Relazione finale - Game Vault

Componente	*Email*
Matteo Calvanico	matteo.calvanico@studio.unibo.it
Filippo Monti	filippo.monti15@studio.unibo

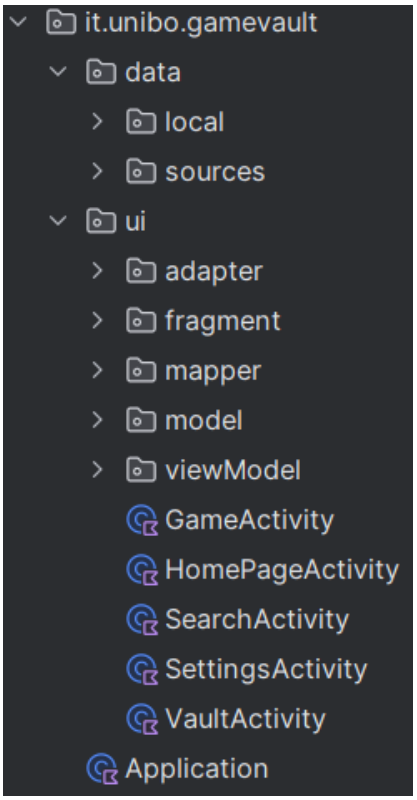
Scopo

Lo scopo del progetto era la creazione di un'app per tracciare tutti i giochi nella nostra collezione, dandogli un voto e salvando le date di inizio e di completamento.

L'idea è nata perché non abbiamo mai trovato un'app che abbia le stesse caratteristiche di Letterboxd ma per i videogiochi. Per questo abbiamo deciso di unire l'utile al dilettevole usando il progetto soprattutto per un uso personale.

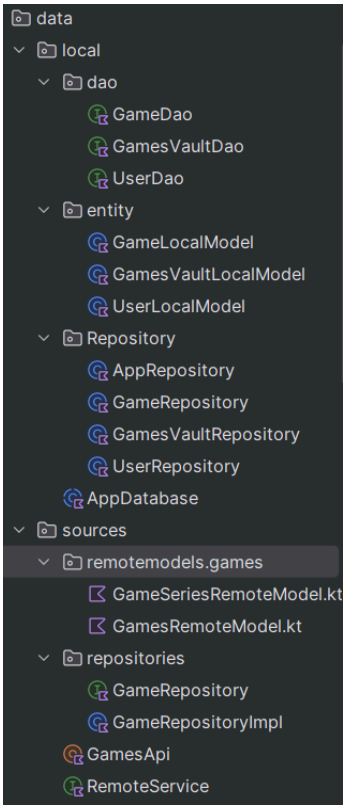
Struttura

Di seguito mostriamo come abbiamo deciso di suddividere il progetto, seguendo il modello MVVM e la clean architecture:



Di seguito, mostriamo nel dettaglio ogni sotto cartella con la spiegazione delle decisioni prese:

Data



Local

Contiene tutte le classi e le interfacce necessarie per l'interazione con il database locale. Suddiviso in tre sottocartelle: ‘`dao`’, ‘`entity`’ e ‘`repository`’.

Dao

Il pacchetto ‘`dao`’ (Data Access Object) contiene le interfacce che definiscono i metodi per l'interazione con le tabelle del database. Questi metodi includono operazioni come inserimento, aggiornamento, cancellazione e recupero dati. Room genera automaticamente l'implementazione di queste interfacce.

GameDao

Interfaccia che definisce le operazioni di accesso ai dati per la tabella ‘game’. Include metodi per ottenere tutti i giochi, recuperare un gioco tramite il suo ‘slug’, inserire, aggiornare e eliminare un gioco. Inoltre è possibile eliminare tutti i giochi e ottenere tutti i giochi sotto forma di un flusso reattivo (‘Flow’).

GamesVaultDao

interfaccia che definisce le operazioni di accesso ai dati per la tabella ‘games_vault’. Include metodi per ottenere tutti i giochi del vault, inserire ed eliminare un gioco del vault, e eliminare tutti i giochi del vault sotto forma di un flusso reattivo (‘Flow’).

UserDao

interfaccia che definisce le operazioni di accesso ai dati per la tabella ‘user’. Include metodi per ottenere tutti gli utenti, ottenere un utente per id, inserire, aggiornare e eliminare un utente, e eliminare tutti gli utenti sotto forma di un flusso reattivo (‘Flow’).

Entity

Il pacchetto `entity` contiene le classi di dati che rappresentano le tabelle del database. Queste classi sono annotate con le annotazioni di Room per definire la struttura delle tabelle e le relazioni tra di esse.

GameLocalModel

Classe di dati che rappresenta un'entità del database locale nella tabella ‘game’, che memorizza le informazioni di un gioco.

