BookYouLove

Sommario

Introduzione

BookYouLove si propone di essere un'applicazione di book tracking, in cui l'utente ha la possibilità di segnare i libri che sta attualmente leggendo, quelli che ha letto in passato e che ha intenzione di leggere; i libri possono essere aggiunti sia manualmente, sia tramite la scansione dell'ISBN, sia tramite ricerca online. In aggiunta vuole offrire un posto sicuro in cui l'utente abbia la possibilità di salvare le citazioni che trova interessanti nel corso della lettura, per poi poterle rileggere in un secondo momento, sia nell'applicazione stessa, sia in un widget su Home Screen; per le citazioni si dà la possibilità di aggiungerle tramite un meccanismo di OCR per scannerizzarle da una foto, senza bisogno di scriverle manualmente. Si forniscono infine grafici relativi ai vari dati raccolti.

Il linguaggio utilizzato per il progetto è Kotlin.

Per quanto riguarda l'architettura si è cercato quanto più possibile di rispettare l'architettura MVVM.

Per quanto riguarda l'orientamento del dispositivo, si permette la modalità landscape solo per i fragement di "Editing", ovvero per fragment in cui l'utente deve inserire input, mentre per gli altri si impone solo la modalità portrait.

L'applicazione è stata sviluppata principalmente per Smartphone piuttosto che per tablet.

I permessi richiesti sono l'accesso ad internet e l'accesso alla fotocamera

Librerie Esterne

Picasso

La libreria esterna "Picasso" è stata utilizzata per acquisire immagini (in particolare di copertine di libri) dalla rete.

Acquisizione

Per acquisire la libreria bisogna inserire le righe

```
def picasso_version = "2.71828"
implementation "com.squareup.picasso:picasso:$picasso_version"
```

all'interno del file gradle

Dipendenze

Nessuna dipendenza particolare

Permessi
Per poter utilizzare la libreria si deve inserire nel Manifest

```
<uses-permission
android:name="android.permission.INTERNET" />
```
Per poter utilizzare la connessione Internet.

MpAndroidChart
MpAndroidChart è stata utilizzata per costruire i grafici sulle statistiche di lettura.

Acquisizione
Inserire nel file Gradle

```
implementation 'com.github.PhilJay:MPAndroidChart:v3.1.0'
```
e

```
android {
…
                    repositories {
            maven { url 'https://jitpack.io' }
    }
}
```

Dipendenze e Permessi
Nulla in particolare

ML Kit
ML Kit (https://developers.google.com/ml-kit) è una libreria per il Machine Learning fornita da Google.

In questo caso sono stati utilizzati due gruppi di API:

- Text Recognition : Scansione testo della citazione da foto
- Barcode Scanning : Scansione ISBN del libro

ML Kit – Text Recognition
Permette di riconoscere il testo di Latin-Based Languages, con analisi della struttura del testo.

La struttura del testo è data da:

- Blocchi : Blocchi di testo (e.g Capoversi)
- Linee : Singole linee di testo
- Elementi : Singole Parole

Acquisizione

Per acquisire la libreria aggiungere nel Gradle:

```
dependencies {
  // ...

  implementation 'com.google.android.gms:play-services-mlkit-
text :recognition:17.0.0'
}
```

è poi raccomandato dalla guida di aggiungere nel Manifest

```
<application  . . .>
  . . .
  <meta-data
       android:name="com.google.mlkit.vision.DEPENDENCIES"
       android:value="ocr"  />
  <!-- To use multiple models: android:value="ocr,model2,model3" -->
</application>
```

Questo fa si che le API siano scaricate insieme all'applicazione dal Play Store, e non all'atto del prima chiamata.

Permessi e Dipendenze

Nulla di particolare

ML Kit – Barcode Scanning

Permette di leggere le informazioni codificate in barcode: la scansione avviene nel dispositivo e non necessita di connessione internet.

Acquisizione

Ci sono due modi per integrare il Barcode Scanning nell'applicazione:

- Legare il model all'applicazione in modo statico
- Non legare all'applicazione e scaricarlo dinamicamente dal Play Store

Per usare il metodo statico aggiungere

```
dependencies {
        // ...
        // Use this dependency to bundle the model with your app
        implementation 'com.google.mlkit:barcode-scanning:17.0.0'
    }
```

Per usare il metodo dinamico

```
dependencies {
        // ...
        // Use this dependency to use the dynamically downloaded model in Google Play Services
        implementation 'com.google.android.gms:play-services-mlkit-barcode-scanning:16.2.1'
    }
```

Per il metodo dinamico, aggiungere al Manifest

```
<application  . . .>
            . . .
            <meta-data
                android:name="com.google.mlkit.vision.DEPENDENCIES"
                android:value="barcode"  />
            <!-- To use multiple models: android:value="barcode,model2,model3" -->
        </application>
```

Per scaricare il model all’atto di download dell’applicazione e non all’atto della prima chiamata.

Permessi e Dipendenze

Nulla da segnalare

UCrop (https://android-arsenal.com/details/1/3054)

Libreria per eseguire il ritaglio e modifica di immagini. È stata usata per permettere all’utente il ritaglio della foto da cui viene scannerizzata la citazione

Acquisizione

Inserire nel Gradle

```
allprojects {
   repositories {
      jcenter()
```

```
      maven { url "https://jitpack.io" }
    }
}
```

E

`implementation 'com.github.yalantis:ucrop:2.2.6'` - lightweight general solution

Ci sono due modi per acquisire questa libreria:

- Lightway: Usata in questo caso perché impatta meno sulla dimensione dell'applicazione
- Native

Bisogna poi dichiarare l'activity di UCrop nel Manifest, in modo che questa possa essere lanciata

```
<activity
    android:name="com.yalantis.ucrop.UCropActivity"
    android:screenOrientation="portrait"
    android:theme="@style/Theme.AppCompat.Light.NoActionBar"/>
```

Permessi

Nulla da segnalare

Dipendenze

- Min SDK : 10
- CPU - armeabi armeabi-v7a x86 x86_64 arm64-v8a (versioni >= 2.1.2)

CardSlider (https://android-arsenal.com/details/1/7856)

Libreria per fornire un'interfaccia utente piacevole per i libri in stato di "In Lettura".

Acquisizione

Nel Gradle

```
allprojects {
  repositories {
   ...
   maven { url 'https://jitpack.io' }
  }
 }
```

E

```
implementation 'com.github.IslamKhSh:CardSlider:{latest_version}'
```

Permessi
Nulla da Segnalare

Dipendenze
Min SDK : 17

Database
Enitities
Vengono definiti due tipi di Entities:

- `Book`
- `Quote`

Book
È l'entity utilizzata per gestire il salvataggio delle informazioni relative al singolo libro nel database. I campi dell'entity sono

- `bookId : è un long utilizzato come chiave e viene assegnato in maniera univoca all'atto dell'inserimento del libro nel DB`
- `title : Titolo del Libro`
- `author : Autore del Libro`
- `startDate, endDate: Long per memorizzare la data; vengono utilizzati per impostare la timeInMillis di un Calendar da cui prendere poi giorno, mese e anno`
- `support : struttura dati per salvare il tipo di supporto utilizzato per leggere il libro`
- `coverName : link della cover`
- `pages :. Pagine del libro`
- `rate : struttura dati per i voti del libro`
- `finalThought : stringa per salvare i pensieri finali del libro`
- `readState : intero che indica lo stato attuale del libro (reading, toRead, ended). Viene usato allo scopo di gestire il passaggio da uno stato all'altro cambiando solo un intero, anziché costruire entities diverse e fare il passaggio tra queste.`

Il Book (e tutte le strutture interne) sono rese Serializable: questo permette di passare il Book come argomento (ad esempio di un Fragment) e di ridurre il numero di accesso al DB (come poi si vedrà).

Quote
Entity utilizzata per salvare una quote all'interno del DB. Le chiavi sono quoteId e bookId

I campi che la definiscono sono:

- `quoteId : Long assegnato univocamente alla citazione in fase di inserimento nel DB. Il quoteId è assegnato relativamente alle quotes che hanno lo stesso bookId`
- `bookId : Long che indica il libro a cui appartiene la quote`

- `quoteText: testo della citazione`
- `bookTitle, bookAuthor: titolo e autore della citazione. Sebbene sia possibile accedere a queste informazioni tramite il bookId, si è ritenuto più comodo e più efficiente, in termini di numero di accessi al DB, mantenerne copia all'interno della quote`
- `favorite: indica se la quote è in stato di "preferita" o meno`
- `toWidget: indica se la quote deve essere visualizzata nel widget (non utilizzato)`
- `quotePage : pagina della citazione`
- `quoteChapter: capitolo della citazione`
- `quoteThought: pensiero relativo alla citazione`
- `date: data in cui la citazione è stata presa. Viene assegnata senza intervento dell'utente e viene utilizzata per lo più per l'ordinamento delle citazioni in ordine temporale`

Anche la Quote è resa Serializable, per lo stesso motivo del Book.

Strutture di Supporto

Le seguenti strutture di supporto sono utilizzate nell'applicazione al fine di ridurre la quantità di informazioni che vengono mantenute in memoria ad un certo istante. Ad esempio, quando si visualizza la lista di tutti i libri letti, non viene visualizzato il finalThought (che può essere anche molto lungo); una cosa simile può dirsi per le quotes oppure per le informazioni da caricare per visualizzare i chart.

Queste strutture di supporto sono:

- `ShowedBookInfo`
- `ShowQuoteInfo`
- `ChartsBookData`

Grazie a Room si possono ritornare direttamente solo i dati che compongono queste classi.

Anche queste strutture sono Serializable.

MainActivity

La MainActivity viene utilizzata come gestore dei vari fragments. In particolare si è scelto, per rendere più semplice la transizione tra i fragment e il passaggio di argomenti tra di essi, di utilizzare la NavigationComponent fornita da Android.

NavigationComponent

La NavigationComponent richiede che si inserisca un NavHost nel layout di un'activity: è all'interno di questo NavHost che si alternano i vari fragment. Il NavHost deve essere legato ad un Navigation Graph, all'interno del quale vengono definite i fragment (detti "destinazioni") e le connessioni tra essi (dette "azioni").

Come NavigationUI si è scelta la BottomNavigationBar, che fornisce un'interfaccia gradevole e user frindly per lo spostamento tra le top level destinations. Per definizione, una Top Level destination è tale per cui, se l'utente è in una di esse, la pressione del Back porta ad uscire dell'applicazione.

In questo caso, si è preferito, piuttosto che avere un comportamento del suddetto tipo, fare si che l'unica destinazione da cui si possa effettivamente uscire alla pressione del back sia il "ReadingFragment", il quale funge anche da start_destination nel grafo di navigazione. Questo comportamento si è ottenuto impostando nelle azioni che conducono alle altre destinazioni un "Pop Behavior" di questo tipo: nel Back Stack delle destinazioni si eliminano tutte le destinazioni fino al ReadingFragment escluso (si è certi che il reading fragment sia presente nello stack perché è la start_destination).

Inoltre, poiché la BottomNavigationBar permette di inserire al suo interno un menu che ha al massimo sette items (con quattro il numero di items suggeriti), si è pensato di fare quanto segue: si sono inserite tre destinazioni principali, ovvero "Reading" , "Ended", "TBR" più una quarta, ovvero "Others". Others non è una vera e propria destinazione, ma un place holder: quando la destinazione Others viene cliccata viene visualizzato un popup menu con ulteriori destinazioni, ovvero "Quotes", "Charts", "Settings" verso cui si può effettivamente navigare.

