente visibili all'interno del file **build.gradle**. Le librerie utilizzate sono le seguenti:

- **room**, utilizzata per la gestione del database all'interno dell'applicazione.
- **play-services-maps**, utilizzata per implementare la mappa di Google.
- **play-services-location**, utilizzata per acquisire la posizione dell'utente tramite i servizi di Google.
- **viewmodel**, utilizzata per tenere alcuni dati in memoria anche dopo la distruzione e la ricreazione di una view.
- **livedata**, utilizzata per osservare il cambiamento di strutture dati mutabili all'interno dell'applicazione.
- **lottie**, utilizzata per inserire delle view animate.
- **glide**, utilizzata per scaricare immagini dalla rete e per cacharle all'interno dell'applicazione.
- **commons-io**, utilizzata come meccanismo di IO nell'applicazione.
- **gson**, utilizzata per deserializzare il file JSON inviato dalla Rest.

2 Struttura dell'applicazione

2.1 Activity e Fragment

L'applicazione è stata sviluppata creando un'unica Activity. Le schermate dell'applicazione sono rappresentate dai dei Fragment. La classe MainActivity contiene al suo interno un FrameLayout vuoto, chiamato **fragment_container**. Grazie all'utilizzo del FragmentManager, quest'ultimo viene continuamente sostituito con uno dei Fragment dell'applicazione, in base alla schermata che si vuole mostrare sul telefono. I Fragment presenti all'interno dell'applicazione sono:

- **FragmentHome**, mostra l'insieme di tutti i collezionabili che l'utente può catturare.
- **FragmentMap**, mostra la mappa implementata con le librerie di Google in cui l'utente può catturare i collezionabili.
- **FragmentProfile**, mostra i collezionabili catturati.
- **FragmentCollectible**, mostra il modello 3D del collezionabile catturato tramite WebView.

2.2 Database dell'applicazione

Il database è composto da due tabelle:

- **collectibles**, contiene l'insieme di tutti i collezionabili che l'utente può catturare.
- **captures**, contiene tutte le catture effettuate dall'utente

Come è possibile vedere dal codice, la struttura del database è stata implementata utilizzando la libreria room fornita da Google. Quest'ultima prevede:

- la realizzazione di una classe Database, chiamata in questo caso **VoxelRoomDatabase**, che si occupa in maniera autonoma di creare il file **.db** e salvarlo nella memoria interna del telefono. Il file **.db** viene poi utilizzato e interpretato da SQLite. Per poter eseguire operazioni sul database da parti distinte dell'applicazione, è stato necessario implementare la classe **VoxelRoomDatabase** come Singleton. In questo modo, tale classe viene istanziata in memoria un'unica volta e viene eliminata dal GarbageCollector al momento della terminazione dell'applicazione. Quando è necessario interagire con il database, quindi, è sufficiente prendere il suo riferimento in memoria con il metodo **getInstance()**.
- la realizzazione di una classe DAO (Data Access Object), generalmente almeno una per tabella, che contiene le query che possono essere eseguite sul database.
- Poichè tutte le operazioni eseguite sul database (quindi tramite l'utilizzo di un DAO) devono essere fatte su Thread secondari, così da non bloccare l'esecuzione del ThreadUI, è oppurtono definire anche una classe di tipo Repository. La classe Repository si occupa di fornire l'insieme di tutte le operazioni che possono essere fatte sul database tramite DAO. Questa è la classe che viene presa come riferimento ogni volta che si vuole eseguire operazioni sul database. Di conseguenza, anche la classe Repository è di tipo singleton, quindi viene istanziata una sola volta e viene preso il suo riferimento in memoria dalle altre classi quando è necessario fare operazioni sul database.

Per riassumere, quindi, ogni volta che è necessario fare delle operazioni sul database:

1. la classe Repository contiene al suo interno il DAO della tabella su cui devono essere eseguite le query.
2. per ogni query che il DAO può eseguire, la Repository fornisce un metodo pubblico che può essere chiamato dalle altre classi.
3. quando una classe vuole eseguire un operazione sul database, quindi, prende il riferimento in memoria della Repository e chiama uno dei suoi metodi.

4. il metodo invocato, viene eseguito da una pool di Thread creati tramite Executor, quest'ultimo dichiarato e istanziato all'interno della classe **VoxelRoomDatabase**.

Questa struttura permette quindi di eseguire operazioni concorrenti sul database e permette di istanziare il database una sola volta per tutto il ciclo di vita dell'applicazione.
Come già anticipato, avendo all'interno del database due tabelle, sono state realizzate più Repository e più DAO. L'intera struttura contiene le seguenti classi:

- **VoxelRoomDatabase**, la classe che si occupa di creare il database locale.
- **Collectible**, la classe che rappresenta le entry della tabella collectibles.
- **CollectibleDao**, insieme di query che possono essere eseguite sulla tabella collectibles.
- **CollectibleRepository**, la classe che viene invocate per eseguire le operazioni sulla tabella collectibles.
- **Capture**, la classe che rappresenta le entry della tabella captures.
- **CaptureeDao**, insieme di query che possono essere eseguite sulla tabella captures.
- **CaptureRepository**, la classe che viene invocate per eseguire le operazioni sulla tabella captures.
- **CapturedCollectiblesDao**, l'insieme di query che possono essere eseguite sulla tabella collectibles + captures.
- **CapturedCollectiblesRepository**, la classe che viene invocate per eseguire le operazioni sulla tabella collectibles + captures.

