Report progetto laboratorio di applicazioni mobili

Filippo Perrina

Giugno 2021

1 Dati personali

Nome: Filippo
Cognome: Perrina
Email: filippo.perrina@studio.unibo.it
Matricola: 0000874462

2 Grafo di navigazione

L'applicazione è composta da 8 activities e 3 fragments. Il seguente grafo di navigazione mostra le possibili direzioni che l'utente può seguire durante l'uso dell'applicazione.

I rettangoli verdi rappresentano le activites, i rettangoli rossi i fragments. Le frecce in nero rappresentano transizioni che vengono attivate dall'utente, solitamente cliccando un bottone, su queste transizioni se l'utente preme il tasto indietro (onBackPressed()) il componente viene semplicemente rimosso dal backstack. Al contrario, la freccia in rosso rappresenta una transizione che va solo in un verso, prima di passare da LoginActivity a MainActivity il backstack viene svuotato, quindi non è possibile tornare indietro.
Per nessuna di queste transizioni è stato usato il componente Navigation, anche

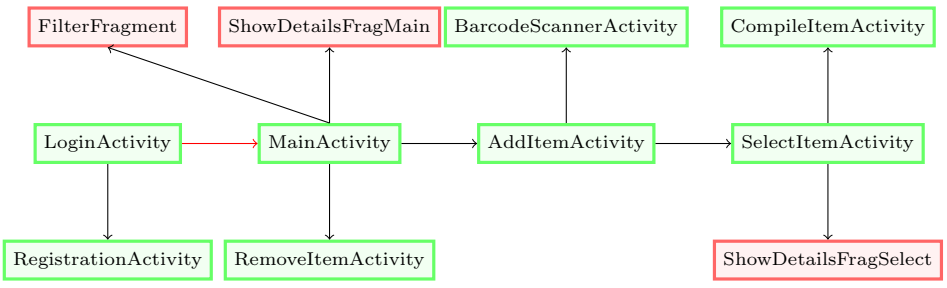

Figure 1: Grafo di navigazione dell'applicazione

se la struttura a grafo potrebbe suggerirlo. Ho ritenuto l'uso del componente non necessario, poichè non ho bisogno della struttura fissata che Navigation offre (menu,drawer...), infatti ogni componente dell'applicazione ha una struttura diversa. Inoltre, volevo gestire il backstack in modo differente a seconda del contesto applicativo in cui ero, ho quindi preferito non usare Navigation per avere più libertà, anche se riconosco che è un approccio più *error-prone*.

3 Modello ModelViewViewModel

L'intera applicazione è sviluppata seguendo il modello ModelViewViewModel (MVVM), con un ulteriore livello di astrazione fornito dalla repository. Quest'ultima contiene solo chiamate al database locale, che è la *single source of truth* che uso per popolare la dispensa. Le chiamate di rete non sono presenti nella repository, poichè reputo i dati contenuti nel server e quelli nel database interno appartenenti a due aree diverse, sarebbe ridondante in questa applicazione effettuare le chiamate di rete e con i risultati di queste andare a popolare il nostro database. Tuttavia, la repository è stata sviluppata per garantire la struttura nel caso di futuri aggiornamenti. Di seguito la struttura dell'applicazione.

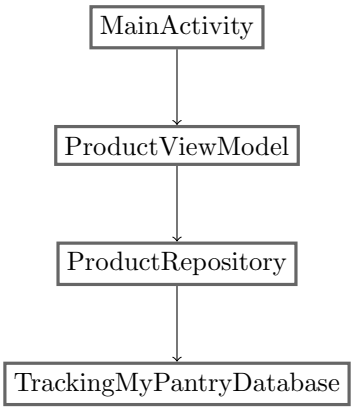

Figure 2: Modello ModelViewViewModel

Dalla figura si può vedere che l'applicazione ha solo una classe ViewModel, l'unico componente ad avere un riferimento a quest'ultimo è la MainActivity.

