RELAZIONE APP “TRAVEL COMPANION”

Membri del gruppo
Alessandro Campedelli, alessandr.campedell3@studio.unibo.it, 0001077616
Nicolas Cola, nicolas.cola@studio.unibo.it, 0001080556

Architettura Generale
Il progetto implementa un'architettura organizzata in due layer principali: data e presentation. Il layer data gestisce la persistenza, l'accesso ai dati e le loro rappresentazioni mentre il layer presentation si occupa dell'interfaccia utente e dell'interazione con l'utente. Ecco la struttura nel dettaglio.

```
├── data/           # Layer dati (Room, Repository)
├── domain/         # Modelli di dominio
├── presentation/   # UI, ViewModel, Fragment
├── di/             # Dependency Injection (Hilt)
├── service/        # Servizi background
├── util/           # Utilities e helper
```

L'applicazione adotta il pattern Model-View-ViewModel (MVVM), il quale facilita la separazione tra logica di business e interfaccia utente migliorando la manutenibilità del codice.
Il ViewModel agisce come intermediario tra le View (Fragment) e il Model (Repository), gestendo lo stato della UI e i cambi di configurazione. L'implementazione utilizza LiveData e StateFlow per la comunicazione reattiva, garantendo aggiornamenti automatici dell'interfaccia quando i dati cambiano.

Il progetto utilizza un approccio Single Activity con Navigation Component portando così vantaggi in termini di performance e user experience. L'unica Activity (MainActivity) funge da contenitore per tutti i Fragment, gestendo la navigazione attraverso il NavController. Questa architettura elimina l'overhead legato alla creazione e distruzione di multiple Activity, garantendo transizioni più fluide e una migliore gestione dello stato applicativo. I Fragment rappresentano le singole schermate dell'applicazione e utilizzano Data Binding per il collegamento con i rispettivi ViewModel.
Ogni Fragment è annotato con *@AndroidEntryPoint* per abilitare l'iniezione delle dipendenze tramite Hilt. L'utilizzo del by *viewModels() delegate* garantisce la corretta gestione del lifecycle e la condivisione dei ViewModel quando necessario.

Per la gestione dei dati locali è stato scelto Room. Esso garantisce la verifica a tempo di esecuzione delle query SQL e integrazione seamless con l'architettura Android.

Il database è configurato con cinque entità principali (Trip, Coordinate, Photo, Note, POI) che rappresentano il domain model dell'applicazione.

I Data Access Object (DAO) definiscono le operazioni disponibili sul database utilizzando annotation Room che generano automaticamente l'implementazione delle query.

Il Repository Pattern fornisce un'interfaccia unificata per l'accesso ai dati, astraendo la sorgente dati specifica. I Repository fungono da accesso generalizzato per i dati dell'applicazione, centralizzando l'accesso.

Hilt è stato scelto come framework di dependency injection per la sua integrazione nativa con i componenti Android e la generazione automatica di codice a compile-time. La configurazione in AppModule definisce le dipendenze singleton come database, DAO e algoritmi di predizione. L'utilizzo degli scope Hilt (@*Singleton*) garantisce la corretta gestione del lifecycle delle dipendenze.

Panoramica dell'applicazione con funzionalità annesse

1. **Home page**

La schermata home rappresenta il punto d'ingresso principale dell'applicazione, progettata per fornire agli utenti una panoramica immediata del loro stato di viaggio attuale attraverso un'interfaccia pulita e informativa.

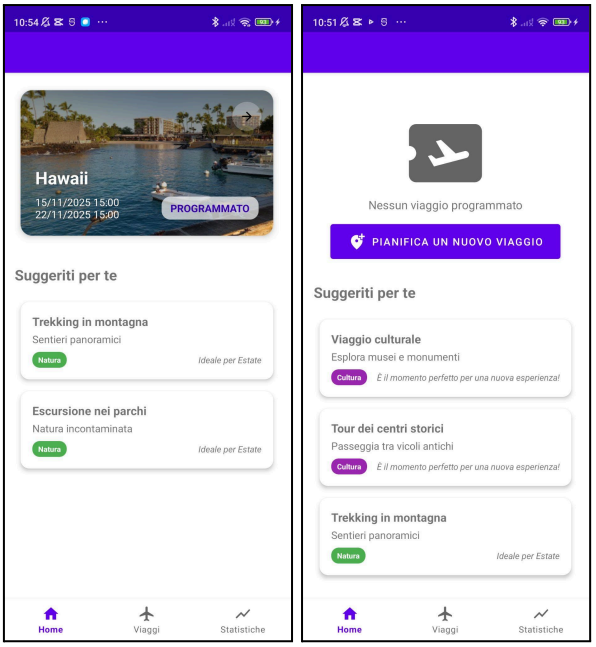
L'implementazione si basa sui seguenti elementi:

- *HomeFragment*: Gestisce la visualizzazione e le interazioni della schermata principale
- *HomeViewModel*: Coordina la logica di selezione e presentazione dei viaggi
- *EmptyStateHelper*: Gestisce gli stati vuoti con call-to-action appropriate

Il *HomeViewModel* implementa una strategia intelligente per determinare quale viaggio mostrare all'utente. Attraverso una `MediatorLiveData` denominata `tripToShow`, il sistema combina due flussi di dati:

- Viaggi in corso (`TripStatus.STARTED`): Massima priorità per viaggi attualmente attivi
- Viaggi pianificati (`TripStatus.PLANNED`): Visualizza il prossimo viaggio cronologicamente se nessun viaggio è in corso

La card viaggio funge da elemento interattivo per la navigazione verso i dettagli del viaggio. Inoltre quando non ci sono viaggi attivi o pianificati, il sistema utilizza *EmptyStateHelper* per mostrare un'interfaccia informativa con callback diretta per la creazione di nuovi viaggi.