Nel popup menu, per i vari items sono state impostate delle icone. Tali icone però non vengono visualizzate per default e quindi si invoca la *setForceIcon(true)*. Questa chiamata però funziona solo per VersionCodes maggiori o uguali a Q: per questo nel caso in cui si abbia una versione superiore il metodo viene chiamato, altrimenti non viene chiamato e le icone non sono visualizzate.

Settings

Le impostazioni dell'applicazione vengono gestite tramite la libreria Preferences fornita da Android. Questa libreria gestisce in automatico il salvataggio di coppie chiave-valore.

In questo caso, le impostazioni gestibili per l'applicazione sono:

- General.
  - Formato della data che si vuole utilizzare
- Ended
  - Tipo di Layout: Lineare o Griglia. La chiave è *endedLinearLayout*
  - Ordine di visualizzazione della lista: Chiave Associata *endedOrderPreference.* I possibili valori sono definiti nell'array *@array/ended_order_array_entry_values*
    - start_date: default

        - end_date
        - title
        - author

- TBR
    - Ordine di visualizzazione della lista. La chiave è *tbrOrderPreference.* I possibili valori sono definiti in *@array/tbr_order_array_values* e sono
        - title: default
        - pages
        - author

Per accedere al file in cui vengono salvate in automatico le preferenze si può utilizzare il *PreferenceManager.getDefaultSharedPreference* per poi invocare il metodo *getString* passando la chiave e il valore di default.

I valori assegnati alle preferenze sono utilizzati nei relativi fragments.

Vi è infine una Preference che non ha una key associata e che viene utilizzata come un bottone per andare al fragment di gestione di GoogleDrive.

Dialogs

AlertDialogFragment

Utilizzato semplicemente per dare una comunicazione all'utente. Il titolo del dialog fragment viene passato dal fragment parent come argomento. Viene impostato un singolo bottone che fa la dismiss del dialog

ConfirmDeleteDialogFragment

Utilizzato per chiedere all'utente conferma sulla cancellazione di un certo oggetto (book o quote).

Il titolo viene passato come argomento al fragment in modo da poter riutilizzare lo stesso dialog per chiedere conferma sulla cancellazione di diversi oggetti.

Per rendere il fragment indipendente da ciò per cui si chiede la conferma di cancellazione, il fragment, se viene data conferma, ritorna con chiave *deleteKey* una coppia *(deleteConfirm, true).* Su questa chiave si mette in ascolto il parent fragment il quale, quando riceve la risposta, reagirà secondo la sua logica.

CoverLinkPickerDialogFragment

Utilizzato per chiedere all'utente l'inserimento del link della cover di un libro.

In questo caso viene utilizzato un custom layout in cui viene inserita una EditText da compilare.

Quando viene inviata la conferma, se il testo della EditText non è vuoto, si imposta il risultato del fragment affinché questo possa essere preso dal parent fragement.

DatePickerFragment

Utilizzato per chiedere all'utente di inserire le date di inizio e fine delle sue letture.

Il Picker può essere chiamato per due motivi: chiedere la startDate o chiedere la endDate. Distinguere il tipo di chiamata è lo scopo dell'argomento *callMode*. Devo distinguere per forza i due tipi di chiamata perché è possibile che lo stesso fragement lo invochi sia per chiedere entrambi i tipi di date (e.g modifyEdndedFragment).

Nella fase iniziale la data del dialog viene impostata a data odierna.

In seguito vengono invocate le funzioni *setMinDate* e *setMaxDate*. Queste funzioni hanno lo scopo di porre un limite superiore e/o inferiore alla data che può essere impostata dall'utente. Ad esempio la endDate non può mai essere minore della startDate, oppure la startDate non può essere maggiore della endDate. Impostare questi limiti mi permette di evitare delle situazioni patologiche che possono presentarsi in altri fragment (in particolare nei fragements che gestiscono i charts).

Nella *setMinDate* la data minima viene impostata solo se il chiamante chiede la endDate e se il chiamante ha passate una minDate non nulla: la minDate viene passata con un Long utilizzato per impostare la data di un calendar tramite la *timeInMillis*

La *setMaxDate* è speculare alla *setMinDate*.

Quando l'utente dà conferma, viene scelta la chiave di ritorno in funzione del tipo di chiamante e impostato il bundle per il ritorno con un Long che rappresenta la data selezionata.

LeavingReadingFragment

Quando un utente si trova nel ReadingFragment e decide di abbandonare un libro ci sono due strade da prendere: eliminare definitivamente il libro, oppure spostarlo nella TBR per salvarlo come da leggere in futuro. Questo dialog permette di chiedere all'utente quale delle due strade vuole prendere.

A seconda della scelta fatta viene ritornato il risultato appropriato.

LoadingDialogFragment

Questo fragment inizialmente era stato utilizzato per evitare che l'utente interagisse con l'interfaccia durante un'operazione lunga.

Il dialog era stato reso non cancellabile.

Tuttavia questo creava un comportamento non user-friendly, impedendo all'utente di tornare indietro, ad esempio, se un'operazione va in blocco o richiede molto tempo.

La logica per gestire un’operazione lunga è gestita in un modo diverso, ma di questo fragment è stato mantenuto il layout.

Filters

Si tratta dei filtri che vengono applicati per i meccanismi di ricerca sia dei Books sia delle Quotes

BookListFilter

Prende come paramentri del costruttore:

- bookSetAll. Insieme dei libri a cui bisogna applicare il filtro
- bookSet. Insieme dei libri in cui bisogna mettere i filtrati
- filterType. Tipo di filtro da applicare:
  - titolo
  - autore
  - data
  - voto. In questo caso il valore ricevuto è un intero; per poter filtrare anche i mezzi voti, dato l’intero, si filtrano sa i libri che hanno voto l’intero sia l’intero + 0.5. Questo permette anche di evitare di ciedere un float all’utente e dover andare quindi a fare dei controlli sul tipo di input che viene inserito
  - titolo o autore. Utilizzata solo per la ricerca in Tbr in cui vi sono poche informazioni da gestire
- adapter. Adapter per il quale bisogna notificare il cambiamento dei dati da mostrare

QuoteListFilter

Stesso concetto di BookListFilter: la ricerca può essere fatta o per contenuto o per preferiti, ma non per entrambi insieme.

Reading

Reading

ReadingFragment

Vengono dichiarate come globali:

- `binding : per l’accesso e la modifica all’interfaccia utente`
- `readingVM : ViewModel`

Il ciclo di vita a cui è legato il ViewModel è l'activity. Il motivo per cui si è scelto di legarlo all'activity piuttosto che al singolo fragment è che quello di Reading viene considerato uno dei fragment a cui si va più di frequente. Per evitare quindi continui ricaricamenti dal DB della lista dei libri che sono correntemente in stato di "In lettura", si fa in modo che la lista perduri finché perdura l'activity.

Si imposta il menù, si inizializza il binding e si impostano gli observers.

Gli observers in questo caso sono due:

- currentListObserver. Si mette in ascolto del LiveData *currentBookList*, il quale viene aggiornato dal ViewModel ogni volta che c'è un cambio nella lista di libri attualmente in lettura.
  - Se l'array è non vuoto, esso viene passato a ReadingAdapter insieme al fragment corrente: passare il fragment corrente permette di reagire al click sul singolo elemento della lista come si vedrà in seguito; il risultato viene impostato come adapter di un cardSlider.
  - Se è vuoto invece si costruisce un array placeholder formato da un solo elemento e passato al ReadingAdapter.
- isAccessingObserver. È un booleano impostato a true dal ViewModel quando vengono eseguite operazioni lunghe come accesso al database o simili e passato a false quando queste operazioni sono terminate
  - Se true, l'effetto è quello di visualizzare un layout di loading con una rotella e di bloccare la reattività dell'activity. Questo però non impedisce all'utente di tornare indietro se cambia idea sull'operazione che sta facendo in quanto il back button è ancora attivo
  - Se false, il layout di loading viene fatto sparire e viene riattivata la reattività dell'activity

Il menù del fragment presenta un unico tasto di add. Quando l'opzione di add viene selezionata, l'effetto è quello di navigare verso il NewReadingBookFragment con argomento *readingModifyBook* posto a null (si veda il paragrafo *NewReadingBookFragment* per i dettagli e l'utilizzo di questo argomento).

Il ReadingFragment implementa poi un'interfaccia definita in ReadingAdapter, che permette al ReadingFragment di essere gestore degli eventi riguardanti i singoli item del menu sul CardSlider. Ci sono le seguenti possibilità:

- R.id.readingContextMenuTakeNoteItem
  - Si naviga verso il ModifyQuoteFragment, fornendo come argomenti le seguenti informazioni del libro selezionato, ottenuto tramite la variabile *bookArray* e tramite la posizione dell'item selezionato nel CardSlider
    - bookId
    - bookTitle
    - bookAuthor
- R.id.readingContextMenuDetailItem
  - Si naviga verso il DetailReadingFragment con argomento il bookId
- R.id.readingContextMenuTerminateItem
  - Si naviga verso EndingFragment con argomento il bookId
- R.id.readingContextMenuLeaveItem

        - Si mostra un'istanza di LeavingReadingDialog per chiedere all'utente cosa vuole effettivamente fare nel momento in cui abbandona il libro
        - Precedentemente si è impostato un listener per reagire al ritorno del valore di modo che se il ritorno è
            - True : si invoca il metodo moveReadingToTbr del VM
            - False : si invoca il metodo deleteReadingBook del VM
- R.id.readingContextMenuQuotesListItem
    - Si naviga verso il QuoteListFragment con argomento il bookId

ReadingAdapter

Il ReadingAdapter, come sono soliti gli Adapter, fa da ponte tra i dati da inserire in un ViewHolder e il ViewHolder stesso. In particolare la singola card del CardSlider presenta un ImageView e una Toolbar. La Toolbar si è inserita per fornire una sorta di menu contestuale che però non richiedesse il long click da parte dell'utente e fosse quindi di accesso più semplice e immediato.

Per la toolbar viene impostata la onMenuItemClick, la quale semplicemente invoca il metodo onReadingItemMenuItemClickListener del fragment che implementa l'interfaccia OnReadingItemMenuItemClickListener che, in questo caso, è proprio il ReadingFragment. Quindi quando viene selezionato il menù della toolbar di uno degli item del CardSlider, l'evento viene fatto gestire dal metodo del ReadingFragment, fornendo posizione della card nell'array e item del menu selezionato per quella card.

Poiché i libri in stato di Reading non dovrebbero essere molti, si è deciso di non implementare un meccanismo di ricerca, quindi l'Adapter non è Filterable.

BookListViewModel

Si tratta del ViewModel che viene utilizzato da TUTTI i fragment di visualizzazione di lista dei libri per gestire queste liste: è usato in particolare da ReadingFragment, EndedFragment e TbrFragment.

Quando uno di questi fragment si attiva, la prima cosa che fa è invocare una funzione tra *getTbrList*, *getReadingList*, *getEndedList*, in funzione dello stato dei libri che voglio visualizzare. Questi metodi non fanno altro che invocare la getRequestedList su uno specifico stato.

getRequestedList

Tramite un'istanza di BookListModel accede al DB per chiedere la lista dei libri nello stato richiesto; nota la lista, questa viene utilizzata per istanziare una BookList, che non è altro che il model che gestisce l'Entità lista di Book. Fatto ciò viene invocata la *sortBookArray*. La variabile loadedOnce serve a non ricaricare più del necessario le informazioni già caricate dal DB.

sortBookArray

Accede alle Preferences per vedere quale è il tipo di ordinamento che l'utente ha impostato: si utilizzano chiavi diverse in base allo stato dei Book della lista richiesta; per Books in stato Reading si utilizza come default un ordinamento per data di inizio.