3.1 RecyclerViewListProducts

La recyclerViewListProducts è una recyclerView che mostra gli elementi che sono presenti nella dispensa locale. E' contenuta nella MainActivity ed è l'unico elemento nell'intera applicazione che viene controllato dal ViewModel. Questa

recyclerView è gestita da un ListAdapter, chiamato ProductAdapter. Ho scelto di usare un ListAdapter poichè i prodotti nella dispensa sono una List, in particolare una *List<ProductEntity>*, e questo adapter mi permette con semplicità di confrontare due liste, mostrando quella più aggiornata.
I prodotti mostrati nella recyclerView sono presi dal database locale, attraverso la chiamata getListOfWords() che ritorna la lista dei prodotti presenti nella dispensa, infatti ritorna una LiveData<List<ProductEntity>>. Ho scelto di utilizzare un LiveData, perchè mi permette di osservare i dati nella dispensa e di reagire attraverso una callback ad ogni cambiamento (insert,delete...) che avviene sulla lista.
Alla recyclerView è associato anche un ViewHolder che mantiene i riferimenti alle view interne, una TextView e un Button. Al bottone è associato un onClickListener, sviluppato con un'interfaccia, che apre il fragmentShowDetailsMain, componente che mostra i dettagli del prodotto specifico. Le classi che gestiscono la recyclerView:

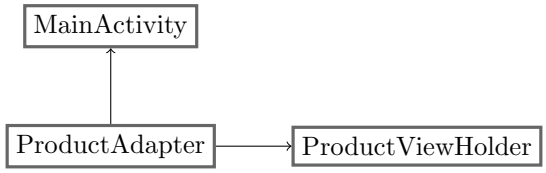

Figure 3: Struttura della recyclerViewListProducts

4 Database locale

Per la gestione del database interno ho utilizzato la libreria Room, poichè mi fornisce un'astrazione oltre SQLite e se usato insieme ai LiveData esegue automaticamente le chiamate su un thread esterno. Il database viene istanziato nella classe TrackingMyPantryDatabase, nella quale viene anche istanziato un pool di thread che mi permette di eseguire in thread esterni le operazioni sul database non riguardanti LiveData.

4.1 Dao Access Object

La classe ProductDAO è un interfaccia Data Access Objects che esegue le operazioni CRUD sul database, in particolare esegue:

- insert(ProductEntity product): inserisce la ProductEntityspecificata.
- remove(ProductEntity product): rimuove la ProductEntity specificata e ritorna quante ProductEntity ha rimosso.

- deleteAll(): rimuove tutte le ProductEntity dal database, chiamata solo all'installazione dell'applicazione.
- addCategory(String id, String category): imposta la categoria del prodotto con idProduct=id.
- addEndDate(String id, String endDate): imposta la data di scadenza del prodotto con idProduct=id.
- addQuantity(String id, String quantity): imposta la quantità del prodotto con idProduct=id.
- getListOfProducts(): ritorna i prodotti presenti nel database.

4.2 Entity

ProductEntity è la sola tabella che fa parte del mio database, contiene quattro dati presi dal server remoto: id, product (nome del prodotto), codice a barre e descrizione, e tre dati impostati dall'utente a suo piacere: la categoria a cui il prodotto appartiene, la data di scadenza e la quantità posseduta del prodotto. La chiave primaria è il nome, poichè su questa colonna impedisco che ci siano duplicati, sono tutti valori unici. Tutti i dati sono memorizzati come String, tranne la quantità che è un intero. Lo schema della tabella è il seguente:

PRODUCT	id	barcode	description	category	endDate	quantity

Figure 4: Schema del database

5 Analisi componenti applicazione

In questa sezione descrivo ogni componente nel dettaglio, spiegandone lo scopo, le proprietà e anche l'implementazione nei punti meno intuitivi. Ognuno dei componenti con un lifecycle ha associato un layout, l'associazione è: NomeComponenteTipoComponente.java –>tipoComponente_nomeComponente.xml (e.g. MainActivity.java –>activity_main.xml). Per tutte le chiamate di rete nominate ho utilizzato la libreria Volley, perchè mi garantisce caching, l'esecuzione su thread esterni e la possibilità di inserire header e corpo della richiesta HTTP. Il permesso per INTERNET è dichiarato nel Manifest, inoltre per ogni chiamata di rete il permesso viene controllato a run-time.