2. Creazione di un viaggio

La funzionalità di creazione di un nuovo viaggio consente agli utenti di pianificare e registrare i propri viaggi.

I componenti principali dell'implementazione sono:

- *NewTripFragment*: fragment che gestisce l'interfaccia utente e le interazioni
- *TripsViewModel*: ViewModel che contiene la logica di business e gestisce i dati
- *TripRepository*: livello di accesso ai dati per le operazioni CRUD
- *TripScheduler*: servizio per la programmazione delle notifiche

All'interno del nostro progetto implementiamo l'API Places SDK for Android di Google. Essa ci permette di utilizzare le interfacce *Places* e *Autocomplete.* Più nello specifico *Places* fornisce accesso programmatico al database di Google di informazioni su attività e luoghi locali mentre *Autocomplete* fornisce widget predefiniti per restituire le previsioni relative ai luoghi in risposta alle query di ricerca degli utenti. In sostanza, implementando questa API, l'app consente all'utente di inserire destinazioni reali per i propri viaggi. Nel nostro caso specifico l'utente viene supportato in fase di ricerca tramite il completamento automatico dei luoghi disponibili e la visualizzazione di un'immagine relativa ad esso.

Una volta che l'utente ha cliccato sul bottone "CREA VIAGGIO" il *NewTripFragment* intercetta l'evento e richiama il *TripsViewModel.* Esso a sua volta eseguirà tutti i controlli sulla validazione dei campi inseriti; qualora l'esito di essi sia positivo viene creato un nuovo oggetto TripEntity che verrà inserito sul db tramite il metodo `addTrip(newTrip)` del *tripRepository*. (Nella pratica succede che il repository richiama il metodo del DAO).

Infine viene creato un allarme tramite il *TripScheduler:* un servizio specializzato nella programmazione e gestione delle notifiche relative ai viaggi, implementato per garantire che gli utenti ricevano promemoria tempestivi riguardo ai loro viaggi pianificati.

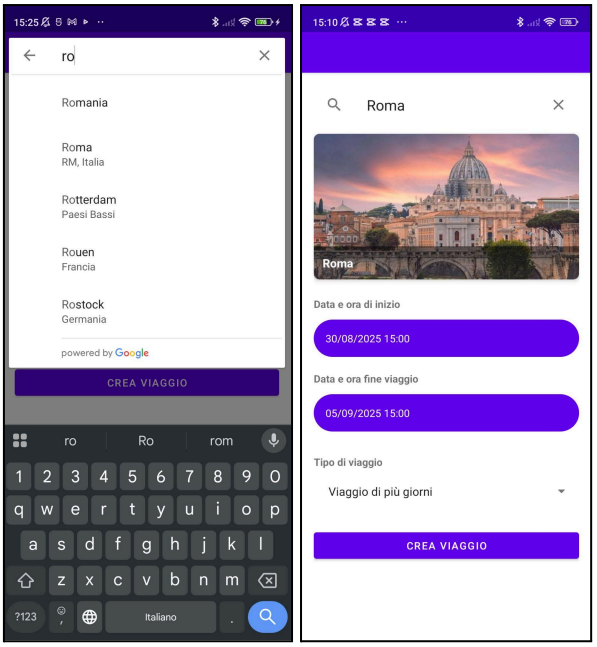
Il *TripScheduler* utilizza il sistema di allarmi di Android (*AlarmManager*) per programmare notifiche che vengono attivate automaticamente in momenti specifici. Quando un nuovo viaggio viene creato con successo, il sistema richiama il metodo `tripScheduler.scheduleTrip(id, newTrip.startDate, newTrip.endDate).` Esso si occupa di creare una notifica di avvio e una notifica di fine viaggio.
Abbiamo adottato questo tipo di implementazione per i seguenti vantaggi:

- Sincronizzazione temporale precisa: tramite l'inoltro di un intent specifico, il *TripScheduler* garantisce che le notifiche vengano inviate esattamente agli orari programmati indipendentemente dallo stato dell'applicazione, aggiornando lo stato dei viaggi sul db

- Ottimizzazione delle risorse: invece di mantenere l'app costantemente attiva per monitorare lo stato dei viaggi (polling), il *TripScheduler* delega questa responsabilità al sistema operativo, ottimizzando il consumo di batteria e le prestazioni del dispositivo.

- Affidabilità del sistema: le notifiche programmate tramite *AlarmManager* funzionano anche quando l'app non è in esecuzione o quando il dispositivo è in modalità risparmio energetico, garantendo che gli utenti ricevano sempre i loro promemoria.