Si invoca quindi un metodo della BookList che ordina la lista e si assegna al LiveData il valore della BookList.

changeSelectedItem

Viene invocato dal fragment quando viene selezionato un elemento nella lista; l'effetto è quello di invocare un metodo della BookList per cambiare un attributo che tiene traccia dell'elemento correntemente selezionato

Notify…

Si tratta dei vari metodi invocati dai fragment quando ricevono segnali di modifica della lista da visualizzare attualmente. Tutti questi metodi lanciano delle Coroutines e invocano dei corrispettivi metodi del BookList; terminati i metodi si assegna alla lista dei libri correnti il nuovo valore di BookList.

La *notifyBookMove* in particolare serve a gestire il passaggio di un Book da stato di Reading a stato di TBR

BookList

Si tratta della classe che gestisce la Lista di Books. Gli attributi sono:

- loadedArray. È l'insieme dei ShowedBookInfo da gestire; viene passata come parametro del costruttore
- selectedItem. Elemento che è stato selezionato all'interno della lista. È importante tenere traccia dell'item e non della sua posizione. Infatti è possibile che la lista che viene visualizzata non sia la lista totale, ma la sua versione filtrata: quindi la posizione che viene ritornata dall'evento di click non corrisponde per forza a quella dell'elemento nella loadedArray; tenere traccia dell'item invece mi permette di ritrovarlo e poterci eseguire delle operazioni. Per ritrovare la posizione del selectedItem nella loadedArray si utilizza il metodo *findOriginalPosition.*
- currentSortType. Attuale tipo di sorting della lista; ne viene tenuta traccia in modo da poterlo reinvocare dall'interno della BookList

on…

Metodi che vengono invocati dal VM quando si verificano dei cambiamenti nella lista; poiché quasi tutti richiedono di fare la ricerca della posizione dell'item nell'array originale, si passa a Dispatcers.Default

Sort…

Metodo invocato dal VM per il sorting; internamente vengono chiamati altri metodi di ordinamento.

Per l'ordinamento come data, poiché le date dei books sono prese come long, semplicemente si ordina in funzione di tali Long.

DetailReading

DetailReadingFragment

Fragment che mostra i dettagli relativi al singolo libro in lettura.

Si richiede l'istanza del DetailBookViewModel (legato al singolo fragment). Si prendono poi dal navArgs() gli argomenti passati al fragment. In questo caso l'argomento passato al fragment è il bookId del libro di cui si chiedono i dettagli.

Noto il bookId, il fragment invoca il metodo del ViewModel per il caricamento del libro di cui si chiedono i dettagli.

Vengono quindi impostati gli observer:

- `isAccessing : vedi sopra`
- `currentBookObserver : Si aggiorna la UI in funzione del libro che è stato caricato e si salva il libro corrente in una variabile` ***`detailBook`***

Il menu è costituito di un solo item di Edit. Quando selezionato invoca la navigazione verso il NewReadingBookFragment con argomento il *detailBook*. Passare direttamente il libro su cui bisogna fare le modifiche permette di evitare che il NewReadingBookFragment debba chiedere il ricaricamento nel database: l'alternativa infatti sarebbe stata passare il bookId e ricaricarlo. Del *detailBook* in realtà viene passata una copia: il motivo è che il passaggio dei parametri avviene per riferimento e quindi se non passassi la copia si potrebbero verificare delle inconsistenze tra ciò che è mostrato e gli aggiornamenti che vengono effettivamente fatti nel NewReadingFragment.

Il Fragment inoltre si mette in ascolto, tramite il navController di dati associati alla chiave *modifiedBook*: questi dati vengono impostati dal NewReadingBookFragment qualora venga eseguita una modifica sul libro ed essa venga salvata. Se si ricevono questi dati, il fragment comunica al VM il nuovo libro tramite la onReadingBookModified che porta l'aggiornamento nel VM del currentBook e quindi l'aggiornamento dell'interfaccia.

DetailBookViewModel

Si tratta del ViewModel che viene utilizzato da tutti i fragment che mostrano il dettaglio di un Book.

Semplicemente si carica il libro di cui si chiedono i dettagli dal DB tramite il bookId.

Il metodo loadDetailBook viene invocato dal fragment passando il BookId ricevuto come argomento per caricare il libro. Anche qui la loadedOnce serve a non fare caricamenti multipli.

deleteCurrentBook viene invocato dai fragment da cui si ha la possibilità di fare la cancellazione del Book che si sta visualizzando. A terminazione viene emesso il valore di deleteCompleted che porta il fragment invocante a invocare la pop sul Back Stack.

onBookModified viene invocato dal fragment quando gli viene notificata tramite la navicationComponent una modifica del book.

changeThought viene invocato dall'EndedDetailFragment quando gli viene notificato il cambiamento del pensiero finale di un libro.

DetailBookModel

Quando si invoca la cancellazione, vengono anche cancellate tutte le Quotes relative al libro.

NewReading

NewReadingBookFragment

Nonostante il nome, è il fragment che viene utilizzato non solo per creare nuovi libri in stato di "reading", ma anche per modificarli

Si chiedono istanze sia di ModifyBookViewModel sia di BookListViewModel:

- la prima è legata al fragment e viene utilizzata per gestire i cambiamenti che l'utente apporta sul libro
- la seconda è legata all'activity e serve notificare alla lista dei libri in stato di Reading l'aggiunta/modifica di un libro

Si prendono gli argomenti dal navArgs: in questo caso l'unico argomento è un istanza di Book. Tale istanza può essere anche null: sarà null nel momento in cui non ci sono libri da modificare e quindi il libro deve essere creato e si chiama il metodo del VM che permette di inizializzare un libro da creare; se invece è un'istanza valida allora viene inizializzato il currentBook del VM con il libro passato come paramentro. Il libro da modificare viene impostato tramite il metodo *setBookToModify* del newReadingVM.

Viene poi invocato il caricamento dell'array degli autori.

In questo fragment viene chiesto il permesso per l'utilizzo della fotocamera tramite la *requestPermissionLauncher*: il motivo è che è presente un'opzione del menu che porta al TakeBookIsbnFragment, il quale usa la fotocamera per catturare l'ISBN del libro.

Nella onCreateView vengono impostati vari clickListener e le doOnTextChanged per le editText e si imposta l'osservazione della chiave *scannedIsbnKey* ritornata dal

TakeBookIsbnFragment per ritornare l'ISBN scansionato: quando viene ritornato l'ISBN, questo viene passato al newReadingVM che lo elabora.

Gli observers che vengono impostati sono:

- authorListObserver. Riceve gli aggiornamenti sulla lista degli autori che viene caricata. Da questa lista si costruisce un adapter passato come paramentro alla setAdapter di della AutoCompleteEditText in cui l'utente può inserire l'autore.
- isAccessingObserver
- currentBookObserver. Quando ricevuto un valore viene aggiornata la UI in funzione dei vari campi
- internetAccessErrorObserver. Viene ricevuto un valore nel momento in cui lato VM si incontra un errore nell'accesso a internet (si veda ModifyBookViewModel). In funzione del valore ricevuto viene visualizzato un AlertDialog con titolo diverso per indicare il tipo di problema riscontrato.
- exitObserver. Riceve l'istanza finale del libro dopo che questa è stata salvata nel DB. A seconda se l'argomento *readingModifyBook* è null o meno, si opera in maniera diversa.
    - Null. Allora si è arrivati al fragment da ReadingFragment e quindi il libro è un nuovo libro: allora si utilizza il metodo *notifyNewReadingBook* del readingVM per notificare l'aggiunta del nuovo libro
    - != Null. Allora si è arrivati al fragment da DetailReadingFragment: allora si passa al fragment precedente nel Back Stack il modifiedBook con chiave *modifiedBook* e usando il metodo *notifyReadingBookModified* del readingVM si comunica la modifica
    - Comunicare le modifiche comporta l'aggiornamento della lista dei reading senza ulteriori accessi ai DB.

Per quanto riguarda l'aggiornamento del Book, le varie interazioni con le View del Layout comportano la chiamata ad un metodo del newReadingVM che aggiorna il valore corrente del book. Fanno eccezione a questo comportamento le:

- newBookCoverImageView.
    - Viene lanciato un Dialog Fragment per acquisire il link della cover.
    - Il valore ritornato dal child fragment viene salvato tramite il metodo *updateCoverLink* del VM e si invoca la libreria Picasso per visualizzare subito la nuova immagine
- newBookStartDateText
    - Viene lanciato un DatePickerDialog con
        - Caller = START_DATE_SETTER
        - Max(info). Non settate: in questo caso infatti, trattandosi di un libro in stato di Reading non è ancora definita una data di fine che fa da tetto a quella di inizio
    - Il valore ritornato è un Long che rappresenta la data impostata: questo valore viene passato ad un metodo del VM e poi vi viene invocato un metodo di DateFormatClass che dato il Long calcola una rappresentazione in stringa della data.
- newBookSaveButton
    - Quando Cliccato, si verifica se titolo e autore sono validi.
        - Qualora non lo fossero, si visualizza uno snackbar e si fanno passare le EditText in stato di errore
        - Se non lo sono si procede con il salvataggio invocando il metodo *addNewBook* del newReadingVM

Il menu di questo fragment presenta un unico item che, se selezionato, invoca la funzione per richiedere il permesso per la fotocamera. Se il permesso è concesso, allora si naviga verso il TakeBookIsbnFragment.

Un’ultima cosa da dire riguarda la EditText per prendere le pagine del libro: il numero di caratteri massimo che può essere inserito è 5 in quanto permettere un numero maggiore di caratteri potrebbe portare a situazioni di overflow essendo il campo *pages* di Book un intero; è comunque difficile che il numero di pagine di un libro superi le 5 cifre.

ModifyBookViewModel

I LiveData che sono definiti da questo VM sono:

- `currentBook. Libri corrente che viene mantento`
- `currentAuthorArray. Array di stringhe da cui si imposta un adapter per la EditText degli autori`
- `isAccessingDatabase`
- `canExitWithBook. Permette di notificare al View la terminazione delle operazioni di salvataggio e di ritornare il Book aggiornato`
- `internetAccessError. Permette di notificare al View errori di accesso ad internet`

Vengono poi definite due variabili per evitare ricaricamenti dal DB non necessari:

- `bookLoadedOnce`
- `authorsLoadedOnce`

In particolare la bookLoadedOnce è importante: mi permette non solo di evitare accessi non necessari al DB, ma anche di evitare che, in caso di modifica di libro esistente, queste modifiche siano resettate a seguito di un cambio di configurazione. Infatti quando il *bookToModify* del NewReadingBookFragment non è null viene invocato il metodo *setBookToModify* del VM: se non imponessi che il currentBook viene impostato solo se non è già stato fatto prima, ad ogni cambio di configurazione le modifiche fatte verrebbero perse.

prepareNewBook

è il metodo che viene invocato per creare una nuova istanza di Book. Il paramentro che viene preso è il bookState, ovvero lo stato in cui inseriamo il Book. Le informazioni inizializzate per il Book sono diverse a seconda dello stato in cui lo stiamo aggiungendo.

Il metodo viene invocato dal fragment qualora questo non abbia ricevuto argomenti, e quindi si sta effettivamente aggiungendo un nuovo libro.