5.1 LoginActivity

La LoginActivity è la schermata iniziale in cui l'utente si trova quando apre l'applicazione, chiede all'utente di inserire i dati (email e password) con cui si è registrato e sull'onClick del Button *Entra* esegue la operazione LOGIN.

Nel dettaglio controlla se il permesso per INTERNET è concesso e in caso affermativo chiama la funzione LoginRequest(), la quale crea l'oggetto JSON contenente email e password dell'utente e lo invia, attraverso una JsonObjectRequest, al server remoto. In caso di risultato positivo, lancio un Intent alla MainActivity, passandogli l'accessToken, e rimuovo LoginActivity dal backstack, questo passaggio è necessario per evitare che l'utente per uscire definitivamente dall'applicazione debba ripassare per questa activity. In entrambi i casi, positivo e negativo, l'utente è notificato con un Toast.
Infine, se l'utente non è ancora registrato, tramite l'onClick sul Button *Registrati* lancia l'activity RegistrationActivity e rimane in ascolto del risultato del risultato della activity tramite un ActivityResultLauncher<Intent>. Quando la registrazione va a buon fine, rimuovo LoginActivity dal backstack per evitare di avere due LoginActivity contemporaneamente sulla pila.

Figure 5: Schermata LoginActivity

5.2 RegistrationActivity

La RegistrationActivity gestisce l'operazione REGISTER, cioè registra l'utente per la prima volta al server. L'operazione viene realizzata dalla funzione registrationRequest() che invia sotto forma di oggetto JSON i dati (email, password e nome utente) al server. Quando il risultato è positivo, notifico la LoginActivity che la registrazione è avvenuta con successo e con un Intent ne rilancio un'altra istanza. Questo passaggio è necessario poichè è l'operazione LOGIN che mi ritorna l'accessToken, necessario per effettuare tutte le successive operazioni di rete, quindi ripassando per la LoginActivity l'utente quando accede ottiene l'accessToken ed è pronto ad utilizzare l'applicazione senza intoppi.

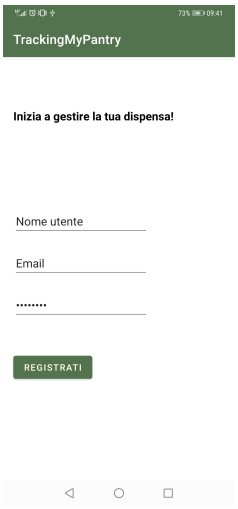

Figure 6: Schermata RegistrationActivity

5.3 MainActivity

La MainActivity come suggerisce il nome è la schermata principale dell'applicazione. Mostra all'utente gli elementi che ha nella dispensa, come già spiegato nella sezione sulla recyclerViewListProducts. I prodotti nella recyclerView vengono inseriti nello stesso modo ma con due tipologie di percorsi diversi, infatti la MainActivity reagisce con un ActivityResultLauncher<Intent>, sia all'operazione di POST PRODUCT REFERENCE sia ad una POST PRODUCT DETAILS. In entrambi i casi, prende i dati che gli vengono passati (in un Bundle) dalla AddItemActitvity e chiama la insertProduct del viewModel. Inoltre, resta in attesa anche di un terzo evento, la rimozione del prodotto dalla dispensa e in quel caso chiama la removeProduct del viewModel.
Questo componente ha anche associato un menù con due possibilità, la prima è la possibilità di filtrare la lista dei prodotti con una ricerca per sequenza di caratteri, la seconda apre un fragment che mostra i possibili filtri all'utente. Per implementare la prima ho dovuto creare una configurazione Searchable (res/xml/searchable.xml) ed associarla nel Manifest alla MainActivity. Successivamente, nella activity ho incluso una searchView e sulla callback per ogni carattere inserito su questa view, ho eseguito il filtraggio. Quest'activity realizza chiamando metodi del ProductAdapter tutti i meccanismi di filtraggio, che spiegherò nel dettaglio nella sezione successiva.