GamesVaultLocalModel

Classe di dati che rappresenta un'entità del database locale nella tabella ‘games_vault’, che memorizza i giochi presenti nel vault dell'utente

UserLocalModel

Classe di dati che rappresenta un'entità del database locale nella tabella ‘user’. La tabella ‘user’ ha chiavi esterne che si riferiscono alla tabella ‘game’, associando quattro giochi preferiti ai dettagli completi dei giochi nella tabella ‘game’. Include informazioni sull'utente, come l'immagine del profilo, i link ai profili PSN, Steam e Xbox, una chiave API e i giochi preferiti.

Repository

Il pacchetto repository contiene le classi che forniscono un'interfaccia di accesso ai dati per il resto dell'applicazione. Il repository gestisce le operazioni di recupero e modifica dei dati, utilizzando i DAO per interagire con il database.

AppRepository

classe che fornisce un'interfaccia di accesso ai dati per il resto dell'applicazione. Fornisce metodi per inserire, aggiornare, eliminare e ottenere utenti, giochi e giochi del vault, utilizzando coroutine per operazioni di threading sicuro, inoltre viene utilizzato ‘Flow’ per

esporre flussi di dati reattivi. È stato realizzato un metodo per ottenere una combinazione di dati dai giochi del vault con dettagli completi.

GameRepository

classe che gestisce l'accesso ai dati per i giochi attraverso ‘GameDao’. Fornisce metodi per inserire, aggiornare, eliminare e ottenere giochi per ‘slug’, utilizzando coroutine per operazioni di threading sicuro, viene utilizzato ’Flow’ per esporre flussi di dati reattivi.

GameVaultRepository

classe che gestisce l'accesso ai dati per i giochi del vault attraverso ‘GamesVaultDao’. Fornisce metodi per inserire ed eliminare giochi del vault, utilizzando coroutine per operazioni di threading sicuro, viene utilizzato ’Flow’ per esporre flussi di dati reattivi.

UserRepository

classe che gestisce l'accesso ai dati per gli utenti attraverso ‘UserDao’. Fornisce metodi per inserire, aggiornare, eliminare e ottenere utenti per ‘id’, utilizzando coroutine per operazioni di threading sicuro, viene utilizzato ’Flow’ per esporre flussi di dati reattivi.

Resto dei file

AppDatabase

Il progetto utilizza Room per la gestione del database locale, suddividendo le responsabilità tra le entità (definizione delle tabelle), i DAO (definizione delle operazioni sul database) e i repository (gestione dell'accesso ai dati). Questa architettura permette una gestione modulare e scalabile dei dati all'interno dell'applicazione, garantendo al contempo un'interfaccia pulita e un accesso efficiente ai dati per il resto dell'applicazione.

Sources

Questa è la parte delle API, qui sono contenuti tutti i file per gestire le chiamate e i modelli dei json di ritorno. L’utilizzo della libreria Moshi e Retrofit è stato fondamentale per la gestione di questi ultimi.

RemoteModels

GameSeriesRemoteModel

In questo file viene rappresentato il modello di ritorno dati dell’API getGameSeries, in alcuni campi usiamo la dicitura @Json per indicare il nome che avrà quel campo nel Json di ritorno in modo da usare una sintassi giusta per i nomi dei campi della data class in Kotlin.

GamesRemoteModel

Come prima ma serve come ritorno per getGame, sfortunatamente questa data class non è uguale al gioco singolo all'interno della lista della classe precedente, infatti questo ha portato alcuni problemi, con la svista della differenza abbiamo gestito i due ritorni in maniera simile all'inizio, usando questa data class all'interno della lista della classe precedente.

Repositories

GameRepository

Interfaccia che viene implementata dalla classe successiva, contiene due metodi che verranno chiamati nella ricerca per restituire il gioco o la lista di giochi.

GameRepositoryImpl

Implementazione dei metodi precedenti, oltre alla chiamata all'API gestione la traduzione da modello utilizzato dall'API a modello utilizzato nella parte UI; per farlo usiamo un Mapper spiegato dopo.