collectible_id	collectible_name	collectible_model	collectible_image	collectible_rarity
id del collezionabile usato come chiave primaria con auto increment	nome del collezionabile	url del collezionabile contente il modello 3D usato dalla WebView	url dell'immagine di anteprima del modello	rarità del modello

Figure 1: Tabella dei collezionabili

capture_id	collectible_id	capture_date	capture_location
id della cattura usato come chiave primaria con autoincrement	id del collezionabile catturato usato come chiave esterna	data di cattura del collezionabile	luogo di cattura del collezionabile

Figure 2: Tabella delle catture

2.3 Inserimento dei collezionabili nel database

Quando l'applicazione viene avviata, la classe MainActivity mette in esecuzione un Thread che implementa l'interfaccia **DownloadThread**. Quest'ultimo si occupa di scaricare dalla rete la lista di collezionabili che l'utente può catturare e li inserisce all'interno del database.
La RestAPI fornisce due differenti endpoint:

- endpoint=models/list, contiene la lista di tutti i collezionabili da inserire all'interno del database. La lista è rappresentata in formato JSON.
- endpoint=db-version, contiene la versione del database lato server. Questo endpoint viene contattato dal Thread per sapere se deve scaricare nuovi collezionabili o meno.

Se è la prima volta che l'utente avvia l'applicazione, quindi non ha mai scaricato un collezionabile fino ad ora, il Thread contatta l'endpoint endpoint=models/list, deserializza la lista in formato JSON inviata dal server ed inserisce i collezionabili all'interno del database.
Successivamente il Thread contatta l'endpoint endpoint=db-version e salva la versione del database (rappresentata come intero) in un file situato nella memoria interna del telefono.

Se l'utente ha già avviato l'applicazione in passato e ha quindi già scaricato dei collezionabili dal server, il Thread contatta prima l'endpoint endpoint=db-version e verifica se la versione del database del server è la stessa di quella che ha memorizzato all'interno del telefono. Se le due versioni coincidono, significa che il server non ha fatto modifiche al database e quindi non è necessario scaricare la lista di collezionabili.
Se le due versioni sono differenti, significa che ci sono dei nuovi collezionabili da inserire nel database. A questo punto il Thread contatta l'endpoint endpoint=mo-dels/list, scarica i nuovi collezionabili e aggiorna il file contenente in memoria la versione del database.

2.4 Funzionamento della WebView

Come descritto precedentemente, la WebView viene utilizzata all'interno del FragmentCollectible per visualizzare il modello 3D del collezionabile.
Per risparmiare il consumo di risorse di rete, la WebView implementa al suo interno un meccanismo di caching. Quest'ultimo è fornito internamente da Android. Attraverso il metodo **webSettings.setCacheMode()**, infatti, è possibile stabilire quale tipo di cache voler utilizzare per lo scaricamento della pagina web. Poichè il modello 3D una volta scaricato non cambia, è stato scelto di implementare come politica di cache **WebSettings.LOAD_CACHE_ELSE_NETWORK**. In questo modo, se la pagina web è presente in cache, allora verrà visualizzata senza dover scaricare la pagina dalla rete.

L'url che la pagina web contatta contiene due parametri di richiesta in GET:

- **model=[nome-modello]**, serve per decidere quale modello 3D renderizzare a schermo
- **mode=[light / dark]**, serve per cambiare il colore di sfondo della pagina web. Viene utilizzato per la modalità chiara o scura del telefono.

2.5 Funzionamento della Mappa

I collezionabili vengono generati in maniera randomica attorno all'utente e la loro posizione rimane in memoria finché l'applicazione non viene chiusa. Quando l'utente si muove nella mappa, dopo una certa distanza, i collezionabili più lontani vengono rimossi e vengono nuovamante generati attorno al giocatore. L'utente non ha la possibilità di muovere la telecamera, ma essa è bloccata sulla sua posizione e segue tutti i suoi spostamenti. I collezionabili sono rappresentati sotto forma di Marker e vengono gestiti dalla classe **MarkerHandler**.

La classe MarkerHandler si occupa di gestire la creazione, la generazione e la cattura dei collezionabili, rappresentati all'interno della mappa come Marker. Per tenere in memoria la posizione dei Marker sulla mappa anche quando l'utente cambia schermata, muovendosi per esempio fra i vari Fragment dell'applicazione, è stato deciso di implementare una classe di tipo Singleton. In questo modo la classe MarkerHandler rimane in memoria e non viene distrutta dal GarbageCollector, finché l'utente (o il sistema) non decide di terminare l'applicazione.

Per catturare un collezionabile, è quindi sufficiente: cliccare su un Marker, premere sul menù a tendina che compare dopo aver cliccato e se l'utente è abbastanza vicino, il collezionabile viene catturato e il Marker rimosso dalla mappa.
Come descritto nella documentazione, tutto il Funzionamento della mappa viene implementato all'interno del metodo **onMapReady()**, quest'ultimo invocato da una chiamata asincrona dal metodo **getMapAsync();**
il metodo onMapReady(), presente all'interno del FragmentMap, implementa 4 metodi:

- **updateLocationUI()**, modifica l'aspetto, le impostazioni e registra i listener della mappa
- **getLocationPermission()**, controllo i permessi di geolocalizzazione. Se non presenti, viene mostrata la richiesta di autorizzazione a schermo
- **getDeviceLocation()**, se l'utente ha conferito i permessi di geolocalizzazione, questo metodo tramite callback si occupa di prendere la posizione dell'utente
- **getMarkersLocation()**, se l'utente ha conferito i permessi di geolocalizzazione, quest metodo si occupa di generare i Marker attorno all'utente