Alma Mater Studiorum | **Università di Bologna**

LABORATORIO DI APPLICAZIONI MOBILI

Laurea in Informatica per il Management

Realtime Geolocalizzed Treasure Hunting App

Student's name: **Jake**
Matricola: **XXXXXXX**
Email: ***xxxxxxxx@studio.unibo.it***

App Overview

CityGem Go! è un'app per smartphone che rende l'esplorazione della città un gioco divertente, simile a una *caccia al tesoro*. L'obiettivo è aiutare gli utenti a scoprire i luoghi più interessanti e nascosti della città, seguendo un percorso speciale. Ogni tappa del gioco rappresenta un posto particolare, che bisogna raggiungere per completare il percorso. Per scoprire quale sarà la prossima tappa, l'utente deve rispondere a un indovinello che offre indizi su dove andare. Una volta capito il posto, bisogna recarsi lì fisicamente per avanzare nel gioco.

Quando si arriva al luogo indicato, l'app richiede di scattare un selfie per dimostrare di essere davvero sul posto. Utilizzando la geolocalizzazione, l'app controlla che la foto sia stata scattata nel punto giusto, e solo allora permette di andare avanti alla prossima tappa. Il gioco continua così, tappa dopo tappa, fino a completare l'intero percorso. Alla fine, l'app salva il tempo impiegato per completare il gioco e lo invia al server, permettendo all' utente di vedere come si posiziona rispetto ai dieci migliori giocatori che hanno completato lo stesso percorso.

Le foto scattate durante il gioco vengono salvate direttamente sul cellulare dell'utente, così da poterle condividere facilmente sui social e mostrare a tutti la propria esperienza. CityGem Go! unisce gioco, scoperta e tecnologia, offrendo un modo nuovo e divertente per esplorare la città, imparando e divertendosi allo stesso tempo.

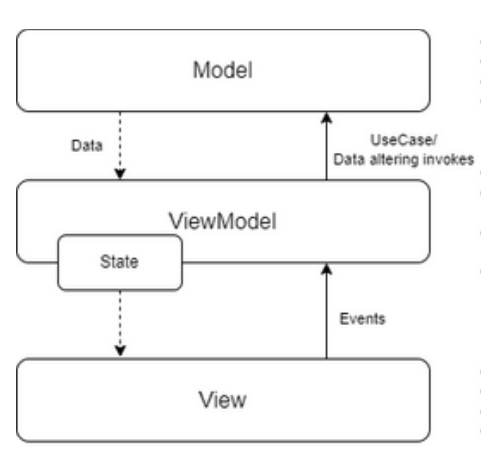
Tecnologie

Per realizzare CityGem Go! è stato usato **React Native** con **Expo** per lo sviluppo *cross-platform* permettendo di scrivere l'app una sola volta e distribuirla sia su *Android* che su *iOS*. Expo semplifica la configurazione e integra facilmente funzionalità essenziali come la fotocamera, utilizzata per acquisire le foto dei partecipanti, e le API di *geolocalizzazione*, impiegate per verificare la posizione dell'utente nelle varie tappe.

Per il linguaggio è stato scelto **TypeScript** contribuisce a migliorare la sicurezza del codice grazie alla tipizzazione statica.

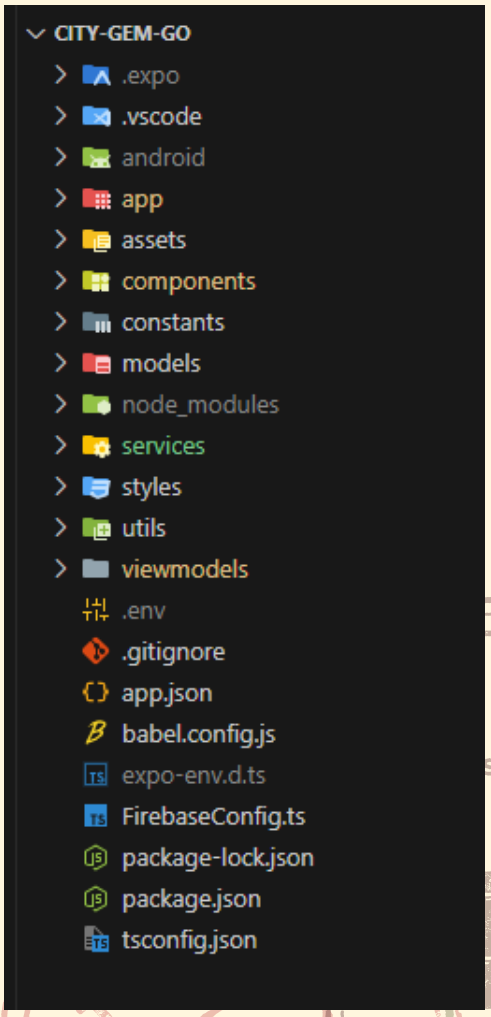
Firebase è stato scelto per gestire l'autenticazione e i dati cloud, mentre **SQLite** è usato per il salvataggio locale dei dati e **AsyncStorage** per memorizzare rapidamente i dati temporanei necessari durante l'uso dell'app.