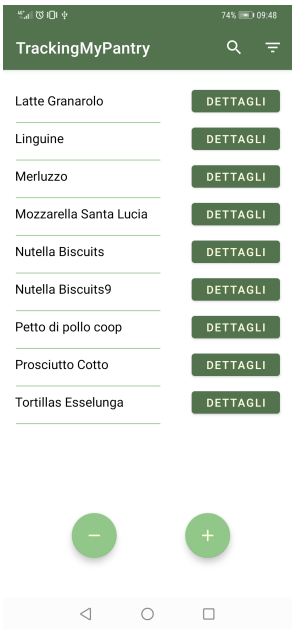

Figure 7: Schermata MainActivity

5.3.1 Filtraggio e fragmentFilter

Il FilterFragment mostra all'utente i possibili filtri che vengono applicati sulla dispensa: categoria, data di scadenza e quantità. Possono essere applicati singolarmente o a gruppi, la scelta è lasciata all'utente. I filtri così selezionati vengono restituiti alla MainActivity attraverso un FragmentResultListener, che riceve i risultati e li passa come parametri della funzione handleFilter, che si occupa realmente del filtraggio.
Per prima cosa, converte la selezione dell'utente in una data effettiva con cui fare il confronto (tramite la funzione convertDate). Successivamente controlla se il productAdapter ha la lista più aggiornata (attraverso un booleano), questo passaggio è necessario per evitare che il filtraggio avvenga su una lista non coerente con quella effettiva presente nel database. Infine, controlla i parametri inseriti dall'utente nel filterFragment, evitando che siano nulli e chiamando di conseguenza la giusta funzione di filtraggio contenuta nell'adapter.
Le funzioni di filtraggio nell'adapter confrontano due liste di ProductEntity, la prima è quella aggiornata (mList), la seconda è una lista vuota che viene riempita con le varie ProductEntity che rispettano i filtri. Questa seconda lista (listFiltered) viene restituita alla MainActivity che la mostra all'utente tramite una submitList, come ultimo passaggio il booleano che controlla se la lista mostrata è quella più aggiornata viene impostato a false, per evitare che il filtraggio venga effettuato su una lista parziale. Questo booleano verrà reimpostato a true in

due casi, quando si attiva l'observer della lista, segnalandoci che la lista è stata modificata, oppure quando l'utente rimuove i filtri, tramite il bottone *Rimuovi Filtri* nel fragmentFilter.
L'intera logica del filtraggio è quindi realizzata nell'adapter, ho preferito realizzarla in questo modo piuttosto che filtrando effettivamente gli elementi, tramite Query SQL sulla product_table. Questa scelta è dettata da due motivi principali, per ritornarmi una List<ProductEntity>seguendo il paradigma MVVM con Room avrei dovuto snaturalizzare parzialmente l'asincronicità dei thread, realizzando un HandlerThread e aspettandone il risultato. Ho valutato anche come opzione di ritornare una LiveData<List<ProductEntity>>, tuttavia questo avrebbe comportato l'attivazione di diversi observer che su certi eventi (e.g. cambio di categoria di un prodotto) avrebbero portato a mostrare una lista di prodotti non coerente.