Infine il *tripScheduler* è in grado di riprogrammare tutti gli allarmi dei viaggi già creati tramite il metodo: `rescheduleActiveTrips(plannedTrips: List<TripData>, startedTrips: List<TripData>).`
Il metodo `rescheduleActiveTrips` lavora in stretta collaborazione con il *TripStatusReceiver*, un *BroadcastReceiver* che funge da gestore centrale per tutti gli eventi di cambio stato dei viaggi. Quando `rescheduleActiveTrips` riprogramma gli allarmi dopo un riavvio del sistema, ogni allarme è configurato per inviare un intent specifico con action ACTION_UPDATE_TRIP_STATUS che viene intercettato dal *TripStatusReceiver*.
Quest'ultimo si occupa automaticamente di aggiornare lo stato del viaggio nel database (da PLANNED a STARTED o da STARTED a FINISHED) e di inviare la relativa notifica push all'utente. Questa architettura garantisce che il sistema di notifiche rimanga completamente autonomo e sincronizzato, assicurando agli utenti un'esperienza coerente anche dopo eventi esterni come riavvii del dispositivo o aggiornamenti dell'applicazione.



3. Sezione note di un viaggio

Prima di spiegare come è stata implementata la funzionalità delle note è necessario spiegare l'architettura degli adapter.
Essa si basa su una gerarchia a tre livelli, dove ogni livello aggiunge funzionalità specifiche mantenendo la compatibilità con il livello superiore:

- *BaseAdapter:* rappresenta la classe base che fornisce le funzionalità comuni a tutti gli adapter dell'applicazione.

- *SelectableAdapter*: estende il *BaseAdapter* aggiungendo le funzionalità di selezione multipla tramite:
    - composizione con il *SelectionManager* (manager di selezione generico)
    - gestione automatica dell'entrata/uscita dalla modalità di selezione
    - aggiornamenti ottimizzati solo per le modifiche di selezione tramite payload specializzati

Questa struttura garantisce:

- Riusabilità del codice: la stessa struttura viene infatti usata per il *TripsListAdapter*, *PhotoGalleryAdapter*, *NotesListAdapter* e *SuggestionAdapter*

- Consistenza comportamentale: tutti gli adapter condividono le stesse gesture di interazione e medesimi feedback visivi per gli stati di selezione e logica uniforme

- Manutenibilità: modifiche al comportamento di selezione nel *SelectableAdapter* e del *SelectionManager* si propagano automaticamente a tutti gli adapter specializzati, riducendo la duplicazione di codice e potenziali bug.

Un'osservazione deve essere fatta sul metodo `handlePayloadUpdate(holder: VH, position: Int, payloads: MutableList<Any>)` il quale permette di aggiornare solo gli

elementi visivi della selezione senza rifare il binding completo dei dati, ottenendo animazioni più fluide e minor consumo di risorse. Questo metodo permette quindi di migliorare le prestazioni dell'applicazione e risparmiare risorse.
Ecco infine un struttura tree per visualizzare meglio quanto spiegato:

```
BaseAdapter<T, VH>  [Livello Base]
        │   ├── Gestione DiffUtil
        │   ├── Sistema payload (PAYLOAD_SELECTION_CHANGED,
PAYLOAD_SELECTION_MODE_CHANGED)
        │   ├── Template methods (bindItem, onItemClick, onItemLongClick)
        │   └── Gestione click listeners unificata
        │
        ├── SelectableAdapter<T, VH>  [Livello Selezione Standard]
        │   │   ├── Integrazione SelectionManager<T>
        │   │   ├── Logica modalità selezione uniforme
        │   │   ├── API selezione (getSelectedItems, clearSelection)
        │   │   └── Ottimizzazioni payload per selezione
        │   │
        │   ├── NotesListAdapter  [Implementazione Note con Selezione]
        │   └── TripsListAdapter  [Implementazione Viaggi con Selezione]
        │
        ├── PhotoAdapter  [Implementazione Custom con Selezione Specializzata]
        │   ├── Estende: BaseAdapter direttamente
        │   └── SelectionManager<PhotoEntity> custom
        │
        └── SuggestionsAdapter  [Implementazione Senza Selezione]
```

Fatta questa premessa, di seguito viene spiegata l'implementazione delle note di un viaggio.

Il sistema di gestione delle note consente agli utenti di documentare esperienze, osservazioni e ricordi durante i loro viaggi. L'implementazione fornisce funzionalità complete di CRUD (Create, Read, Update, Delete) con supporto per selezione multipla e interfacce intuitive.

L'implementazione si basa sui seguenti componenti:

- *NotesListFragment*: gestisce la visualizzazione della lista delle note
- *NoteDetailsFragment*: gestisce la visualizzazione e modifica delle singole note
- *NotesViewModel*: contiene la logica di business e gestisce i dati delle note
- *NotesListAdapter*: adapter personalizzato con supporto per selezione multipla
- *SelectionManager*: manager generico per la gestione della selezione multipla

Quando l'utente naviga verso le note di un viaggio, il *NotesListFragment* richiede al *NotesViewModel* di caricare i dati. Il ViewModel utilizza switchMap per collegare dinamicamente l'ID del viaggio alle note corrispondenti, garantendo aggiornamenti automatici quando i dati cambiano nel database.