Architettura del Progetto



Il progetto segue il pattern architetturale **MVVM** (**Model-View-ViewModel**), che facilita la separazione della logica di business dalla logica di presentazione. In questo schema, i **Model** gestiscono i dati e le interazioni con le risorse esterne, le **View** sono responsabili della presentazione e dell'interfaccia utente, mentre i **ViewModel** fungono da intermediari, mantenendo la logica di business e fornendo i dati alla View senza che questa debba interagire direttamente con il Model.

La struttura del progetto riflette questa separazione. La cartella **app** contiene i file relativi alle varie schermate e alla configurazione della navigazione dell'applicazione, costituendo l'interfaccia visibile agli utenti. La cartella **assets** raccoglie risorse come font e immagini usate all'interno dell'applicazione. Nella cartella **components** si trovano i componenti riutilizzabili dell'interfaccia, permettendo di costruire una UI modulare. La cartella **constants** ospita variabili e valori costanti utilizzati in più parti dell'app per garantire coerenza, come i colori e altre configurazioni statiche. La cartella **models** gestisce i dati e le strutture utilizzate dall'applicazione, seguendo il pattern architetturale MVVM. La cartella **services** contiene le implementazioni dei servizi esterni e interni, come la gestione delle notifiche e il database, responsabili dell'interazione con risorse esterne dall'app. La cartella **styles** è dedicata agli stili, dove ogni file di stile definisce l'aspetto di specifiche parti dell'interfaccia utente. La cartella **utils** include utility e funzioni ausiliarie per operazioni generiche, mentre la cartella **viewmodels** ospita la logica di presentazione e funge da ponte tra i modelli e le viste, in linea con il pattern MVVM, fornendo i dati alle schermate dell'app.