La loadedOnce viene utilizzata per evitare che la inizializzazione sia invocata più di una volta, ad esempio a causa di un cambio di configurazione.

setBookToModify

Viene invocata dal fragment qualora abbia ricevuto un argomento: in questo caso non si sta creando un nuovo libro, ma si sta modificando un libro esistente, quindi viene modificato il currentBook a quello passato come parametro.

addNewBook

Funzione che invoca il salvataggio del libro nel DB. Si distinguono due casi in funzione di bookId

- `0. Allora si sta aggiungendo un nuovo libro`
- `!= 0. Allora si sta aggiornando in qualche modo un libro preesistente`

Modify…

Metodi che vengono invocati dai fragment per modificare gli attributi del current book in funzione di quella che è l'interazione dell'utente.

askBookByIsbn

Metodo che viene invocato dal fragment quando riceve un ISBN da TakeBookIsbnFragment. Quello che viene fatto è invocare un metodo della calsse GoogleBooksApi passando come paramentri del costruttore il VM stesso e la coda di richieste Volley.

Il VM implementa quindi un interfaccia Definita in GoogleBooksApi per ricevere le informazioni del libro di cui si è passato l'ISBN.

ModifyBookModel

computeNewBookId

Funzione che calcola il bookId del nuovo Book. Semplicemente si prende il massimo bookId e si somma 1.

changeQuotesInfoInDatabase

Invocata per aggiornare le informazioni sulle citazioni associate ad un libro modificato. In particolare vengono aggiornati titolo e autore mantenute nella Quote. Si può accedere alla lista delle citazioni del libro attraverso il bookId mantenuto in tutte le Quote

GoogleBooksApi

È la classe che serve per utilizzare le API di GoogleBooks.

Il metodo che viene esposto è findBookByIsbn.

findBookByIsbn e setNetworkResponseAsBook

Per trovare le informazioni del libro a partire dall'ISBN si utilizzano le API di Google Books. In generale queste API necessitano di chiave per accedere alle informazioni dell'utente, ma in caso di ricerca questo non è necessario.

Per eseguire una ricerca si può fare una richiesta http al link specificato in GOOGLE_BOOK_API_WITH_ISBN_URL. Si possono eseguire ricerche per vari campi, come ad esempio titolo e autore e, nel nostro caso, ISBN. Tutte queste

richieste ritornano risposte in formato Json da cui estrarre le informazioni relative al libro.

Tramite la libreria Volley, quindi, si fa una richiesta http per ottenere l'oggetto Json che descrive il libro con l'ISBN dato. Se la richiesta ha successo, allora si invoca la funzione *setNetworkResponseAsBook;* in caso contrario si aggiorna si invoca il metodo di ritorno del listener passando un valore di NetworkBook null.

Quando la richiesta ha successo si fa il parsing dell'oggetto Json ritornato estraendo le informazioni di interesse (in realtà ci sono moltissime informazioni che possono essere utilizzate per futuri aggiornamenti dell'applicazione per fornire una User Experience sempre più completa). Dopo il parsing dell'oggetto si invoca il metodo del Listener per ritornare il NetworkBook valido all'interno del quale sono contenute le informazioni da impostare per il currentBook del VM.

TakeBookIsbnFragment

È il fragment che permette, tramite la libreria CameraX e ML Kit, di scansionare l'ISBN di un libro.

Anche qui viene creato un piccolo VM, ovvero TakeBookIsbnViewModel, con un unico LiveData, *canExitWithIsbn* su cui viene impostato l'Observer *canExitWithIsbnObserver*: quando viene ricevuto una stringa contenente l'ISBN, si ritorna al fragment precedente nel Back Stack: Ritornarlo al fragment precedente nel Back Stack rende questo fragment indipendente dal fragment da cui si naviga verso di lui e quindi ne permette il riutilizzo (cosa fatta infatti in ModifyTbrFragment).

Ciò che lega le due librerie utilizzate è ImageAnalysis: questo oggetto permette di specificare chi è che analizza le immagini che vengono catturate dalla Camera e in questo caso è IsbnAnalyzer che sfrutta le API di ML Kit. In particolare nell'ImageAnalysis che viene costruita viene applicata la politica KEEP_ONLY_LATEST, che permette di non analizzare altre immagini finché il precedente non è stato analizzato. Per quanto riguarda la risoluzione ML Kit consiglia una risoluzione 1280x720 per utilizzi comuni.

Viene quindi legato il cameraProvider a i vari UseCases.

IsbnAnalyzer

Implementa l'interfaccia Analyzer di ImageAnalysis che contiene la funzione analyze.

Il metodo *analyze* riceve un imageProxy da cui è possibile estrarre la mediaImage e da questa l' image.

Viene quindi chiesto un'istanza di BarcodeScanning, con opzione che permette di accettare solo Barcode.FORMAT_EAN_13, il comune formato degli ISBN dei libri (ci sono molti altri formati che si possono accettare). Ottenuta l'istanza viene processata l'immagine

- Successo. Viene passato come parametro una lista di ISBN trovati nell'immagine. Si verifica che essa non sia vuota e, per semplicità, si utilizza solo il primo di questi. Questo valore viene utilizzato per aggiornare la *canExitWithIsbn* tramite il metodo *setScannedIsbn* del VM.
- Fallimento. Sollevo eccezione
- Completamento. Chiudo il proxy corrente: finché non lo faccio, non vengono acquisite nuove immagini da analizzare e quindi non metterlo significherebbe analizzare solo la prima immagine

Ending

EndingFragment

Si tratta del fragment che gestisce il passaggio tra stato di "In Lettura" a stato di "Terminato".

Vengono chieste le istanze dei seguenti ViewModel:

- endingVM. Legato al Fragment
- readingVM. Legato all'activity: per notificare il cambiamento della lista dei libri in lettura
- chartsVM. È il ViewModel che gestisce il fragment dei charts. È legato anche lui all'activity: il motivo di questa scelta è che dovendo, nella costruzione dei charts, fare una serie di calcoli, si preferisce caricare le informazioni ed eseguire i calcoli solo quando necessario, ovvero ad un evento di modifica dei dati stessi.

Il fragment riceve come argomento il bookId del libro che deve essere terminato e tramite metodo del VM ne chiede il caricamento.

Gli observer che sono impostati in questa fase sono:

- terminateBookObserver. Serve a ricevere il valore del libro da terminare per preparare la UI
- canExitObserver. Serve a comunicare al fragment la terminata operazione di salvataggio del libro terminato nel DB, in modo che possa comunicare:
  - terminazione del libro a readingVM
  - cambiamento dei dati dei grafici a chartsVM
- isAccessingDatabase

Vengono impostati i gestori delle interazioni delle varie view all'interno del layout. In particolare, quando premuta la CardView contenente la EndDate, si lancia un DatePickerDialog con argomento un Long che rappresenta la startDate del libro. Quando il datePicker ritorna il valore, questo valore viene ricevuto, questa viene passata come metodo dell' endingVM per impostare la data del libro corrente e ne viene calcolata una rappresentazione stringa.

EndingViewModel

Quando il libro viene caricato, prima di essere emesso il valore del LiveData, viene invocata la funzione *prepareTerminateBook*. In questa funzione vengono impostati i valori che caratterizzano un libro terminato rispetto ad uno in lettura; questi sono:

- `voti. Impostati a 0 per default`
- `finalThought. Stringa Vuota`
- `endDate. La endDate per default è impostata alla data di inizio del libro. Il motivo di ciò è che non farlo può portare a situazioni di inconsistenza nelle date. Supponiamo, ad esempio, che l'utente imposti la data di inizio del libro al 20 Gennaio 2020, sebbene la data odierna sia il 15 Gennaio; se l'utente andasse a terminazione il 17 Gennaio e non aprisse il DatePicker per prendere la data (che nel Picker è limitata perché impostato l'argomento min) avrei una endDate < startDate, con conseguenti problemi in altre situazioni.`

EndingModel

Nella fase di salvataggio lo stato del libro viene passato a *ENDED_BOOK_STATE* e il libro viene salvato in DB.

Ended

Ended

EndedFragment

Si tratta del fragment in cui è visualizzata la lista dei libri "Terminati".

Viene presa l'istanza di BookListViewModel legata al fragment e inizializzate le variabili per la gestione della ricerca. Il motivo per cui si è deciso di prendere il VM legato al fragment è che la quantità di informazioni che sono caricate in questo fragment può essere elevata quando il numero di Books comincia ad essere alto e, poiché visualizzare i libri letti è un operazione poco frequente e, se eseguita, comunque è continuativa, si è deciso quindi di ricaricare la lista ogni volta.

Nella onCreate viene restaurato il valore di *searchField*(vedere dopo) e richiesta al VM il caricamento della lista.

Viene inizializzato un piccolo spinner in cui l'utente ha la possibilità di inserire il tipo di informazione per cui eseguire la ricerca. I possibili tipi di ricerca sono:

- `Titolo`
- `Autore`
- `Voto`
- `Anno`

Poiché i tipi di dato per cui la ricerca si effettua sono diversi (stringa per titolo e autore , intero per voto e anno), per evitare problemi nell'esecuzione della ricerca si impone che il tipo di ricerca possa essere cambiato solo quando la searchView è vuota; questo comportamento è garantito disabilitado lo spinner quando il testo nella

SearchView è diverso dalla stringa vuota. Non fare questo infatti potrebbe comportare il seguente problema: un utente esegue ricerca per titolo (stringa) e poi , senza cancellare il precedente testo della SearchView, passa a ricerca per anno (intero); di conseguenza appena digita un nuovo carattere ho il confronto di una stringa (il vecchio titolo) con un intero (le pagine del book) (vedere in seguito per dettagli sulla gestione della ricerca). Quando viene cambiato il tipo di ricerca che si fa si cambia il tipo di tastiera di input della searchView.

Gli Observers che sono impostati sono:

- `isAccessingObserver`
- `currentReadListObserver. Riceve il valore relativo alla lista dei libri caricati. Quando Ricevuto si verifica quale layout l'utente preferisce accedendo al *defaultSharedPreferences*: in funzione della preferenza si imposta il LayoutManager della RecyclerView. Si salva poi il valore dell'array corrente in una variabile e si imposta l'adapter della RecyclerView ad un'istanza di EndedAdapter, passando l'array dei valori caricati, il fragment corrente che implementa l'interfaccia per reagire al click sul singolo elemento della RecyclerView e il linearIndicator.`

Nella onPrepareOptionsMenu si restaura il precedente stato della SearchView:

- `se searchField è vuoto, allora l'utente non aveva ancora eseguito ricerca, oppure l'aveva terminata: iconifico la SearchView`
- `se non è stringa vuota, allora c'è una ricerca in corso, quindi reimposto il testo nella SearchView.`
- `Si imposta la setOnQueryListener. È importante impostare la onQueryListener PRIMA della restaurazione della query; impostarla dopo, infatti, comporta che non c'è ancora nessuno che reagisce al cambio di testo e quindi il filtraggio non viene rieffettuato, comportando quindi la perdita della lista filtrata al cambio di configurazione.`

Nella *onQueryTextChange* si abilita e disabilita lo spinner in funzione del testo della query (se è vuoto si permette il cambio di tipo di ricerca, altrimenti no), si prende la posizione corrente dello spinner e si invoca il metodo dell'adapter che permette di filtrare la lista. In questa fase è importante fare la verifica che l'adapter non sia null: è possibile infatti che vi siano casi in cui all'interno della SearchView rimangono dei residui di ricerca effettuati in altri fragment e ciò porterebbe una ricerca ad essere eseguita con adapter ancora non impostato. L'EndedAdapter espone un attributo per specificare il tipo di ricerca da effettuare: il tipo di filtro quindi si imposta a quello impostato nello Spinner. Prima di invocare il metodo di filtraggio dell'adapter si effettua un controllo sul tipo di input e sulla sua lunghezza: se si sta eseguendo una ricerca per voto o anno, non si può permettere l'inserimento di un input eccessivamente lungo, in quanto si rischierebbe overflow e quindi la ricerca è effettuata solo se la lunghezza dell'input è inferiore di 4 caratteri, che è il numero di cifre che compongono un anno.