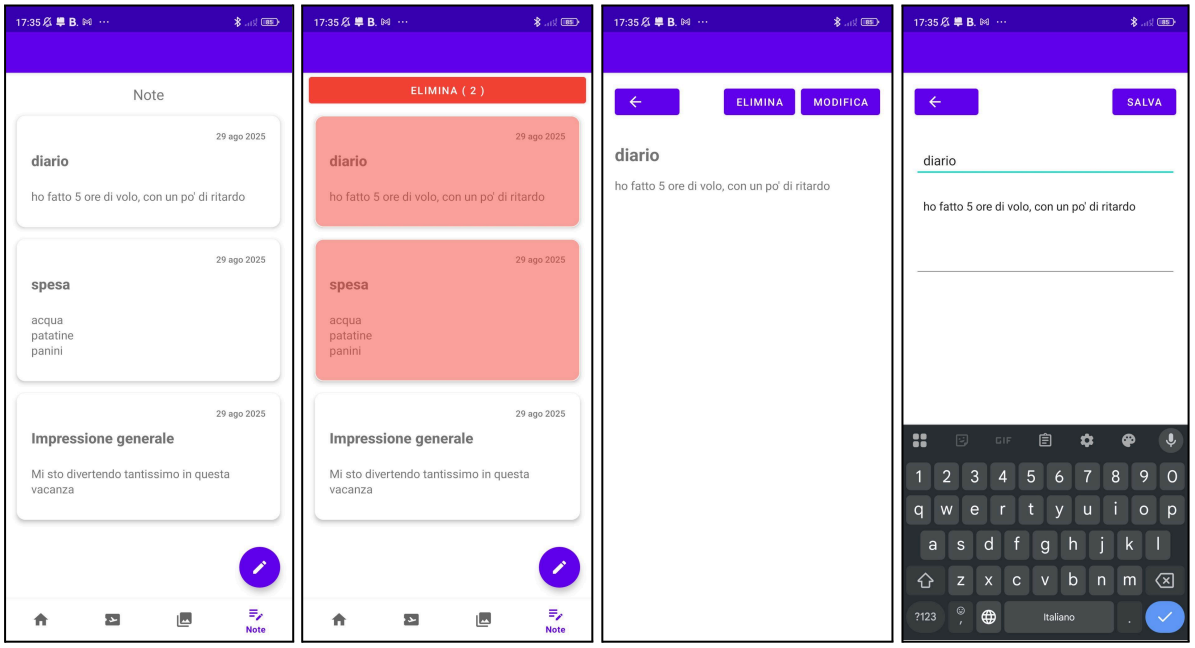
Il *NotesListAdapter* riceve la lista tramite observer pattern e collabora con il *SelectionManager* per gestire la selezione multipla.

Il *NoteDetailsFragment* gestisce due modalità distinte: visualizzazione ed editing. La transizione comporta modifiche dinamiche dell'interfaccia con validazione in tempo reale.

Durante il salvataggio, il *NotesViewModel* utilizza coroutines per eseguire operazioni database in background, mantenendo l'interfaccia responsiva.

La cancellazione multipla coinvolge *SelectionManager* per identificare elementi selezionati, *SelectionHelper* per dialoghi di conferma, e ViewModel per l'operazione effettiva. Dopo la cancellazione, tutti i componenti si sincronizzano automaticamente.

Il sistema gestisce anche stati vuoti con messaggi informativi personalizzati e fornisce feedback visivi durante le operazioni. Questa architettura garantisce che ogni componente abbia responsabilità ben definite, facilitando manutenzione ed estensione del sistema.



4. Sezione galleria di un viaggio

La sezione galleria fotografica del viaggio condivide la medesima architettura modulare degli adapter precedentemente descritta, ma presenta caratteristiche specializzate per la gestione di contenuti multimediali e layout complessi.

L'implementazione si basa sui seguenti elementi:

- *PhotoGalleryFragment*: gestisce la visualizzazione della galleria con layout a griglia
- *PhotoFullScreenFragment*: visualizzazione a schermo intero delle singole fotografie
- *PhotoGalleryViewModel*: logica di business specifica per foto e sincronizzazione con il sistema
- *PhotoAdapter*: adapter specializzato che estende direttamente BaseAdapter senza funzionalità di selezione integrate

Il *PhotoAdapter* presenta una particolarità architettonica significativa: invece di estendere *SelectableAdapter*, estende direttamente *BaseAdapter* e implementa la propria istanza di *SelectionManager*. Questa scelta progettuale è motivata dalla necessità di gestire due tipi di elementi distinti:

- *PhotoGalleryItem.DateHeader*: Intestazioni con date che occupano l'intera larghezza della griglia
- *PhotoGalleryItem.Photo*: Singole fotografie disposte in layout a griglia 3x3

La gestione della selezione multipla è implementata tramite composizione diretta del SelectionManager, consentendo un controllo accurato su quale tipo di elemento può essere selezionato. Solo le fotografie sono selezionabili, mentre le intestazioni di data rimangono elementi puramente informativi. Se invece il *PhotoAdapter* avesse implementato la gestione della selezione multipla tramite composizione di *SelectableAdapter*, sarebbero stati selezionabili tutti gli elementi.