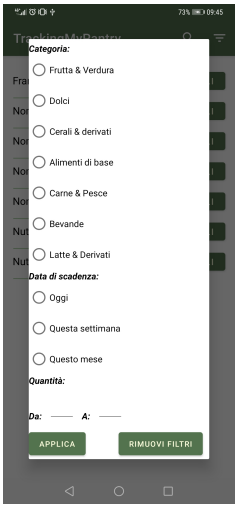

Figure 8: Schermata FilterFragment

5.3.2 ShowDetailsFragmentMain

Ad ogni elemento nella recyclerView della MainActivity è associato un button e un onClickListener su questo button, realizzato tramite interfaccia nell'adapter. Quando l'utente attiva questo listener, cliccando sul bottone *dettagli*, gli viene mostrato un fragment contenente le caratteristiche del prodotto. Le tre proprietà categoria, data di scadenza e quantità sono modificabili dall'utente tramite l'utilizzo di differenti dialog. I risultati così impostati vengono passati alla MainActivity tramite FragmentResultListener, che chiama poi il ViewModel per aggiornare la proprietà del prodotto selezionato.

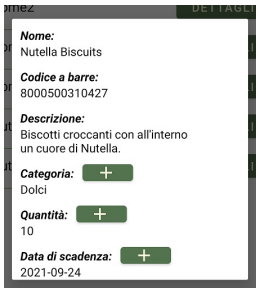

Figure 9: Schermata ShowDetailsFragmentMain

5.4 RemoveItemActivity

Questa activity chiede all'utente di inserire il nome del prodotto che vuole eliminare e sull'onClick del button *Rimuovi dalla tua dispensa*, passa il nome alla MainActivity, la quale si occupa di eliminare effettivamente l'elemento dal database locale.
Inzialmente, questa activity permetteva anche all'utente di eliminare un prodotto dal server remoto, quindi l'operazione REMOVE PRODUCTS. Ho deciso nella versione finale di rimuovere questa possibilità, poichè permette all'utente di eliminare prodotti a disposizione di tutta la comunità e questo potrebbe causare *side-effects* indesiderati, ritengo che questa logica sia più efficace se gestita lato server.

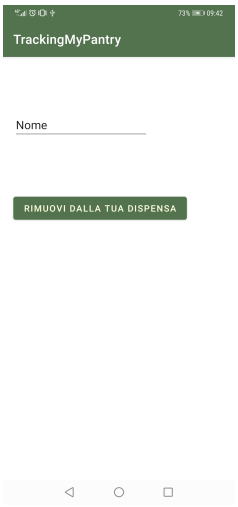

Figure 10: Schermata RemoveItemActivity

5.5 AddItemActivity

L'activity prende il codice a barre del prodotto da cercare e lo utilizza per interrogare il server sui prodotti associati a quel barcode. Il codice a barre può venire inserito in due modi, manualmente dall'utente, oppure attraverso scansione tramite la fotocamera. Il codice così ottenuto viene usato come campo nel corpo della POST getProductsRequest, se questa richiesta ha esito positivo lanciamo un Intent alla SelectItemActivity, passandogli il sessionToken e una List<Product>contenente la lista dei risultati da mostrare.
Inoltre, la AddItemActivity ascolta in attesa di due eventi, se l'elemento viene selezionato con una POST PRODUCT PREFERENCE o viene aggiunto con una POST PRODUCT DETAILS, in entrambi i casi passa poi l'elemento aggiunto alla MainActivity, questo meccanismo è realizzato tramite un ActivityResultLauncher<Intent>.

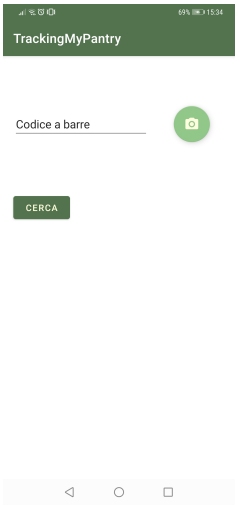

Figure 11: Schermata AddItemActivity

5.5.1 BarcodeScannerActivity

Questo componente utilizza la libreria CodeScanner, tramite un oggetto di tipo CodeScanner scannerizza il codice a barre e lo passa alla AddItemActivity. Il permesso per usare la fotocamera viene chiesto a run-time, quando l'utente clicca sul floating action button con l'immagine della fotocamera, se il permesso non viene concesso l'esperienza dell'utente non è limitata, può continuare ad avere le stesse funzionalità, dovendo però inserire il codice manualmente.