Resto dei file

GamesAPI

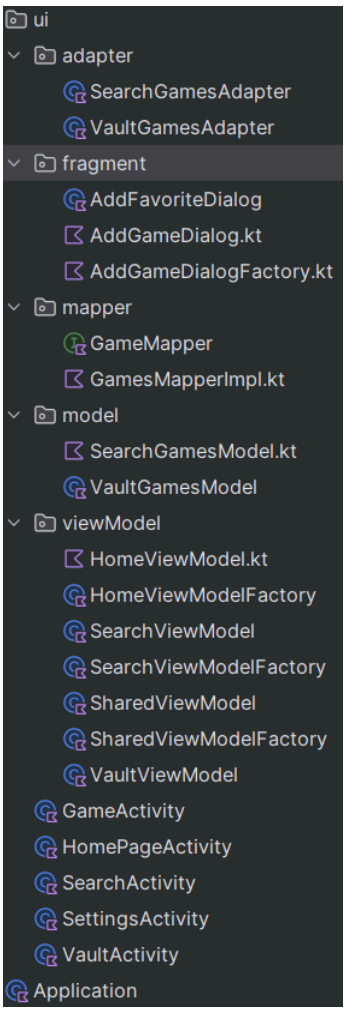
Oggetto dove inseriamo tutte le informazioni per il funzionamento dell'API come il base URL (la radice sempre uguale delle API) e la chiave, che dovrebbe essere presa dal database e inserita dall'utente ma abbiamo lasciato così per facilitare lo sviluppo e la dimostrazione.
Qui avviene anche la creazione di Moshi e Retrofit le due librerie per far funzionare le API, la prima per gestire i Json e la seconda per le chiamate effettive.

RemoteService

Qui ci sono i due metodi per ottenere i giochi, con la dicitura @GET indichiamo che quella chiamata vogliamo prendere qualcosa, i campi passati sono la chiave e lo slug del gioco o serie.
Si è scelto di usare la dicitura @Path per andare a dire esattamente dove mettere lo slug del titolo.

UI

In questa parte sono contenuti tutti i file che vanno a lavorare direttamente sui layout e con cui l'utente interagisce, alcune activity usano il viewModel per una divisione dei compiti, persistenza e per seguire il pattern MVVM. Sono anche presenti adapter per le activity con recycler, model per rappresentare i dati, i vari fragment che vengono usati nelle activity e il mapper citato precedentemente.



Adapter

SearchGamesAdapter

Adapter della Activity di ricerca dei giochi, si occupa di aggiornare la recycler con i dati che prende dal dataSet aggiornato dal ViewModel.
Da qui parte anche l'activity con i dettagli del gioco, infatti rimaniamo in ascolto di eventuali click sui poster dei giochi appena cercati con successivo passaggio alla GameActivity.

VaultGamesAdapter

Adapter della Activity del vault, si occupa di aggiornare la view con tutti i giochi aggiunti che prende dal dataSet aggiornato dal ViewModel.
Qui c'è la gestione dello slide verso destra che permette di eliminare un gioco da vault, per farlo utilizziamo `ItemTouchHelper`, classe di utilità che permette lo swipe, il drag and drop, ecc negli item di un RecyclerView.

Fragment

AddFavoriteDialog

Classe che estende BottomSheetDialogFragment, creata apposta per mostrare il classico Dialog che esce dal basso e che non copre tutto lo schermo, da qui gestiamo l'effettiva aggiunta dei giochi preferiti, con la chiamata al ViewModel condiviso per la ricerca nel database e l'observer per aggiornare il poster.

AddGameDialog

Classe che estende BottomSheetDialogFragment, come la precedente, da qui gestiamo l'effettiva aggiunta/update dei giochi con la possibilità di inserire il punteggio e le due date.
Se il gioco è già nel Vault allora mostriamo i vecchi dati.
Per mostrare le date e salvarle correttamente usiamo un piccolo listener per aggiornare correttamente le date nella view.

AddGameDialogFactory

Factory per la creazione del dialog con passaggi del gioco.

Mapper

GameMapper

Interfaccia che verrà implementata dal mapper, definisce due metodo, uno per trasformare il gioco che arriva dalle API nel model usato dalla UI, l'altro fa la stessa cosa ma ogni gioco lo mette in una lista.

GameMapperImpl

Implementazione dell'interfaccia precedente, trasforma ogni gioco passato in un'istanza della classe Game, il model che viene usato nella UI

Model

Tutti i model sono Parcelize, per garantire un passaggio corretto fra Activity o Fragment

SearchGamesModel

Model che rappresenta come i giochi ottenuti dalla ricerca sono fatti, salviamo solo le informazioni che ci servono ignorando tutti gli altri parametri del json.
La classe ha anche un metodo per formattare la descrizione in modo appropriato, andando a rimuovere tutte i caratteri che l'API inserisce nella descrizione di ogni gioco.

VaultGamesModel

Model che rappresenta come i giochi salvati nel Vault sono fatti.

viewModel

HomeViewModel

ViewModel della home page, qui vengono salvati in dei LiveData le informazioni dell'utente e dei giochi preferiti. Tramite la funzione loadUserData, che viene eseguita alla creazione, prendiamo i dati dal database e li carichiamo nei LiveData corrispondenti per essere osservati nell'Activity principale.

HomeViewModelFactory

Factory necessario per la creazione del ViewModel precedente con il passaggio del repository.