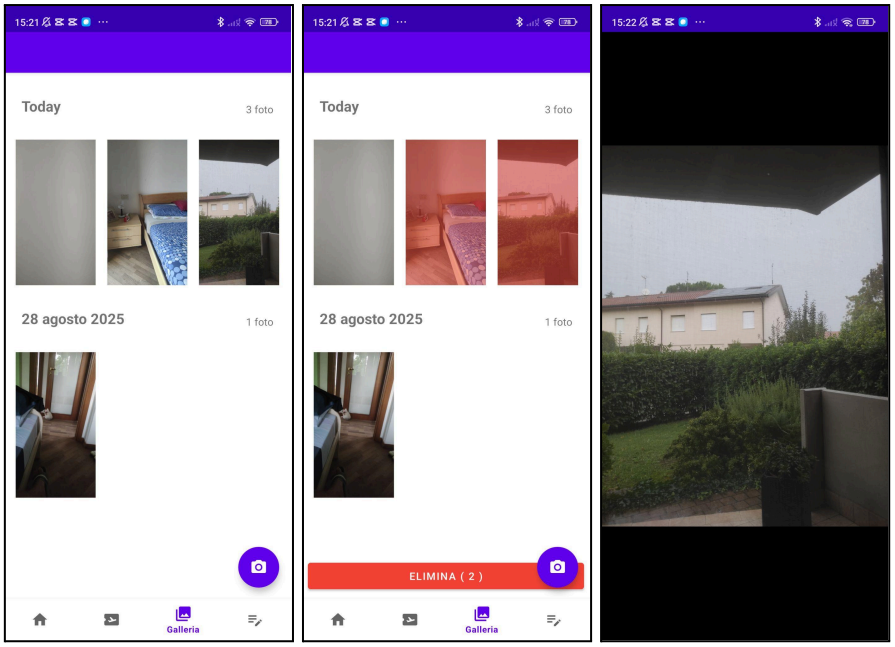
Il *PhotoGalleryViewModel* implementa una logica di raggruppamento temporale utilizzando la trasformazione *map* su LiveData. La funzione `groupPhotosByDate()` organizza le fotografie per data, creando automaticamente intestazioni quando necessario e ordinando cronologicamente i contenuti.
Inoltre è stata implementata la sincronizzazione bidirezionale con la galleria di sistema Android. Il ViewModel implementa `syncWithSystemGallery()` che verifica periodicamente l'accessibilità delle URI memorizzate nel database, rimuovendo automaticamente i riferimenti a file eliminati esternamente dall'applicazione.

Il fragment gestisce dinamicamente i permessi di camera e storage, utilizzando gli *Activity Result Contracts* moderni per richiedere autorizzazioni e lanciare l'intent della fotocamera. Le nuove fotografie vengono automaticamente salvate nella directory dedicata TravelCcompanion del MediaStore di sistema.

Il *PhotoFullScreenFragment* implementa la visualizzazione a schermo intero dell'immagine caricata tramite l'utilizzo di *Glide*.

Anche in questo adapter ci sono payload specializzati per aggiornamenti granulari che vengono gestiti nel metodo `handlePayloadUpdate()`. Quando cambia solo lo stato di selezione di una foto, il sistema utilizza payload come PAYLOAD_SELECTION_CHANGED per aggiornare esclusivamente l'overlay di selezione tramite `updateSelectionVisuals()`, evitando di ricaricare l'intera immagine.

Infine il lazy loading automatico è implementato attraverso Glide nel metodo `bind()` del PhotoViewHolder. La configurazione `.centerCrop().placeholder(R.drawable.ic_menu_gallery)` garantisce che le immagini vengano caricate progressivamente durante lo scroll, mostrando un placeholder fino al completamento. Per la visualizzazione fullscreen, il sistema utilizza inoltre `.diskCacheStrategy(DiskCacheStrategy.ALL)` nel metodo `loadPhoto()`, ottimizzando il caching su disco per accessi successivi.



5. **Sezione viaggi + filtri di visualizzazione**

La sezione lista viaggi dà la possibilità all'utente di visualizzare tutti i viaggi presenti sull'applicazione e applicare filtri di visualizzazione su di essi. Questa sezione dell'applicazione rappresenta l'implementazione più diretta dell'architettura *SelectableAdapter.*

L'implementazione si basa sui seguenti componenti principali:

- *TripsFragment*: gestisce la visualizzazione della lista viaggi con sistema di filtri avanzato
- *TripListAdapter*: estende *SelectableAdapter* seguendo il pattern architetturale standard
- *TripsViewModel*: logica di business per CRUD operations sui viaggi
- *FiltersViewModel*: gestione dedicata dei filtri di ricerca e data