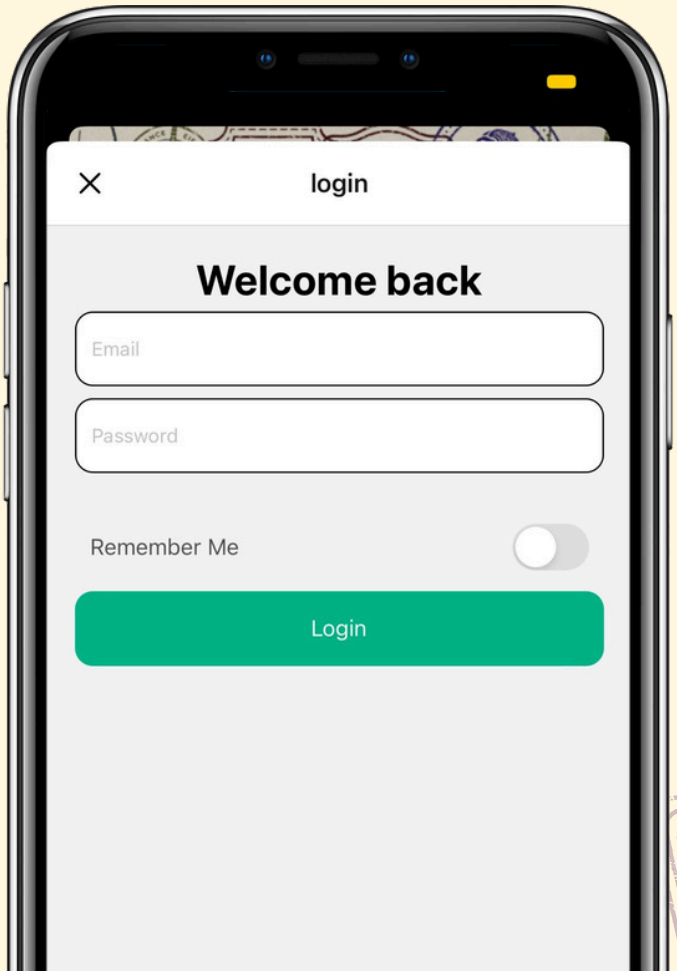


Implementazioni

Avvio dell’app

All'avvio dell'app, l'utente viene accolto da una schermata introduttiva contenente un componente **TitleCard** che visualizza il titolo e il logo dell'app. Sotto la TitleCard, è presente un gruppo di pulsanti, gestiti dal componente **LoginRegisterButton**, che permette all'utente di scegliere tra le opzioni di autenticazione. Il pulsante "**Log in**" consente di accedere direttamente alla schermata di login per chi ha già un account, mentre il pulsante "**Register with Email**" reindirizza alla schermata di registrazione per creare un nuovo profilo.

Questa schermata introduttiva è progettata per fornire un accesso semplice e immediato alle opzioni di autenticazione. Inoltre, all'interno dell'**useEffect**, viene istanziato il **NotificationHandler**, che richiede i *permessi di notifica* all'utente, assicurando che l'app possa inviare **notifiche push** in futuro.

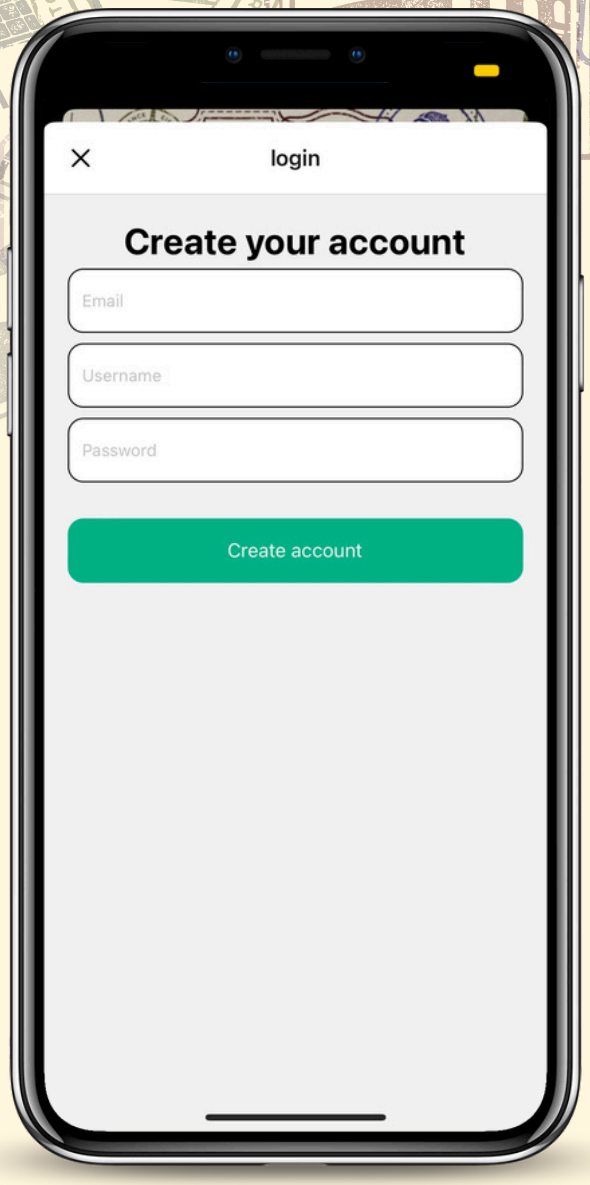


Login

Quando l'utente seleziona l'opzione di **Login**, accede a una schermata dove può inserire le proprie credenziali, **email** e **password**, per effettuare l'accesso. L'opzione "***Ricordami***", disponibile tramite uno switch, consente di salvare le credenziali su **AsyncStorage**, facilitando un accesso più veloce nelle sessioni future.