SearchViewModel

ViewModel dell'Activity di ricerca, qui viene svolta l'effettiva ricerca tramite API, e salviamo nel LiveData i dati ottenuti. Ci sono anche LiveData secondari che gestiscono gli errori e il caricamento, sarà infatti tramite quelli che gli comunichiamo se il caricamento è finito o meno e se sono avvenuti errori.

SearchViewModelFactory

Factory necessario per la creazione del ViewModel precedente con il passaggio del repository per le API.

SharedViewModel

ViewModel condiviso tra la Dialog di aggiunta preferiti e la SettingsActivity. Sarà il ViewModel che, preso il nome del gioco scelto dall'utente nel dialog descritto in precedenza e dopo aver fatto la ricerca, inserirà il risultato nel LiveData il quale informerà l'Activity dei settings.
Anche qui sono presenti dei LiveData secondari per errori e progress bar come nel ViewModel precedente.

SharedViewModelFactory

Factory necessario per la creazione del ViewModel precedente con il passaggio del repository per il database.

VaultViewModel

ViewModel per la gestione del Vault, con le funzioni per cancellare i giochi dal database allo swipe e per l'aggiunta (non utilizzata), ovviamente è presente il LiveData che tiene i giochi.

Resto dei file

GameActivity

Activity del gioco, dove viene modificato il layout con tutte le informazioni del gioco stesso, aperto dopo il click del poster nella sezione ricerca che gli passa tramite intent l'istanza Game contenente le info prese tramite API.

HomePageActivity

Activity principale e la prima ad essere creata, qui si carica il layout della home, si prendono i dati dell'utente se è presente, altrimenti tramite Toast lo si invita ad andare nei settaggi e creare il suo profilo, e si caricano nelle parti di layout corrispondente. Da qui si può andare al Vault tramite tasto in fondo o nella ricerca al click della lente.
Al click dei tre loghi si aprono i link che portano ai profili degli utenti su altre piattaforme.

SearchActivity

Activity della ricerca dei giochi, qui si osservano i dati presenti nei LiveData, caricati dal ViewModel precedentemente spiegato, e si passano al adapter o per l'errore e la progress bar si mostra un Toast o si nasconde/mostra la barra.
Da qui parte anche la chiamata andando a usare il metodo offerto dal ViewModel.

SettingsActivity

Activity più complessa e lunga, dove si prendono e salvano le configurazioni dell'utente, come preferiti, link degli account e foto profilo.
All'inizio si caricano i dati dell'utente, tramite il metodo rispettivo, tramite chiamata al database.
Qui c'è la chiamata del dialog per l'aggiunta dei preferiti e l'observer per il gioco preferito, aggiunto per aggiornare il database in seguito e l'immagine aggiornarla immediatamente.
La parte finale del file comprende tutti i metodi per i permessi necessari per aprire la galleria e far scegliere all'utente l'immagine.

VaultActivity

Activity del Vault dove si gestisce la visualizzazione del testo in caso di vault vuoto, chiamata al adapter con slide per cancellare. Qui viene gestita la search bar per filtrare i giochi in base al nome, con successivo update del LiveData per l'adapter.

Application

File che viene chiamato all'inizio dove c'è la creazione del database e delle repository, abbiamo aggiunto anche quelle specifiche ma sono ridondanti visto che usiamo solo quella generale.

Punti di forza

La suddivisione del codice in package è stata fatta in maniera ottimale, seguendo le linee guida della clean architecture.

Abbiamo diviso i vari valori utilizzati nella parte di layout nei giusti file (dimens, string, ecc).

L'utilizzo di design pattern, librerie e componenti di terze parti ci hanno migliorato sul punto di vista della ricerca e comprensione della documentazione alleggerendo anche certi carichi di lavoro evitando di "ricreare la ruota da zero".

Migliorie

Gestione più accurata di alcuni ViewModel, per esempio quello della Home, infatti la HomeActivity non è persistente e al cambio di configurazione i dati vengono persi.

Un altro problema è il Settings, troppo grande con troppe linee di codice; si potrebbe optare per un altro suo personale ViewModel con l'inserimento dei metodi e dei permessi in un file separato, andandoli ad astrarre in minima parte e rendendoli utilizzabili in altre parti del progetto.

Alcuni bug non sono stati risolti come l'errore quando si cerca di inserire immagini troppo grandi della foto profilo con successivo crash dell'Activity.

Come si può notare dal branch 'feature/map' di github, si è tentata, ma con pochi risultati, la realizzazione di un 'Google Maps' il cui scopo è quello di fornire all'utente, la posizione dei negozi di videogiochi nei dintorni, grazie alla geolocalizzazione del proprio smartphone.