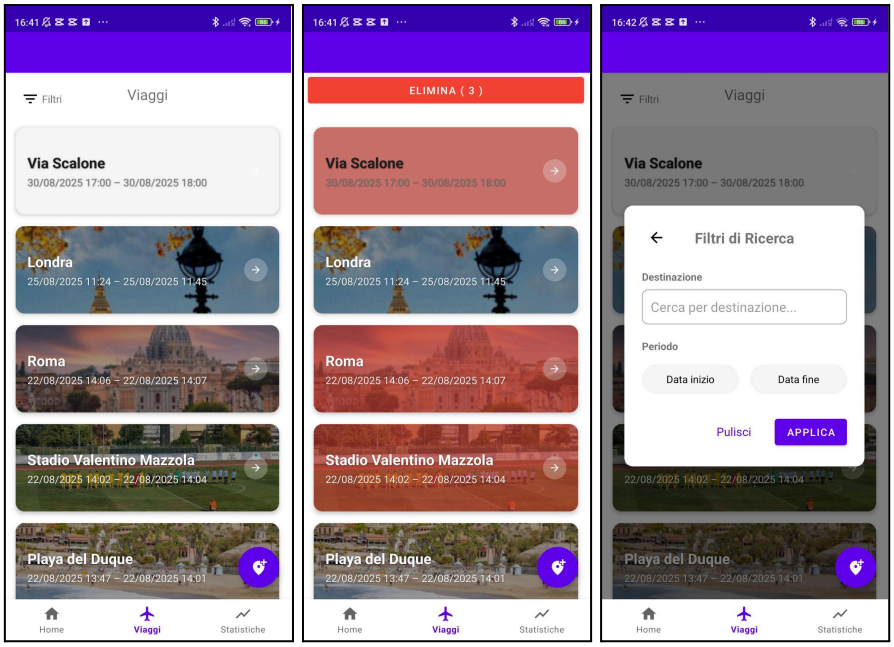
Il *TripListAdapter* rappresenta un'implementazione paradigmatica di *SelectableAdapter*, utilizzando tutte le funzionalità integrate senza necessità di customizzazioni.

Una caratteristica distintiva è il *FiltersViewModel* dedicato che implementa filtri multi-criterio:

- Filtro per destinazione: ricerca testuale con pattern matching
- Filtri temporali: selezione range di date tramite MaterialDatePicker

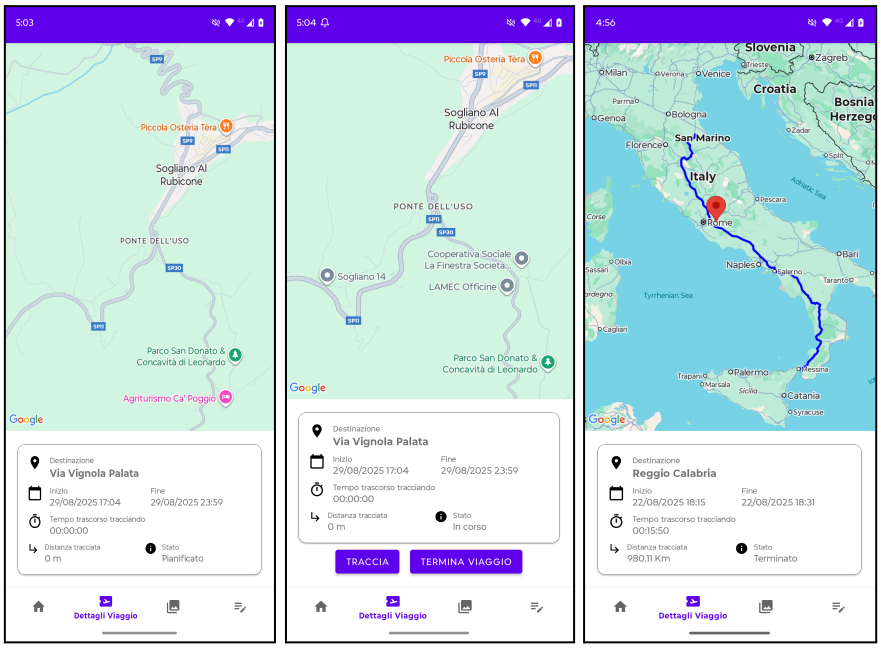
La logica di filtraggio utilizza il metodo `filterTrips()` che applica criteri combinati.

Una menzione speciale deve essere fatta per il *TripViewHolder,* che tramite il metodo `setupImageAndOverlay(trip: TripEntity),` gestisce dinamicamente la presenza/assenza di immagini associate ai viaggi, ottimizzando il layout e garantendo leggibilità del testo attraverso overlay e colori adattativi.



6. Dettagli Viaggio

La schermata di dettaglio di un viaggio si presenta come segue:



In particolare:

- Per la maggior parte di schermo è visualizzata la mappa del Google Map SDK che mostra il tracciamento del viaggio. Permette all'utente di interagire per:
  - creare punti di interesse (marker rosso, immagine 3), cliccando sulle attività già presenti sulla mappa oppure tenendo premuto su un qualsiasi punto
  - eliminare punti di interesse, cliccando su uno di essi e cliccando sul bottone "Elimina"

- sotto la mappa sono presenti i dettagli del viaggio e i bottoni per la gestione del tracciamento e la terminazione anticipata del viaggio. I bottoni sono visualizzati solo quando il viaggio è iniziato (immagine 2), mentre quando non è ancora iniziato (immagine 1) o è terminato (immagine 3) non verranno mostrati
- il bottone Traccia/Ferma permette all Fragment di interfacciarsi, tramite l'invio di intents specifici, con un *TrackingServic*e, il quale è responsabile:
    - del tracciamento**,** che consiste nel recupero delle coordinate usate dalla mappa per mostrare le linee del percorso tracciato e del cronometraggio del tempo di tracciamento.
    - del geofencing che, quando il servizio rileva l'entrata o l'uscita dal raggio (100m) dei luoghi di interesse preventivamente inseriti, invia un intent ad un BroadcastReceiver che lo processa e in base al tipo di evento ricevuto (entrata/uscita) invia una notifica all'utente