La **LoginViewModel** gestisce lo stato delle credenziali e la logica di autenticazione. Una volta che le credenziali vengono inserite e con una connessione Internet attiva, queste vengono inviate a **Firebase** per la verifica. In caso di successo, l'app richiama Firebase per ottenere lo **username** associato all'account, che viene memorizzato su **AsyncStorage**; solo dopo questa operazione l'utente viene reindirizzato alla schermata principale dell'applicazione.

Implementazioni



Registrazione

Per creare un nuovo account, l'utente accede alla schermata di registrazione e inserisce *email*, *username* e *password*. Utilizzando la **LoginViewModel**, l'app raccoglie questi dati e attraverso la **LoginModel** invia una richiesta a **Firebase**, che crea l'account e salva i dettagli dell'utente nel database. Al completamento della registrazione, l'username appena registrato viene salvato su **AsyncStorage** per essere utilizzato durante il gioco.

In seguito, l'utente viene automaticamente reindirizzato alla schermata principale dell'app, dove può iniziare a utilizzare i servizi offerti. In caso di errore, come un'email già associata a un account esistente, l'app mostra un messaggio di avviso per informare l'utente e invita a modificare le credenziali fornite per completare la registrazione.

Routes

Prima di caricare la schermata principale, la **RoutesViewModel** chiama la **RouteModel** tramite il metodo **getLocalRoutes()**. La RouteModel si interfaccia con la classe **DatabaseManager**, la quale esegue una chiamata in background al database di **Firebase** per ottenere gli itinerari e gli indovinelli disponibili in base alla posizione attuale dell'utente. Questi dati, attraverso il DatabaseManager, vengono salvati localmente nel database **SQLite** del dispositivo per ottimizzare le prestazioni.

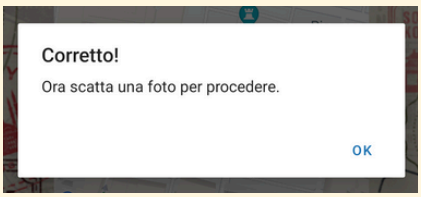
Una volta completato il salvataggio locale, la RouteModel ottiene l'elenco degli itinerari disponibili utilizzando **getLocalRoutes()** e lo fornisce alla RoutesViewModel.

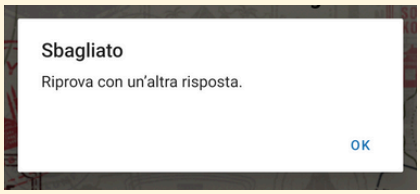
Implementazioni

Routes (cont.)

Successivamente, l'utente accede alla schermata principale, dove viene mostrata una lista di itinerari filtrati in base alla posizione rilevata. Quando l'utente seleziona un itinerario, l'app procede alla schermata successiva, passando la **routeId** selezionata come parametro. Questo permette alla schermata successiva di caricare gli indovinelli e le informazioni specifiche per l'itinerario scelto.

Nella parte inferiore destra della schermata è presente un pulsante "**Help**", un componente gestito da **FloatingButton**, che mostra le istruzioni di gioco. Dopo aver letto le istruzioni, l'utente può tornare alla schermata principale attraverso un pulsante dedicato alla fine delle indicazioni.