La *onRecyclerViewItemSelected* è il metodo dell'interfaccia implementato dal fragment per reagire al click sull'elemento della RecyclerView. Riceve come paramentro la posizione dell'elemento nella lista corrente. Quando eseguito, si accede all'attuale bookSet dell'EndedAdapter e, attraverso la posizione, si estrae l'item selezionato che viene passato al endedVM (BookListViewModel) per impostare l'item corrente del suo BookList.

EndedAdapter

Riceve nel costruttore

- bookSetAll. Lista completa dei ShowedBookInfo
- Listener per il click sull'item
- linearLayoutIndicator. Indica se l'utente vuole un layout lineare o griglia

Vengono inizializzati gli attributi

- filterType. Per il tipo di filtraggio che l'utente ha selezionato
- bookSet. È l'insieme di ShowedBookInfo che vengono effettivamente utilizzate per la visualizzazione e in cui vengono messi gli elementi che vengono filtrati

Viene implementato come Filterable e si ritorna un BookListFilter.

BookListFilter

È il fitro che viene utilizzato nelle liste di Books in cui è implementata la ricerca.

Prende nel costruttore:

- bookSetAll
- bookSet
- filterType. Tipo di filtraggio da attuare
- adapter. Adapter per il quale bisogna eseguire il filtraggio. Passarlo permette di invocare la corrispettiva *notifyDataSetChanged()* per notificare l'avvenuto filtraffio della lista

EndedDetail

EndedDetailFragment

È il fragment che mostra i dettagli dei libri "terminati". Vengono chieste le istanze di tre ViewModels:

- `DetailBookViewModel.`

- ChartsViwModel. Permette di notificare l’avvenuto cambiamento delle informazioni in caso di cancellazione di un book.
-

L’argomento passato è il bookId del libro di cui si chiedono i dettagli: tramite questo si può chiedere al detailEndedVM di eseguire il caricamento.

Si imposta l’osservazione di due chiavi nel savedStateHandle:

- changedFinalThoughtKey. Permette di ricevere il cambiamento del finalThought del libro corrente. Quando ricevuto questo viene preso e salvato in una variabile e viene poi invocato il metodo dell’detailEdedVM che permette di aggiornarlo nell’istanza di Book da lui mantenuta
- endedModifiedBook. Permette di ricevere l’istanza di libro quando questa viene modificata (dal ModifyEndedFragment) e di comunicare, tramite i loro metodi, l’aggiornamento sia ad endedDetailVM sia a ad endedModel

Per quanto riguarda gli observer vengono impostati:

- isAccessingObserver
- currentBookObserver
- deleteCompletedBook. Viene ricevuto il valore a seguito di una cancellazione avvenuta con successo. In questo caso si notifica il cambiamentosia a chartsVM sia, tramite savedStateHandle, al previous fragment nel Back Stack (il previous sarà EndedFragment) e si torna indietro

Nel menu ci sono quattro items:

- Delete
  - Lancia un DeleteConfirmDialogFragment per richiedere conferma all’utente. Il risultato di questo fragment viene catturato e, se positivo, viene invocato il metodo del VM che provvede alla cancellazione del libro
- Edit.
  - Permette la navigazione verso il ModifyEndedFragment passando come argomento l’intero libro da modificare (una sua copia)
- takeNote
  - permette di navigare verso il QuoteModifyFragment per aggiungere una citazione e passando come argomento
    - modifyQuote. Impostato a null: la quote è da aggiungere e non da modificare
    - bookId. Id del libro attuale
    - title
    - author

Ci sono per finire due bottoni che permettono la navigazione verso

- QuoteListFragment. Per listare tutte le citazioni relative a questo libro
- FinalThoughtFragment. Per leggere e/o modificare il finalThought del Book; vengono passati sia il finalTought (salvato in variabile) sia il bookId

EndedThought

EndedThoughtFragment

Fragment che permette la lettura e la modifica del pensiero finale relativo al libro.

Viene chiesta un'istanza di ViewModel (EndedThoughtViewModel) e istanziata una variabile *isEditing* per gestire lo stato del fragment (modalità editing e lettura). La Variabile *isEditing* viene salvata in un bundle a seguito del cambio di configurazione e restaurata in fase di onCreate.

Dopo la restaurazione viene invocata la setUI che si occupa di modificare la UI in funzione dello stato attuale:

- `Editing. Viene fatta scomparire la TextView che mostra il pensiero finale e sostituita con una EditText`
- `Reading. La EditText viene portata in stato di GONE e disabilitata mentre la TextView viene fatta comparire.`

Viene impostata la *doOnTextChanged* della EditText che, quando invocata, invoca un metodo del VM per salvare il pensiero corrente.

Nella *onPrepareOptionsMenu* si cambia l'icona dell'unico item del menu a seconda che si sia in uno o nell'altro stato.

Nella *onOptionsItemSelected* si modifica la reazione alla selezione dell'item del menu in funzione dello stato del fragment: se si sta editando e si clicca, allora si invoca il salvataggio nel DB del pensiero finale e il ritorno del nuovo pensiero finale al fragment precedente nel Back Stack; altrimenti si modifica l'icona; in entrambi i casi si invoca la *setUI*.

EndedThoughtViewModel

La modifica del DB dovrebbe essere, per rispettare l'architettura MVVM, all'interno di un Model; trattandosi però di una singola riga di codice si è preferito evitare.

ModifyEnded

ModifyEndedFragment

Si tratta del fragment per la modifica di un libro già terminato.

Si chiedono le istanze di ChartsViewModel e ModifyBookViewModel, il primo legato all'activity e il secondo legato al Fragment.

Si prendono poi gli argomenti da navArgs: l’argomento dato a questo fragment è il Book da modificare. Se il Book è diverso da null, allora si invoca il metodo del modifyEndedVM per impostarlo come current.

Vengono impostati i gestori per le varie interazioni con le View nel Layout e vengono impostati gli observers.

Per gli observers abbiamo:

- currentBookObserver. Imposta le varie Views del Layout e salva la endDate e la startDate in due variabili in modo che possano essere accessibili successivamente
- isAccessingObserver
- canExitObserver. Riceve l’istanza del libro a seguito del suo salvataggio. Questo Book viene tornato al Fragment precedente nel Back Stack e viene poi invocato il cambio dello stato delle informazioni del fragment dei Charts (può essere ad esempio cambiato un numero di pagine o delle date). In questo caso, la modifica del libro alla lista di Ended non viene segnalata direttamente dal NewReadingBook, ma viene segnalata dal DetailEndedFragment quando riceve il libro modificato (si sarebbe comunque potuto utilizzare anche qui il metodo onNotifyBookChanged di endedVM): il motivo di questa differenza è che tanto l’endedDetailFragment ha l’istanza di endedVM per comunicare la cancellazione e quindi, sebbene creare un’istanza anche in questo fragment non comporti ulteriore dispendio di memoria, si è preferito far passare la comunicazione per il fragment intermedio.

Per quanto riguarda le reazioni al click delle views, ho una gestione normale tranne che per:

- startDateCard. In questo caso, trovandoci in un libro terminato ho una endDate e quindi una data massima che la startDate può assumere; quando chiamo il DatePickerDialog quindi passo il maxDateMillis.
- endDateCard. In questo caso ho una startDate e quindi una data minima di cui passo i riferimenti

Quando premuto il bottone dei salvataggio viene invocato il metodo del ViewModel che salva il libro

TBR

TBR

TbrFragment

Si chiede l’istanza di BookListViewModel; tale istanza è legata al fragment: Il motivo di questa decisione è che vedere i libri da leggere è considerata un’azione rara e non frequente come può essere invece la consultazione dei libri letti o in lettura.

Anche qui vengono salvate delle variabili per la gestione della ricerca all’interno del fragment: seacrhField viene restaurato nella onCreate.

Viene quindi caricata la lista dei libri da leggere.

Vengono impostate le reazioni a vari eventi:

- `modifica`
- `aggiunta`
- `inizio`
- `cancellazione`

di un libro da leggere. In ognuno di questi casi si invoca un metodo del VM per reagire all'evento o si naviga verso un fragment per eseguire l'operazione.

Gli observers impostati sono:

- `isAccessingObserver`
- `currentTbrArrayBookObserver. L'array ricevuto, che contiene le ShowedBookInfo dei libri in stato di "da leggere" viene passato al TbrAdapter insieme a` *`this`* `per impostare il fragment come listener dell'evento di click non su uno degli elementi, ma sulla toolbar di uno degli elementi.`

Il metodo implementato per reagire a questa intrazione è onTbrListItemToolbarMenuClicked. Nel menu della toolbar ho item per:

- edit. Navigazione verso ModifyTbrFragment con parametro l'intero showedBookInfo.
- Start. Navigazione verso lo startFragment con argomento l'id del libro che si vuole cominciare
- Delete. Apertura di un DeleteConfirmDialog

Nel caso di Tbr, trattandosi di poche informazioni da mostrare (titolo, autore, pagine) si è deciso di non creare uns shermata per il detail, ma di mostrare tutte le informazioni nell'elemento del RecyclerView; l'utente ha quindi la possibilità di accedere alle varie funzioni su uno di questi item direttamente dalla toolbar del singolo.

Per quanto riguarda la gestione della ricerca è molto simile a quella dell'EndedFragment: viene implementata la onTextChanged e quando il testo cambia si invoca il filtraggio sull'adapter della RecyclerView. A differenza dell'EndedFragment, la ricerca è consentita solo per titolo e autore: fare una ricerca per pagine potrebbe rivelarsi complicato in quanto ricercare il numero esatto di pagine fornirebbe un User Experience non molto buona e ricercare un intervallo di pagine potrebbe non restringere troppo il campo.

TbrAdapter

Molto simile all'EndedAdapter. Anche qui ho un attributo pubblico tbrSet che viene esposto per permettere al gestore dell'evento di reperire, tramite la posizione, l'item

che è stato cliccato. Anche qui ho l'implementazione dell'interfaccia Filterable e la getFilter ritorna sempre un BookListFilter con il filterType impostato staticamente a SEARCH_BY_TITLE_OR_AUTHOR.

TbrModify

Il tutto è gestito in maniera molto simile alla NewReadingBook, compresa l'acquisizione dell'ISBN del libro, la richiesta con Volley ecc.

L'unica differenza sostanziale è che in questo caso l'argomento passato non è un Book ma uno ShowedBookInfo, in quanto esso contiene tutte le informazioni necessarie da visualizzare nel TbrModifyFragment

Starting

StartingFragment

È il fragment che permette all'utente di cominciare un nuovo libro. Riceve come argomento il bookId del libro da cominciare e, attraverso di esso, si chiede al VM il caricamento di tale libro.