  Inoltre, ogni volta che si clicca su "Traccia", si aggiorna, assegnando il timestamp del click, una variabile nelle SharedPreferences per tenere conto dell'ultima volta che si è tracciato un viaggio. Tale variabile serve ad un *InactivityCheckWorker* che, ogni 7 giorni, controlla se siano o meno passati 60 giorni dall'ultima volta che si è tracciato un viaggio e, se lo sono, invia una notifica all'utente per ricordargli di tracciare i suoi viaggi.

- in fondo è presente una barra per navigare nei fragment Home, Note e Galleria del viaggio

7. Sezione statistiche

La sezione statistiche implementa visualizzazioni avanzate e analisi dei dati di viaggio attraverso interfacce interattive.

L'implementazione si basa sui seguenti elementi specializzati:

- *StatisticsFragment*: gestisce due modalità di visualizzazione (mappa heatmap e grafici statistici) con toggle dinamico
- *StatisticsViewModel*: logica di business per elaborazione dati statistici e coordinate geografiche
- Integrazione Google Maps: visualizzazione heatmap dei percorsi effettuati
- *MPAndroidChart*: libreria per grafici a barre interattivi

Il *StatisticsFragment* implementa un sistema di visualizzazione innovativo con due modalità principali gestite tramite enum *ViewType*.

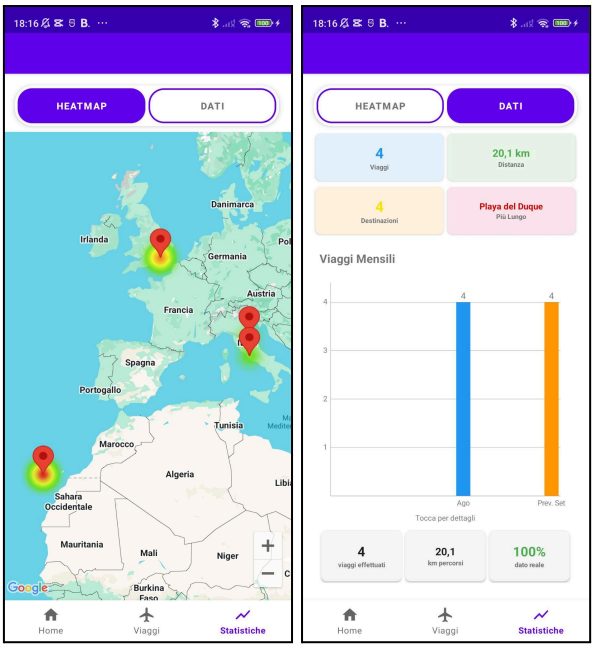
La modalità MAP utilizza Google Maps SDK per creare una heatmap termica dei percorsi effettuati. Il sistema recupera tutte le coordinate dei viaggi completati tramite `getAllCoordinatesForCompletedTrips()` e applica un campionamento intelligente con `sampleCoordinates()` per ottimizzare le performance con dataset di grandi dimensioni.

La modalità STATS presenta grafici a barre interattivi utilizzando *MPAndroidChart*, dove ogni barra rappresenta i viaggi mensili con visualizzazione dei dati storici.

Il *StatisticsViewModel* implementa una logica per la gestione delle coordinate geografiche. Il sistema filtra automaticamente i viaggi completati con coordinate valide tramite controlli di validazione sui valori di latitudine e longitudine. La creazione della heatmap utilizza *HeatmapTileProvider* con configurazioni personalizzate.
Il metodo `centerMapOnCoordinates()` implementa un algoritmo di centratura intelligente che calcola automaticamente i bounds ottimali utilizzando `LatLngBounds.builder()` per visualizzare tutti i dati con padding appropriato.

L'implementazione di *MPAndroidChart* si basa sull'interazione grafico/utente. Il listener `OnChartValueSelectedListener` e il metodo `updatePredictionCardsForMonth()` consentono all'utente di visualizzare il numero di viaggi e i km percorsi nel mese selezionato. Quando si clicca un altro mese sul grafico, i valori vengono aggiornati automaticamente.

Il sistema inoltre calcola, a livello generale, le seguenti metriche viaggi completati, distanza percorsa, numero di destinazioni e viaggio più lungo tramite il metodo `updateStatisticsCards()`



8. **Modello di previsione** (funzionalità gruppi da 2)

La funzionalità di predizioni rappresenta uno degli aspetti più innovativi dell'applicazione, integrando algoritmi per analizzare i pattern di viaggio degli utenti e generare sia previsioni statistiche che suggerimenti personalizzati per viaggi futuri.