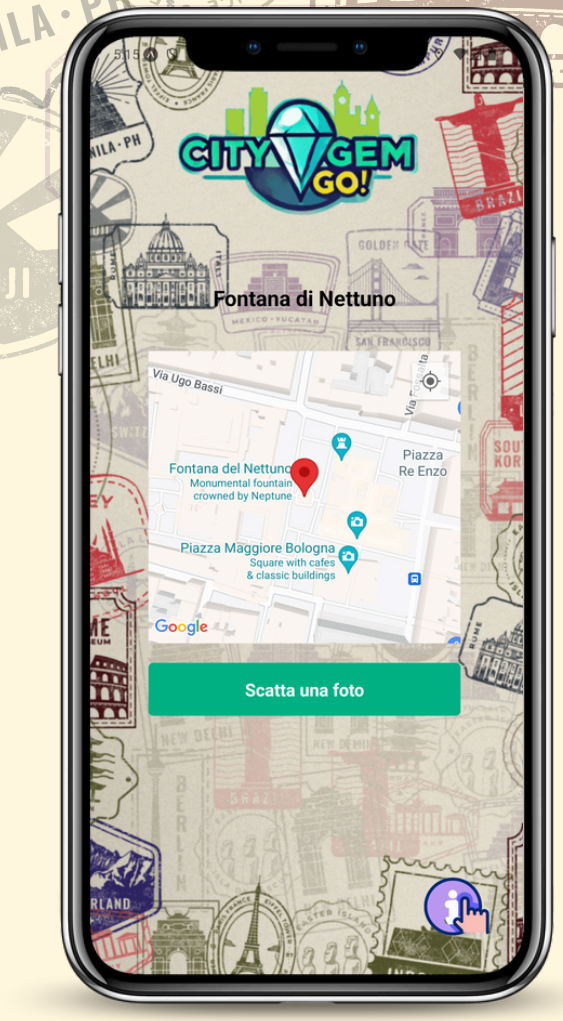




Riddles

In base all'itinerario selezionato, la **RiddlesViewModel** carica tutti gli indovinelli tramite la funzione **loadRiddles** della **RiddleModel**, che ottiene gli indovinelli dal database e li salva in locale. La funzione **fetchRiddles** nella RiddlesViewModel utilizza **setRiddles** per memorizzare gli indovinelli nello stato e avvia il gioco impostando **currentIndex** a **0**, visualizzando così il primo indovinello all'utente.

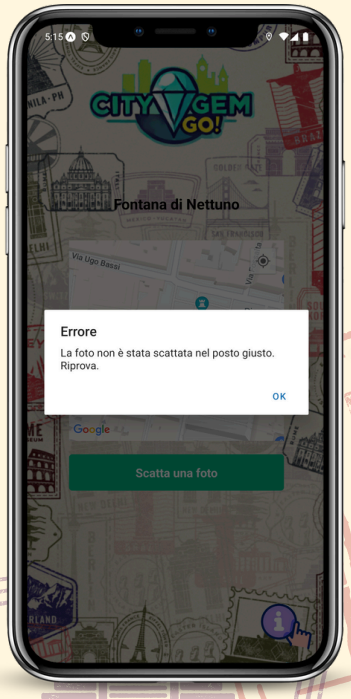
Per ciascun indovinello, viene mostrata una domanda che richiede una risposta testuale che rappresenta il nome del luogo da raggiungere. La RiddlesViewModel gestisce la verifica della risposta dell'utente tramite il metodo **checkAnswer**, confrontando l'input con la risposta corretta dell'indovinello attuale, ottenuta con **getCurrentRiddle**. Se la risposta è corretta, **setIsCorrect(true)** viene attivato, e l'utente vede un messaggio di conferma; in caso contrario, **setIsCorrect(false)** e l'input viene ripulito per consentire un nuovo tentativo.



Riddles (cont.)

Quando la risposta è corretta, la mappa del luogo da raggiungere viene mostrata insieme a un pulsante per scattare una foto. L'utente deve recarsi nel luogo e scattare una selfie con il punto di riferimento in vista. La verifica della posizione viene gestita tramite **handlePhotoVerification**, che riceve un parametro booleano **success** (true se la posizione è corretta). Se **isLastRiddle()** risulta **true**, la chiamata a **UserSessionManager.completeSession** calcola il tempo totale impiegato per completare il percorso, aggiornando **setCompletionTime** con questo valore. Se non è l'ultima tappa, **moveToNextRiddle()** aggiorna currentIndex, caricando l'indovinello successivo e reimpostando gli stati **isCorrect**, **userAnswer** e **photoVerified** per la nuova tappa.

