Marvel Heroes

Mobile Programming 2020 – Università degli Studi di Roma “Tor Vergata”



# Realizzato da Tiburtina Valley

0253055 Falcone Gian Marco

0242178 D'Orazio Antonio

0252255 Fanali Francesco

0253186 Minut Robert Adrian

0252345 Caliandro Pierciro

Indice

Introduzione

1. Descrizione del progetto
2. Strategia di progetto e sviluppo
3. Manifest
4. Descrizione activities

# Progettazione e scelte implementative

1. RecyclerView
2. Adapter
3. Caching con Shared Preferences
4. Volley
5. Menu e pagina about us
6. Bundle, Parcelable e Intent
7. Glide
8. Menu contestuale
9. SectionedRecyclerView
10. Permessi, controlli e Toast
11. Loading
12. Localizzazione e stringhe
13. Fragments
14. Animazioni
15. Room

Introduzione

# Descrizione del progetto

Il progetto sviluppato dal nostro gruppo riguarda il mondo Marvel.

Scopo del progetto è stato lo sviluppo un’applicazione in grado di sfruttare al massimo le potenzialità offerte dal portale sviluppatori *developer.marvel.com*.

Questo portale fornisce una chiave di accesso ai servizi Marvel per sviluppatori, consentendo l’accesso ai loro dati e permettendo quindi all’applicazione di ricavare, attraverso richieste, tutte le informazioni necessarie al suo funzionamento.

Le query implementate permettono di ricercare:

* un eroe Marvel (sia tramite il suo nome intero, che per iniziale del nome);
* un fumetto in cui è presente un eroe;
* una serie in cui è presente un eroe;
* un evento relativo ad un eroe;
* gli URL del sito web Marvel che riguardano un certo eroe;
* eroi, autori, fumetti ed eventi correlati a partire da una serie;
* eroi, autori, fumetti e serie correlati a partire da un evento;
* eroi, autori, serie ed eventi correlati a partire da un fumetto;

La maggior parte di queste query sono state utilizzate per definire pagine strutturate dove vengono riportate tutte queste informazioni (attraverso un insieme di RecyclerView), in modo da essere logicamente correlate tra loro. Alcune query non sono state utilizzate poiché fornivano informazioni parziali e spesso incomplete.

Le funzioni implementate sono:

* la ricerca di un eroe;
* la navigazione fra tutte le caratteristiche dell’eroe cercato, offrendo dei risultati correlati ad esso per approfondimenti;
* la sezione preferiti, che permette all’utente di salvare ed eliminare con i suoi eroi preferiti, salvandoli localmente nell’archiviazione del dispositivo, garantendo un accesso rapido
* le schermate di dettaglio per ogni elemento relativo agli eroi che è possibile trovare nel sito, come fumetti, eventi, serie ed autori;
* la schermata home, che ogni ventiquattro ore si aggiorna automaticamente mostrando all’utente quelli che sono gli eroi, i fumetti e le serie che la nostra applicazione consiglia di giorno in giorno;

Da qualsiasi scenario, l’utente può tornare al menu principale dell’applicazione semplicemente cliccando sul pulsante “X” di uscita presente in ogni schermata.

Il risultato finale è stato come da specifica un’applicazione ben strutturata, sia a livello di codice che di esperienza utente, che presenta tutte le informazioni messe a disposizione dal sito, piacevolmente fruibili.

Il tutto è stato gestito cercando di garantire la massima efficienza possibile, utilizzando strumenti implementativi come *cache* (utilizzate per salvare i dati correnti dell’applicazione) e *thread* che operano in parallelo e permettono all’applicazione di essere leggera e di non bloccarsi.

Infine, abbiamo curato la grafica nei minimi dettagli, implementando gradevoli effetti visivi (come quelli di scorrimento e selezione) e una schermata di caricamento, per fare da transitorio tra le varie schermate.

# Strategia di progetto e sviluppo

La progettazione e lo sviluppo dell’applicazione si sono svolti in due macro-fasi: nella prima ci siamo soffermati maggiormente sulla parte di back-end, andando a creare un’interfaccia grafica minimale che avesse come unico scopo quello di mostrare i dati per gli eventuali test.