Vengono chieste le istanze di:

- `BookListViewModel legato all'activity. L'unica istanza di BookListViewModel legata all'activity è quella di ReadingFragment e quindi questo VM mi permette di comunicare l'aggiunta di un libro in lettura`
- `StartingViewModel`

Vengono quindi impostati i gestori dei click sulle varie View del Layout e il listener per la startDate ritornate da DatePickerFragment. Il listener invoca un metodo del VM per impostare la startDate aggiornata e la visualizza nel Layout.

Gli Observers impostati sono:

- `isAccessing`
- `canExitWithBookObserver. Viene utilizzato per comunicare l'avvenuto passaggio allo stato di "In Lettura" del libro. Quando ricevuto il valore si utilizza sia per notificare l'aggiunta di un nuovo libro al readingVM, sia per notificare la rimozione all'tbrFragment`
- `currentBookObserver`

StartingViewModel

loadStartingBook, prepareStartingBook

La *loadStartingBook* viene invocata quando il fragment chiede di caricare il Book da segnare come started. In seguito al caricamento viene invocata la *prepareStartingBook* sul libro caricato.

La *prepareStartingBook* ha lo scopo di impostare le informazioni che distinguno un libro in stato di "In Lettura" da uno in stato di "Da Leggere". Queste informazioni in

particolare sono la startDate, inizializzata a data odierna, e il supporto di lettura, inizializzato tutto a false

Quotes

QuoteList

QuoteListFragment

È il fragment con cui viene visualizzata la lista delle citazioni salvate dall'utente.

Si prende un'istanza di QuoteListViewModel e gli argomenti da navArgs. In questo caso l'argomento passato al fragment è il bookId, per indicare di quale libro debbano essere mostrate le citazioni. Se il bookId è 0, significa che il libro non è specificato, quindi devono essere caricate le citazioni di tutti i libri; se è un bookId valido allora si caricano le citazioni del singolo libro.

Si inizializzano delle variabili per la gestione della ricerca.

Si impostano i gestori per la ricezione di valori dal navController e in entrambi i casi si chiamano dei metodi del VM per gestire l'evento; gli eventi a cui si risponde sono:

- delete
- modified

e sono ricevuti entrambi da DetailQuoteFragment. Quando ricevuti si invocano i corrispettivi metodi del VM

Gli observers implementati sono:

- isAccessing
- currentQuoteArrayObserver. Riceve l'array di quotes che è stato caricato dal VM. Lo passa a QuoteListAdapter insieme a this, per indicare che il fragment implementa l'interfaccia OnQuoteListHolderClick per rispondere al click sul singolo elemento della RecyclerView.

La ricerca è gestita sempre allo stesso modo.

Quando un elemento viene cliccato, viene estratto, tramite la posizione, l'item cliccato dal quoteSet dell'adapter e poi si naviga verso QuoteDetailFragment, passando il bookId e il quoteId dell'elemento cliccato.

Anche qui per motivi di risparmio di memoria non si caricano tutte le informazioni della quote, ma solo un sottoinsieme, che corrisponde all'insieme di informazioni necessarie alla visualizzazione. Un utente infatti potrebbe solo voler rileggere le citazioni, senza andare poi effettivamente a vederne i dettagli.

QuoteListAdapter
È l'adapter per la lista di quotes.

Anche qui viene esposto un quoteSet a cui il fragment può accedere per ricavare l'elemento che è stato selezionato.

Viene implementata l'interfaccia Filterable che, con il suo metodo *getFilter* ritorna un QuoteListFilter

QuoteListViewModel
Viene creata un'istanza di QuoteList per gestire la lista delle citazioni

getAllQuotes , getQuotesByBookId
Metodi chiamati per il caricamento della lista di quotes.

La prima è chiamata in caso di bookId non specificato, mentre la seconda in caso lo sia.

Entrambe modificano loadedOnce.

onQuoteDeleted , onQuoteModified
Metodi chiamati nel caso in cui il QuoteListFragment riceva dei valori dal navController.

Entrambe invocano i corrispettivi metodi della QuoteList e, dopo la terminazione dell'esecuzione dei metodi, estraggono il valore corrente di QuoteList. Per eseguire i metodi vengono lanciate delle Coroutines perché i metodi di BookList invocano la ricerca dell'item selezionato all'interno della lista originale (sempre perché la posizione nell'originale può non corrispondere alla posizione nel visualizzato a causa del meccanismo di ricerca).

changeSelectedQuote
Viene inocato dal fragment quando viene selezionata una quote e ha come effetto quello di invocare un metodo per cambiare l'elemento corrente della QuoteList.

QuoteListModel
sortShowInfoByDate
All'interno di ShowQuoteInfo è presente un campo date. Questo campo è impostato in automatico all'atto dell'aggiunta della quote e non può essere modificato dall'utente. L'ordinamento dell'array che viene caricato dal DB viene fatto in base a questo campo in maniera del tutto simile a come viene fatto l'ordinamento per date degli array di Book.

L'array di quotes quindi, dopo essere stato caricato, viene ordinato in base alla data di salvataggio della citazione.

QuoteList

È la classe che rappresenta la lista di Quotes. Il funzionamento è molto simile alla BookList.

QuoteDetail

QuoteDetailFragment

Fragment che permette di visualizzare i dettagli relativi alla citazione presa.

Prende come argomento:

- bookId
- quoteId

che insieme formano la chiave della citazione nel DB: questi argomenti vengono passati al metodo del VM che carica la citazione.

Viene impostato il gestore per la ricezione del risultato di un childFragment per la cancellazione: quando ricevuto si comunica al fragment precedente nel BackStack, ovvero quoteList che c'è stata una cancellazione.

Si imposta poi il gestore per la ricezione della modifica della quote. Quando si riceve una modifica, ed in particolare si riceve la quote modificata (cosa che si può fare in quanto la Quote è stata resa serializable), si invoca un metodo del VM per gestire la modifica della quote, e si comunica questa modifica anche al fragment precedente nel Back Stack (ovvero QuoteListFragment).

Nel menu abbiamo tre items per:

- edit. Quando selezionato, si naviga verso ModifyQuoteFragment, passando come argomenti della navigazione
  - copia della quote
  - bookId = 0
  - bookTitle , bookAuthor = null
  - Vedere seguito per significati di questi argomenti
- delete. Lanciato un DeleteConfirmDialogFragment per chiedere conferma della cancellazione.
- Share. Viene costruita la forma della citazione da condividere, l'intent per la condivisione e si fa la startActivity.

QuoteDetailViewModel

Nulla da Segnalare

QuoteDetailModel

Nulla da Segnalare

ModifyQuote

ModifyQuoteFragment

È il fragment che permette di aggiungere una nuova Quote. Da questo fragment è possibile navigare verso il QuoteWithCameraFragment con cui si da la possibilità di scannerizzare la citazione. È necessario quindi chiedere il permesso per la camera qualora questo non sia già stato dato in precedenza: il meccanismo di richiesta del permesso è simile a quello utilizzato in NewReadingBookFragment.

A seconda degli argomenti del fragment abbiamo due casi:

- modifyQuote != null. Significa che siamo in caso di modifica di una Quote preesistente e quindi bisogna impostare questa come currentQuote del VM: ciò viene fatto attraverso un metodo del VM.
- modifyQuote == null. Significa che siamo in caso di aggiunta di una nuova quote. In questo caso vengono passati come argomenti non nulli
    - bookId
    - title
    - author

  Il motivo per cui vengono passati questi valori è che quando si aggiunge una quote è necessario conoscere il bookId del Book a cui questa Quote appartiene. Questi parametri quindi vengono impostati ogni volta che dal ReadingFragment o dall' EndedDetailFragment si vuole aggiungere una quote.

  Titolo e autore invece vengono passati per non doverli andare a recuperare dal database tramite bookId: quando viene chiesta di aggiungere la quote di un libro infatti il titolo e l'autore di tale libro sono già noti.

Vengono quindi impostati i gestori del cambiamento delle varie View del Layout ed in particolare delle EditText, ognuna delle quali invoca un diverso metodo del VM per modificare un campo della *currentQuote*.

Si imposta quindi un gestore per la ricezione dal NavController di un valore con chiave *scannedQuoteKey*. Il valore associato a questa chiave corrisponde alla quote che viene scannerizzata dal QuoteWithCameraFragment. Quando il valore viene ricevuto, non c'è bisogno di invocare il metodo del VM, ma semplicemente cambiare

il testo della EditText della quote: il cambio di testo attiva la relativa *doOnTextChanged* che invoca a sua volta il metodo del VM.

Per quanto riguarda gli Observers abbiamo:

- isAccessing
- currentQuoteObserver. Riceve il valore della currentQuote e aggiornal la UI
- canExitWithQuoteObserver. Riceve la Quote finale dopo che questa è stata salvata nel DB. Nel caso in cui la quote passata per argomento sia nulla, allora è un'aggiunta semplice, quindi si torna indietro e basta; altrimenti la modifica della quote deve essere comunicata al chiamante del Fragment e quindi si ritorna il valore al fragment precedente nel Back Stack.

Nel menu abbiamo due item:

- favorite
- addWithCamera. Se selezionato invoca la richiesta di permesso. Se il permesso è dato allora si naviga verso il QuoteWithCameraFragment

ModifyQuoteViewModel

La Quote viene inizializzata con dei valori di default, di cui in particolare

- quoteId = 0. Impostata in fase di salvataggio nel DB
- bookId = 0. Impostata quando il fragmento comunica le informazioni relative al libro di cui si prende la citazione (se si sta prendendo una nuova Quote)
- startDate = data odierna

setQuoteBookInfo

Imposta i campi della Quote quando stiamo creando una quote da zero

saveQuote

In questa funzione si distingue il caso di modifica da quello di creazione tramite il quoteId.

- quoteId = 0. Allora non è stata mai impostata dal fragment una quote preesistente e quindi è una nuova
- quoteId != 0. Allora si tratta di una modifica di una vecchia quote, quindi invoco l'update.

In entrambi i casi, al termine dell'operazione di salvataggio nel DB, viene emesso il valore di *canExitWithQuote*.

ModifyQuoteModel

computeNewQuoteId

è la funzione che calcola il nuovo quoteId da assegnare ad una Quote che viene aggiunta per la prima volta nel DB. Il quoteId non è globale nel DB, ma è relativo al libro a cui la Quote appartiene, quindi deve essere calcolato in funzione delle altre quote dello stesso libro. Da ciò quindi il caricamento delle quotes dello stesso libro. Per il resto il quoteId si assegna sempre come massimo quoteId già esistente + 1

QuoteWithCamera

QuoteWithCameraFragment

È il fragment che permette all'utente di scannerizzare la citazione tramite la fotocamera.

Viene richiesta un'istanza di QuoteWithCameraViewModel sui cui LiveData vengono impostati due observers:

- isAccessing
- canExitWithTextObserver. Riceve il valore del testo scannerizzato dopo che questo è stato elaborato. Quando il testo è ricevuto,
    - null. Mostro un alert dialog di errore
    - valido. Lo ritorno al fragment precedente nel Back Stack (il quale sarà ModifyQuoteFragment, che lo riceve e lo imposta come testo della quote)

A differenza della cattura dell'ISBN, qui non si va ad impostare un Analyzer, ma un ImageCapture. Infatti, mentre nella cattura dell'ISBN è più comodo analizzare continuamente le immagini finché in esse non si individua un Barcode valido, qui non ha senso scannerizzare continuamente il testo, perché la citazione che si vuole catturare è in un'area ben specifica che solo l'utente sa dove si trova.