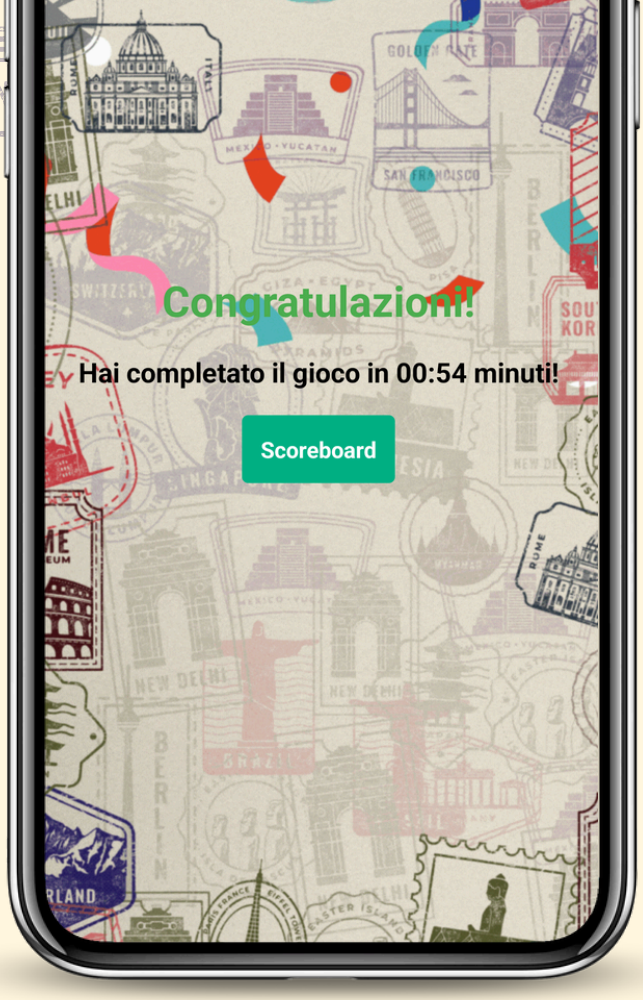
L'interfaccia utente rimane sempre sulla stessa pagina, aggiornando soltanto i contenuti per ogni nuovo indovinello. Al completamento di tutte le tappe, l'utente viene indirizzato alla schermata dei risultati, dove il tempo totale, calcolato e memorizzato in **completionTime**, viene visualizzato. Durante l'intera sessione, la **RiddlesViewModel** interagisce con **UserSessionManager** per gestire le tempistiche di inizio e fine del percorso e con **NotificationHandler** per inviare promemoria all'utente, incoraggiandolo a completare l'avventura.

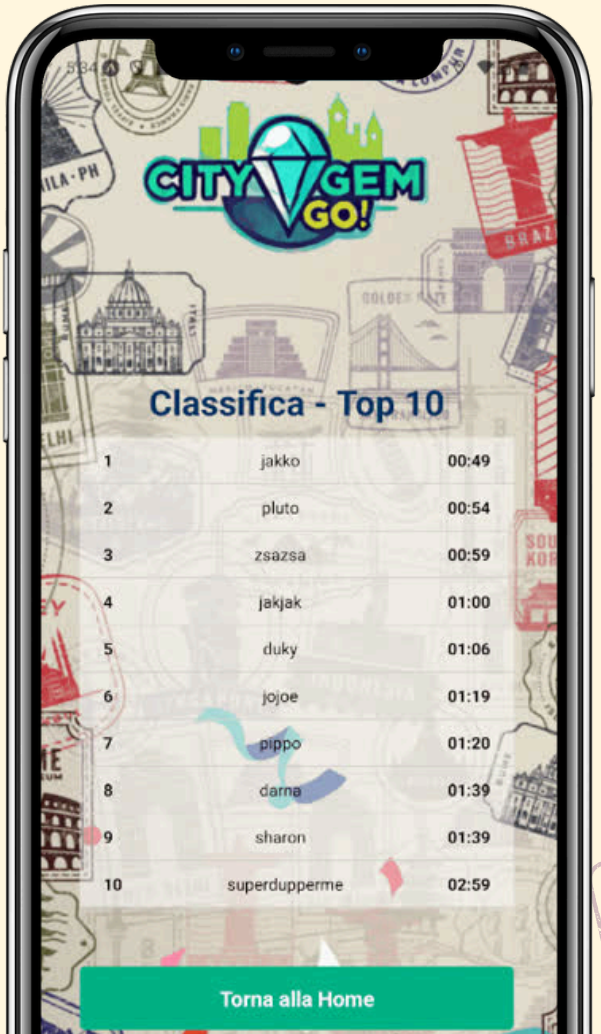


Implementazioni

Results

Al completamento del gioco, viene mostrata una schermata di riepilogo con il tempo totale impiegato, passato come parametro **totalTime** dalla schermata precedente. Il valore di totalTime viene formattato in minuti tramite la funzione **formatTime** di **timeFormatter**, un componente stand-alone, e visualizzato insieme a un messaggio di congratulazioni. L'interfaccia include un pulsante "Scoreboard" che permette di procedere alla schermata successiva, dove l'utente può visualizzare i migliori tempi di completamento dell'itinerario. Quando viene premuto il pulsante, la funzione **handleRouteSelect** utilizza **router.push** per navigare allo scoreboard, passando l'routeId selezionato come parametro, così da visualizzare i risultati relativi all'itinerario completato.