Figure 12: Schermata BarcodeScannerActivity

5.6 SelectItemActivity

La SelectItemActivity mostra all'utente tutti i risultati che sono presenti nel server, associati al codice a barre inserito nella AddItemActivity. Il meccanismo per mostrare i risultati è lo stesso che per la recyclerView della MainActivity, cambia solo che in questo caso invece che usare un ListAdapter, viene usato un semplice RecyclerView.Adapter, poichè la lista non deve reagire a cambiamenti. L'adapter si chiama SelectAdapter e dichiara anch'esso un'interfaccia per gestire il click sul button legato ad ogni elemento.

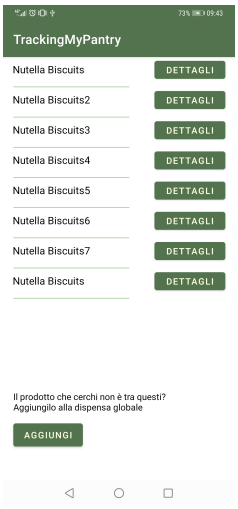

Figure 13: Schermata SelectItemActivity

Il click mostra uno ShowDetailsFragmentSelect, simile a ShowDetailsFragmentMain, contenente però solo gli elementi ritornati dal server, e contenente in più il button *Seleziona*. Sull'onClick di quel button il fragment notifica che l'elemento corrispondente è stato selezionato. La SelectItemActivity reagisce effettuando una POST PRODUCT PREFERENCE, aggiungendo un voto al prodotto scelto, in seguito passa il prodotto scelto alla AddItemActivity e poi viene tolto dal backstack. Nel caso nessun risultato fosse trovato, viene mostrata una textView che notifica l'utente.
Inoltre, se l'utente non fosse soddisfatto dei risultati trovati, tramite il click sul bottone *Aggiungi* può raggiungere la CompileItemActivity e inserire lui stesso l'elemento desiderato. La SelectItemActivity rimane in ascolto dell'eventuale inserimento e anche in questo caso reagisce passando l'elemento alla AddItemActivity e togliendosi dal backstack. In tutti questi passaggi nominati, le caratteristiche del prodotto sono passate attraverso un dato Parcelable, in particolare un'istanza della classe Product, che mi permette di gestire il passaggio dei dati meno verbosamente.

Figure 14: Schermata ShowDetailsFragmentSelect

5.7 CompileItemActivity

In questo componente gestisco l'inserimento di un nuovo elemento nel server remoto da parte dell'utente. Il nome, la descrizione e il codice a barre inseriti, vengono usati (insieme al sessionToken e al campo test) come oggetto della POST PRODUCT DETAILS. In caso di successo, il prodotto inserito viene passato alla SelectItemActivity e l'applicazione si toglie dal backstack, scatenando così una serie di rimozioni dal backstack necessarie per mostrare immediatamente all'utente la sua dispensa aggiornata. In caso negativo, l'utente viene notificato attraverso un Toast.

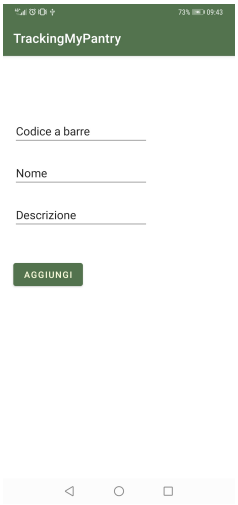

Figure 15: Schermata CompileItemActivity