La cattura dell'immagine viene attivata alla pressione del bottone di cattura. Quando questo viene cliccato, vengono aperti (o creati se non esistenti) due files:

- inputFile
- outputFile

Input file è utilizzato come file all'interno del quale viene salvata l'immagine catturata.

Se si verifica un errore nel salvataggio dell'immagine, allora si mostra un AlertDialog.

Se invece il salvataggio va a buon fine, allora si costruisce un'istanza di UCrop. UCrop vuole due Uri per la costruzione

- Uri del file da cui prendere l'immagine da ritagliare
- Uri del file in cui salvare l'immagine ritagliata

Una volta costruita l'istanza, si fa la launch dell'activity di Ucrop, tramite il launcher definito alla creazione del fragment (registerForUCropResult) per ricevere l'immagine ritagliata.

Quando si riceve il risultato dell'activity, se tutto è andato a buon fine, allora l'activity ritorna la Uri del file in cui l'immagine tagliata è stata salvata. Questa Uri viene passata al metodo del VM che scansiona il testo. Se si è verificato un errore si mostra un dialog per comunicarlo.

QuoteWithCameraViewModel

Attraverso la Uri del file si ricava l'immagine in esso salvata. Si prende poi un'istanza di TextRecognition e si esegue l'analisi dell'immagine. Se questa va a buon fine, allora si parte da una *scannedQuote* vuota; dopodiché si cicla su blocchi, linee ed elementi e si aggiungo alla *scannedQuote* i singoli elementi separati da spazi: questo permette di eliminare gli a capo o la separazione tra capoversi e rendere la citazione scannerizzata più compatta. Per finire si emette il valore di *canExitWithText* come la quote appena scannerizzata.

Charts

Charts

ChartsFragment

È il fragment in cui vengono visualizzati i grafici con le statistiche di lettura. Poiché ci sono due tipi di statistiche, totali e per anno, per fornire una User Experience più confortevole si è pensato di utilizzare un ViewPager2 in cui mostrare due fragment che gestiscono ognuno un tipo di statistica. Questo fragment quindi gestisce il caricamento dei dati necessari agli altri due fragment per fare i loro calcoli e l'alternanza nel ViewPager.

In questo caso viene disabilitato lo Swipe del ViewPager: il motivo di questa scelta è stato dovuto al fatto che lo swipe del ViewPager entra in conflitto con lo spostamento dell'asse x dei grafici della libreria MpAndroidChart.

Vengono caricati i dati per i grafici. Questo fragment non ha un observer per l'array di dati caricati: questi observer sono invece presenti negli altri due fragments che vanno effettivamente ad utilizzare questi dati.

ChartsViewModel

È il ViewModel che fornisce ai fragment dei Charts i dati da visualizzare.

I LiveData sono:

- currentYearList. Lista degli anni per ChartsYearFragment: viene utilizzata per impostare un adapter di Spinner per permettere all'utente di scegliere l'anno
- isAccessing
- totalChartData. Dati totali che vengono utilizzati da ChartsTotalFragment
- ChartsYearInfo. Dati per anno che vengono utilizzati da ChartsYearFragment
- readyArray. Booleando che serve a comunicare a ChartsYearFragment e ChartsTotalFragment quando i dati sono stati caricati dal DB e quindi possono essere richieste le elaborazioni.

Vengono poi istanziati due attributi che sono:

- chartsTotalModel. Gestisce le informazioni totali
- chartsYearModel. Gestisce le informazioni relative all'anno

getAllChartsData

Quando ChartsFragment si attiva invoca subito getAllChartsData che carica dal DB i dati per eseguire i vari calcoli.

changeLoadedStatus

Viene invocato da un fragment che cambia la lista dei libri Letti.

getChartsTotalInfo

Viene invocato da ChartsTotalFragment per calcolare le informazioni totali.

Viene inizializzata chartsTotalModel con costruttore currentChartsDataArray (ovvero l'array di dati caricati dal DB) e invocato il metodo di computazione delle informazioni totali. Fatto questo si estrae il valore.

getYearList

Metodo invocato da ChartsYearFragment per calcolare la lista degli anni. Qualora la lista sia vuota, allora si emette una lista placeholder contenente solo uno 0.

Se la lista non è vuota si inizilizza chartsYearModel passando al costruttore l'array delle informazioni dei charts e la lista degli anni.

changeSelectedYear e computeChartsYearInfo

changeSelectedYear viene invocato da ChartsFragmentYear per calcolare le informazioni relative ad un certo anno. Internament il metodo invoca computeYearInfo che invoca un metodo di chartsYearModel per poi estrarne il valore.

ChartsModel
I dati per computare le informazioni da mostrare nei chart vengono caricati nella struttura ChartsBookData. L'array che viene caricato viene, per convenzione, ordinato in base alla endDate dei libri.

ChartsTotal
ChartsTotalFragment
È il fragment che mostra le statistiche totali, ovvero le statistiche basate su dati a partire da quando l'utente ha cominciato ad utilizzare l'applicazione.

Viene chiesta l'istanza di ChartsViewModel e si impostano gli observer a:

- readyDataObserver. Quando ricevuto si ha la certezza che i dati sono pronti e quindi si può invocare il calcolo su di essi.
- chartsTotalInfoObserver. È l'observer che riceve i dati da utilizzare per riempire i grafici. Il valore ricevuto è una classe TotalChartData all'interno di cui vengono scritti i dati da visualizzare in seguito ai calcoli fatti. Utilizzare una classe permette di rendere più snello il passaggio dei dati finali.

Ricevuta la classe TotalChartData viene costruito un PieChart con i supporti e visualizzato il numero di libri e pagine lette.

Un piccolo accorgimento che viene fatto in questo fragment, ma anche in ChartsYearFragment, è relativo al colore del testo del grafico. Di base il colore del testo dei Charts di MpAndroidCharts è nero; questo colore però crea dei problemi nella modalità notturna in quanto si confonde con lo schermo. Si voleva quindi un colore dinamico, che cambiasse insieme con la modalità: poiché la EditText ha questo comportamento, con un testo bianco in modalità notte e nero in modalità giorno, si è impostato il colore del testo al *currentColor* di una EditText.

ChartsTotalModel
Abbiamo l'attributo totalChartData che è istanza di TotalChartData: è la classe con cui si ritornano i vari dati relativi al totale; questo attributo viene ritornato con la funzione getValue.

La funzione più importante è totalChartComputeInfo che calcola proprio le varie informazioni del Totale.

ChartsYear
ChartsYearFragment
Fragment che permette di visualizzare le statistiche relative ad un singolo anno.

Per visualizzare le statistiche relative al singolo anno, si è pensato di utilizzare due spinner per :

- selezionare l'anno
- selezionare il tipo di grafico che si vuole vedere per quell'anno. Questo spinner è stato introdotto per evitare di appesantire eccessivamente il fragment con un numero eccessivo di grafici tutti insieme. Quando un'elemento di questo spinner è selezionato vengono mostrati solo i grafici del tipo selezionato, mentre gli altri vengono fatti sparire.

Viene quindi impostato l'adapter per lo spinner del tipo di grafico e impostati gli observers:

- readyDataObserver. Per notificare il fatto che i dati sono pronti e quindi si può computare su di essi. Quando ricevuto un valore true, viene invocato il calcolo della lista degli anni.
- currentYearListObserver. Viene utilizzato per ricevere la lista aggiornata degli anni. Quando ricevuto l'array, viene usato per custruire l'adapter per lo spinner di selezione dell'anno. Quando è impostato lo spinner, questo attiva onItemSelected che invoca a sua volta la cangeSelectedYear del VM.
- currentChartsYearInfoObserver. Viene utilizzato per ricevere un'istanza della classe ChartsYearInfo che contiene le informazioni necessarie per costruire i grafici. Quando ricevuto vengono invocate una serie di funzioni per costruire i grafici e il valore dell'array di libri relativi all'anno selezionato viene salvato in una variabile (vedere in seguito per il motivo)

Ho poi le varie funzioni invocate per costruire i charts: nei charts in cui è specificato il mese, il nome abbreviato del mese viene messo sull'asse x per facilitare la lettura; per tutti i grafici è disabilitato lo zoom, ma non lo scorrimento su asse x, che è utile quando si devono mostrare i dati relativi ai dodici mesi. Gli unici charts per i quali viene impostata la reazione al click sono i charts relativi ai voti.

La reazione che viene impostata è il mostrare un piccolo Snackbar. Quando viene cliccato un punto del grafico, punto che è relativo al rate di un libro, la Snackbar mostra il titolo del libro. Le informazioni relative al libro vengono prese nell'array di ChartsBookData che viene salvato quando si riceve il valore di ChartsYearInfo. In particolare il libro selezionato è quello che si trova in posizione x dell'array, dove x è l'ascissa del punto del grafico selezionato (questo per come il grafico è costruito). Lo snackbar presenta anche un piccolo bottone per andare alla pagina dei dettagli del libro. Nella action che si passa alla navigate del navController abbiamo, oltre che il bookId del libro da visualizzare, anche un intero: questo intero viene utilizzato per

indicare che si sta entrando nella EndedDetailFragment non da EndedList, ma dai Charts. Questo intero è utilizzato nella EndedDetail per distinguere il caso in cui si entra dai charts: in questo caso infatti sono disabilitate le opzioni del menu che permettono la modifica del Book. Questo viene fatto per evitare complicazioni nella notifica ai charts di un'eventuale modifica dei dati.

Per quanto riguarda gli spinner, se cambia il valore di:

- anno. Si invoca una funzione del VM che calcola i dati per l'anno selezionato
- tipo di grafico. Si modificano le visibilità dei vari layout inclusi e in cui sono contenuti i grafici: I grafici sono stati gestiti costruendo dei layout a parte e inserendoli tramite delle include all'interno del Layout

ChartsYearInfoModel

È il model che gestisce le informazioni per il ChartsYearFragment.

Prende nel costruttore le informazioni totali e la lista degli anni. Nella init viene inizializzata una map che mappa un anno su un'istanza di ChartsYearInfo che contiene le informazioni per quell'anno. Questa mappa viene utilizzata per ritornare le informazioni di un anno direttamente senza ricalcolarle qualora queste siano già disponibili; all'inizio tutte le informazioni relative sono null.

computeChartsYearInfo, computePagesPerMonthArray

La comuteChartsYearInfo invoca varie funzioni per il calcolo dei dati relativi ad un anno e solo se questo anno non presenta informazioni che sono state calcolate in precedenza

In generale un libro si considera appartenente ad un certo anno o mese se terminato in quell'anno o mese: questo è fatto per evitare di andare a calcolare percentuali sulla distribuzione del singolo libro nel tempo.

Per le pagine, invece, si è preferito fare un lavoro più preciso. In particolare le pagine vengono distribuite equamente in tutto il tempo di lettura, dalla data di inizio a quella di fine. Per ogni libro dell'array di ChartsBookData, si vede se l'anno selezionato ricade nell'intervallo tra l'anno di inizio e l'anno di fine del libro, ovvero se il libro è stato letto durante quell'anno. Se ricade, allora si calcola il numero di giorni di lettura totali del libro e il numero di pagine al giorno lette, facendo la divisione: ovviamente questa è solo una stima di massima in quanto non è assolutamente detto che si legga lo stesso numero di pagine ogni giorno.

Se l'anno di inizio e di fine del libro coincidono con quello selezionato, allora il libro è stato letto tutto nello stesso anno. Se il mese di inizio e fine è lo stesso, allora si

aggiungono tutte le pagine del libro al mese corrispondente; se invece è stato letto nello stesso anno, ma tra mesi diversi allora ripartisco le pagine tra i mesi.