Scoreboard

La schermata dello scoreboard mostra i migliori 10 giocatori per l'itinerario selezionato, offrendo una visuale delle performance degli utenti in base al routeId scelto. La **ScoreboardViewModel** utilizza la funzione **fetchTopScores** della **ScoreModel** per recuperare i punteggi dal database, passando il **routeId** come parametro. I risultati vengono salvati nello stato scores e visualizzati nella View tramite una lista (**FlatList**), dove ogni elemento mostra il posizionamento, il nome dell'utente e il tempo totale formattato tramite **formatTime**.

Durante il caricamento dei dati, viene mostrata un'animazione di attesa per migliorare l'esperienza dell'utente. Se non ci sono dati disponibili, viene visualizzato un messaggio informativo. Alla fine della classifica, un pulsante consente all'utente di tornare alla schermata principale, richiamando **router.replace('/routes')** per riportare l'utente alla Home e consentirgli di selezionare un nuovo itinerario.

Implementazioni

FirebaseManager

La classe FirebaseManager gestisce operazioni CRUD su documenti in una collezione Firebase, inclusi il recupero, l'aggiornamento, l'aggiunta e l'eliminazione. Usa getDocument per ottenere un singolo documento e getAllDocuments per recuperare tutti i documenti di una collezione. Fornisce anche metodi avanzati per ottenere documenti ordinati (getOrderedDocuments), documenti con filtri specifici e ordinati (getFilteredAndOrderedDocuments), e documenti filtrati (getFilteredDocuments). Per aggiungere, aggiornare o eliminare documenti, utilizza rispettivamente addDocument, updateDocument e deleteDocument, assicurando una gestione strutturata e robusta dei dati nel database Firebase.

SQLiteManager

La classe SQLiteManager gestisce il database locale, utilizzando una tabella adventures per memorizzare informazioni su ogni domanda e posizione associate a un'avventura specifica. La funzione resetDatabase elimina e ricrea la tabella, mentre init assicura che la tabella esista. Il metodo upsertAdventure inserisce o aggiorna i record delle avventure, garantendo che le modifiche si riflettano correttamente. getAdventures recupera un elenco univoco di avventure, e getQuestionsById permette di ottenere tutte le domande associate a un determinato adventure_id. Infine, deleteAdventureQuestion elimina una domanda specifica in base al question_id.

NotificationHandler

La classe NotificationHandler gestisce le notifiche e il monitoraggio dello stato dell'app per inviare promemoria all'utente. Configura le impostazioni di notifica, richiede i permessi e permette di programmare notifiche con titolo e messaggio specifici tramite scheduleNotification. Inoltre, rileva i cambiamenti di stato dell'applicazione, programmando una notifica se l'app passa in background per più di 10 secondi. La funzione handleAppStateChange gestisce la transizione tra stati attivi e in background, annullando il timeout se l'app torna attiva, mentre addAppStateListener e removeAppStateListener aggiungono e rimuovono il listener per i cambiamenti di stato.

distanceCalculator

E' un componente stand-alone con una funzione getDistance che calcola la distanza in metri tra due coordinate GPS (latitudine e longitudine) utilizzando la formula dell'haversine, che considera la curvatura terrestre.

timeFormatter

Componente stand-alone, con la funzione formatTime che converte un valore di tempo in millisecondi in una stringa formattata come "MM:SS".

Lavori Futuri

Per migliorare l’esperienza utente e ampliare le funzionalità dell’app, in futuro è possibile implementare alcune estensioni. Potremmo aggiungere la possibilità di modificare le tappe direttamente durante il gioco, offrendo agli utenti una maggiore flessibilità e personalizzazione del percorso. Inoltre, sarebbe utile introdurre livelli di accesso differenziati, con ruoli di amministratore per consentire la gestione e configurazione delle avventure, e ruoli standard per i partecipanti. Infine, un’altra funzionalità interessante potrebbe essere l’integrazione della condivisione social, permettendo agli utenti di pubblicare le foto scattate direttamente dai loro profili sui principali social network.