L'implementazione si articola attraverso diversi livelli specializzati:

- *PredictionViewModel*: coordina il caricamento e l'osservazione delle predizioni e suggerimenti

- *HomeViewModel*: gestisce l'integrazione dei suggerimenti nella schermata principale
- *TripPredictionAlgorithm*: algoritmo di analisi predittiva basato su dati storici
- *TripSuggestionsEngine*: motore di generazione suggerimenti contestuali
- *SuggestionsAdapter*: visualizzazione dei suggerimenti nella home screen

Il *TripPredictionAlgorithm* implementa un sistema di analisi sofisticato che elabora i dati storici per generare predizioni accurate. L'algoritmo analizza i viaggi completati negli ultimi 4 mesi (`MONTHS_TO_ANALYZE`) e richiede un minimo di 3 viaggi (`MIN_TRIPS_FOR_PREDICTION`) per garantire affidabilità statistica.

Il processo di predizione si articola in diverse fasi:

- Analisi dei trend temporali: il sistema raggruppa i viaggi per mese e calcola trend attraverso `calculateTrend()`, confrontando l'attività recente con quella passata per determinare se l'utente sta aumentando, diminuendo o mantenendo stabile la propria frequenza di viaggio.
- Calcolo metriche aggregate: bengono calcolate medie per numero di viaggi mensili e distanza percorsa, applicando moltiplicatori basati sui trend rilevati (1.2 per trend crescenti, 0.8 per decrescenti, 1.0 per stabili).
- Valutazione confidenza: il metodo `calculateConfidence()` determina l'affidabilità della predizione basandosi su due fattori: quantità di dati disponibili e consistenza temporale, restituendo un valore tra 0.0 e 1.0.

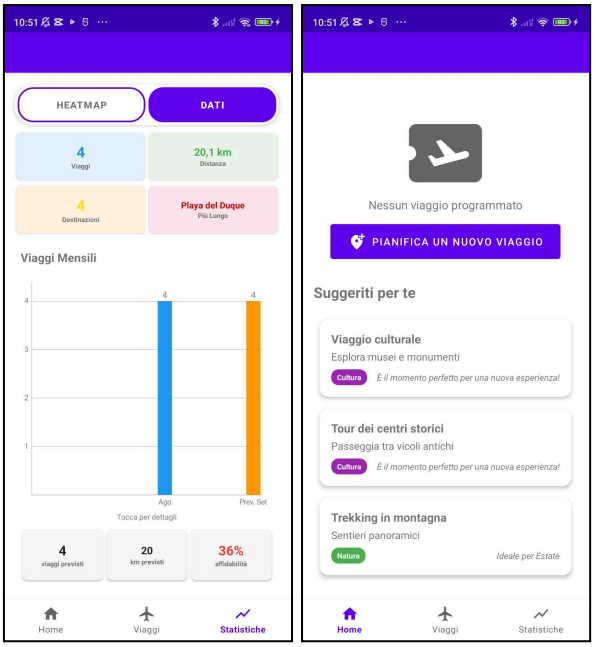
Il *TripSuggestionsEngine* utilizza un approccio multi-criterio per generare suggerimenti personalizzati attraverso template predefiniti categorizzati per tipo di esperienza (Cultura, Natura, Mare, Gastronomia, Romantico, Business, Relax, Avventura).

L'algoritmo di generazione combina tre strategie principali:

- Suggerimenti motivazionali: attivati quando l'utente ha pochi viaggi pianificati o presenta trend decrescenti, con priorità HIGH per stimolare l'attività di viaggio.
- Suggerimenti personalizzati: basati sull'analisi delle preferenze utente tramite `analyzeUserPreferences()`, che identifica i tipi di viaggio più frequenti e le destinazioni già visitate.
- Suggerimenti stagionali: generati attraverso `generateSeasonalSuggestions()` che propone attività appropriate alla stagione corrente (es. mare in estate, cultura in inverno).

Il *PredictionViewModel* si integra con il sistema statistico esistente. Le predizioni vengono inserite nel grafico a barre come tredicesima colonna (indice 12) con colore differenziato (arancione vs blu per dati storici).

Il *HomeViewModel* orchestra la presentazione dei suggerimenti nella schermata principale, limitando la visualizzazione a 3 suggerimenti (`MAX_HOME_SUGGESTIONS`) per evitare sovraccarico informativo. Essi sono gestiti dal *SuggestionsAdapter.*
I suggerimenti vengono aggiornati automaticamente grazie al metodo `observePredictionChanges(),` il quale monitora il cambiamento dei viaggi sul db.


10:51
HEATMAP
DATI
4
Viaggi
20,1 km
Distanza
4
Destinazioni
Playa del Duque
Più Lungo
Viaggi Mensili
4
4
4
3
2
1
Ago
Prev. Set
Tocca per dettagli
4
viaggi previsti
20
km previsti
36%
affidabilità
Home
Viaggi
Statistiche
10:51
Nessun viaggio programmato
PIANIFICA UN NUOVO VIAGGIO
Suggeriti per te
Viaggio culturale
Esplora musei e monumenti
Cultura
È il momento perfetto per una nuova esperienza!
Tour dei centri storici
Passeggia tra vicoli antichi
Cultura
È il momento perfetto per una nuova esperienza!
Trekking in montagna
Sentieri panoramici
Natura
Ideale per Estate
Home
Viaggi
Statistiche