Se il libro è stato letto in anni diversi e quello selezionato è quello di inizio o di fine; allora si ripartiscono equamente le pagine tra i mesi intermedi, mentre per i mesi di inizio e fine si ripartiscono in funzione del numero di giorni del mese in cui il libro è stato letto.

Se infine l'anno selezionato è uno intermedio tra quello di inizio e di fine, allora semplicemente ripartisco le pagine per tutti i mesi dell'anno

QuoteOfTheDay

Implementa il widget per la citazione del giorno.

Per implementare il widget si è scelto di utilizzare, all'interno del widget stesso, una list view. Il motivo di ciò è che si vuole un widget che permetta lo scroll: citazioni diverse infatti possono avere lunghezze diverse, e quindi impostare semplicemente una TextView che si adatta al contenuto non basta, in quanto la dimensione del widget non si adatta alla dimensione della TextView. Allo stesso modo non si può impostare semplicemente una TextView che permetta lo scroll, in quanto il widget non lo permette. Quindi si è scelto di utilizzare una ListView (intrinsecamente scrollabile) con all'interno un unico elemento che contiene proprio le informazioni della citazione.

Per aggiornare il widget si costruisce un intent in cui si dichiara il RemoteViewsService che fornisce le View da inserire nella ListView.

La RemoteViewsService ha un ruolo simile a quello dell'adapter, ovvero ritornare la View all'intero della quale sono state caricate le informazioni da visualizzare. Essendo un service, esso va dichiarato all'interno del Manifest. La QuoteOfTheDayWidgetService quindi ritorna un'istanza di QuoteOfTheDayWidgetFactory che va a inserire le informazioni nell'item della ListView.

Il metodo più importante della QuoteOfTheDayWidgetFactory è *onDataSetChanged*. Da documentazione, questo metodo permette di eseguire anche delle operazioni lunghe al suo interno, in quanto lo stato del widget non cambia finché questo metodo non termina. In questo caso specifico, sebbene per visualizzare il contenuto di un widget sia consigliato utilizzare un ContentProvider, si è scelto di non utilizzare un Provider (che è stato comunque implementato). Il motivo è che si ha necessità di accedere al DB in maniera sincrona per fare in modo che, quando la quote (o un cursore che la contiene) è stato caricato, poi venga chiamata la getViewAt che imposta le infomazioni della Quote; non si può accedere al DB senza cambiare

contesto nel ContentProvider, perché Android non permette l'accesso a DB da MainThread, ma allo stesso tempo non posso lanciare una Coroutine, perché questa non mi garantisce sincronia. Poiché il metodo *onDataSetChanged* permette di eseguire lavori lunghi al suo interno (come, appunto, accesso a DB) si accede senza intermediazione del Provider.

In seguito alla chiamata della *onDataSetChanged*, viene chiamata la *getViewAt*, la quale non fa altro che inserire i dati della quote nell'item della ListView; qualora non sia ritornata nessuna Quote (caso in cui non ne siano mai state salvate), si imposta una citazione place holder

Inoltre l'utente ha la possibilità di cambiare lo stato di preferito di una citazione: questo viene fatto inserendo tra gli intent-filters accettati dal Widget una action custom, che permette di rispondere al click sulla citazione. Nella onUpdate in particolare si imposta un pending intent per tutti gli items della ListView; nella getViewAt invece, se la Citazione non è nulla, si imposta un Intent con questa action custom che viene lanciato al click della citazione; al click della citazione quindi verrà invocata la onReceive del widget che cambia lo stato di favorite della citazione e invia un Toast e una Notifica per comunicare all'utente l'avvenuto cambiamento.

GoogleDrive

Per permettere all'utente di salvare i dati relativi alle sue letture in un luogo sicuro oltre al semplice dispositivo mobile, si è deciso di implementare un backup dei dati su Google Drive.

Android fornisce già di base un backup automatico dei dati delle applicazioni su Drive: si parla di AutoBackup. Questo però presenta alcuni limiti:

- non si può andare oltre ai 25 MB di dati per applicazione; limite che, sebbene non troppo stringente visto che 25 MB sono considerevoli, si preferisce non avere
- viene eseguito solo quando il dispositivo è in certe condizioni, che non è detto che si verifichino sempre
    - connesso a Wi-Fi
    - connesso alimentatore
    - backup abilitato

Si è deciso quindi di implementare un Backup che non avesse limiti e che l'utente potesse gestire a sua discrezione.

Librerie

Sono state aggiunte alle librerie esterne importate le seguenti

```
implementation 'com.google.guava:listenablefuture:9999.0-empty-to-avoid-
conflict-with-guava'

implementation('com.google.api-client:google-api-client-android:1.23.0')
implementation('com.google.apis:google-api-services-drive:v3-rev136-1.25.0')
implementation 'com.google.android.gms:play-services-auth:19.2.0'
```

- la prima libreria ha lo scopo di evitare un conflitto tra altre librerie (libreria guava)
- seconda e terza hanno lo scopo di connettersi a GoogleDrive per scambiare dati
- La quarta serve per implementare l'autenticazione all'account Google dell'utente

È stato poi introdotto, anche qui per evitare conflitti tra pacchetti, il seguente blocco nella parte Android del Gradle

```
packagingOptions {
    exclude 'META-INF/DEPENDENCIES'
    exclude 'META-INF/LICENSE'
    exclude 'META-INF/LICENSE.txt'
    exclude 'META-INF/license.txt'
    exclude 'META-INF/NOTICE'
    exclude 'META-INF/NOTICE.txt'
    exclude 'META-INF/notice.txt'
    exclude 'META-INF/ASL2.0'
    exclude("META-INF/*.kotlin_module")
}
```

Autenticazione

Per implementare l'autenticazione dell'utente si è seguita la guida ufficiale di autenticazione di Android:

https://developers.google.com/identity/sign-in/android/start-integrating

Si impostano quindi le opzioni di autenticazione e si impostano gli Scope a cui si vuole accedere (vedere più avanti per gli scopes). Dopodiché si avvia un'activity per chiedere il profilo con cui ci si autentica.

Per eseguire la Sign Out si ha un'implementazione più o meno speculare. Se l'activity restitisce un risultato valido, allora si invoca un metodo del VM, altrimenti si mostra un AlertDialog con comunicazione di errore.

Comunicazione con Google Drive

Per poter comunicare con Google Drive è necessario prima di tutto iscriversi alla *Google Cloud Platform.* Dopo averlo fatto si deve creare un nuovo pegetto.

Dopo averlo fatto, si deve andare nella sezione “Abilita API e Servizi” e selezionare i servizi a cui si vuole accedere (nel nostro caso, Google Drive).

Abilitate le API di interesse si devono creare delle credenziali. Per creare delle credenziali si deve indicare:

- Tipo di API a cui si chiede accesso
- Tipi di dati a cui si vuole accedere: nel nostro caso si tratta dei dati della singola applicazione

Si procede poi inserendo li altri dati richiesti.

Dopo aver creato le credenziali si deve creare un ID client OAuth: Google Drive infatti prevede accesso solo tramite OAuth2. Nella crezione dell’ID vengono chieste due cose principali:

- Scopes. Gli Scopes indicano a cosa nello specifico l’applicazione per cui si sta creando questo ID vuole accedere. Come indicato nella documentazione (https://developers.google.com/drive/api/v3/about-auth) ci sono diversi tipi di scope: quello che interessa a noi è *drive.appdata*: I vantaggi di questo scope sono che
    - I dati sono salvati in una cartella di drive non accessibile direttamente all’utente, che quindi non può modificare i dati che vi vengono inseriti
    - Fa parte della categoria degli scopes Recommended, che quindi richiedono il minimo grado di autorizzazione per accedere.
- Chiave SHA-1. La chiave SHA-1 identifica l’applicazione in modo univoco agli occhi di Google Drive, fungendo da certificato. Per generarla è sufficiente andare nella sezione Gradle di Android Studio (alto a destra) e fare click su *signingReport*.

A questo punto si può eseguire il Backup su Drive.

GoogleDriveFragment

All’interno del fragment troviamo quattro bottoni:

- Login / logout
- Upload / Download del Backup

Si invoca subito il metodo getUser() del VM per verificare se vi sono degli utenti già autenticati.

Si impostano quindi gli observers:

- isAccessing
- currentUserObserver. Il valore viene ricevuto quando un'utente si autentica a con il suo account o quando fa logout. Qualora lo user sia
  - null. Si abilita solo il button di login
  - != null. Si abilitano tutti gli altri bottoni
- operationCompletedObserver. Il valore viene ricevuto al termine di una operazione di Upload o di Download con all'interno la stringa che indica l'esito dell'operazione. Questo valore viene utilizzato come titolo di un AlertDialog per comunicare all'utente il risultato dell'operazione. La *resetMessage* viene invocata per resettare il valore del titolo in modo da non ricevere valori ripetuti.

I bottoni di Download e di Upload chiamano entrambi metodi del VM.

GoogleDriveViewModel

DriveStart

È la classe a partire da cui si prende un'istanza di *Drive* con cui poi si può accedere effettivamente al servizio.

Il suo metodo *getDriveService* viene invocato all'interno della *getUser* del VM nel caso in cui l'utente non si null.

Il builder del Drive vuole in particolare un'istanza di credenziali, le quali vengono acquisite con il metodo *getCredentials*.

ViewModel

Dopo aver ottenuto l'istanza di Drive è possibile eseguire le operazioni di Upload e di Download che vengono gestite da GoogleDriveModel.

uploadBackup

Per l'upload del Backup si procede nel seguente modo. Si creano prima di tutto due JSONArray attraverso le funzioni *createJsonBookArray* e *createJsonQuoteArray*; si crea quindi un JSONObject all'interno del quale si inseriscono i due array con chiave Book e Quote.

Creato l’array esso viene scritto all’interno di un file: questo è il file che verrà caricato su Drive.

Si verifica poi l’esistenza di un backup precedente e nel caso si ritorna il suo id: l’id è una stringa univoca che viene assegnata dal Drive all’atto del caricamento del File.

Si crea quindi un nuovo Google File di cui si impostano come campi nome e parent (appDataFolder).

Per finire se un backup esisteva, questo si cancella per sostituirlo con il nuovo e poi si carica il nuovo Google File con contenuto quello del file in cui è stato caricato il JSONObject.

Per finire si elimina la copia locale del DB per evitare sprechi di memoria interna

downloadBackup

La procedura di Download è speculare a quella di Upload. Si verifica prima di tutto l’esistenza del file di backup in Drive: se esso esiste allora si procede al download.

Per il download bisogna specificare l’Alt a “media” per poter scaricare il contenuto del file.

Scaricato il contenuto del file, esso viene convertito in un JSONObject e poi sugli array interni con chiavi Book e Quote vengono invocate due funzioni che ciclano, convertendo il JSONObject in un Book (o Quote) per poi inserirlo nel DB.

What’s Next

Nei prossimi aggiornamenti dell’applicazione vorrei aggiungere una serie di funzionalità tra cui:

- Guess The Quote. Un giochino per accoppiare a citazione a libro/autore
- Catalogazione dei libri tramite etichette di vario tipo (sia per stile e genere, sia per tematiche trattate) anche attraverso le etichette date dalle API di Google Books
- Costruzioni di Mappe Mentali a partire dalle etichette, e quindi costruzione di mappe che legano i libri tra loro in base, ad esempio, alle tematiche
-