Nella seconda fase invece, è stato curato il font-end, andando a ritoccare l’interfaccia con l’aggiunta di effetti grafici ed animazioni, uniformando i colori e personalizzando il font, andando ad aggiungere infine dettagli come la ProgressBar e la ToolBar.

La chiave di volta di tutta la fase di progettazione e sviluppo è stata la scelta di adottare una metodologia di sviluppo agile: infatti, è stata prodotta in tempi rapidi (nel giro di qualche settimana) una prima versione semplificata dell’interfaccia, ma che presentava tutte le funzionalità offerte, in modo che fossero ben definite e funzionanti.

Successivamente sono state compiute una serie di iterazioni cicliche sul codice che hanno permesso di implementare tutte le funzionalità richieste nel modo più efficiente possibile.

Terminata così la fase di progettazione e sviluppo del back-end, si è passati ad una progettazione più accurate del front-end e di tutto ciò che riguarda gli aspetti legati alla grafica.

Anche in questo caso si è seguita una metodologia di tipo agile e ogni schermata è stata man mano arricchita di dettagli, andando a reiterare più volte per non tralasciare nulla.

Andando nello specifico, per la parte di back-end, è stato creato un primo scheletro dell’applicazione, che permetteva di effettuare la ricerca di un eroe e di visualizzare I dettagli di un eroe selezionato, secondo un approccio di sviluppo tipicamente top-down.

Dopo di che, sono stati aggiunti tutti gli elementi correlati all’eroe, quali fumetti, serie, eventi ed autori e, per ognuno di essi, è stato sviluppato un primo prototipo di schermata che lo mettesse in relazione tutti gli altri elementi citati.

Infine, dopo aver sviluppato e collegato fra loro tutti questi elementi, si è proceduto a perfezionare i singoli aspetti che li caratterizzavano, tenendo a mente come l’obiettivo da raggiungere fosse garantire la migliore navigabilità dell’applicazione e la migliore esperienza utente possibile.

# Manifest

La nostra applicazione è identificata dall’id univoco “it.tiburtinavalley.marvelheroes”.

L’hardware necessario per eseguire la nostra applicazione è relativamente molto economico. È sufficiente un dispositivo dotato di connessione ad Internet che abbia uno schermo di dimensione adeguate. Allo scopo di implementare quest’ultimo requisito, abbiamo impostato nel manifest, nell’impostazione <supports-screens>, l’attributo android:smallScreens a false.

Uno schermo piccolo è definito come uno schermo con un rapporto d'aspetto inferiore rispetto allo schermo "normale" (HVGA tradizionale, ossia 320x480). In questo modo non sarà disponibile per dispositivi a schermo piccolo la possibilità di installare l’applicazione da servizi esterni (come Google Play).

Nonostante ciò è comunque possibile procedere all’installazione scaricando direttamente l’apk o tramite Android Studio. In tal caso l’applicazione si limiterà a segnalare la possibile incompatibilità all’utente, ma potrà comunque essere utilizzata.

Per quanto riguarda invece il software, è necessario disporre di una versione del sistema operativo Android maggiore o uguale alla 8.0.

La nostra applicazione è sicura, e l’unico permesso che richiede è l’accesso alla rete.

Evidenziamo infine che tutta la nostra applicazione è stata sviluppata tenendo conto del fatto di dover funzionare sia in modalità portrait che landscape, fatta eccezione per la Main Activity, in cui il landscape è stato bloccato nel manifest.xml poiché comprometterebbe l’uso della toolbar.

# Descrizione activities

* **MainActivity**

Activity principale dell’applicazione, eseguita automaticamente.

È dotata di una barra di navigazione posta nella parte inferiore dello schermo per navigare tra le diverse funzionalità.

Consente di spostarsi fra la home contenente i suggeriti del giorno, la ricerca e i preferiti.

Tramite il menu superiore invece è possibile raggiungere l’About activity.

* **HeroDetailsActivity e FavoriteHeroDetailsActivity**

Activities che permettono e gestiscono la navigazione dei dati relativi a un personaggio (tra cui immagine e descrizione) e di tutti gli elementi a lui correlati (serie, fumetti ed eventi).

* **DetailsActivity di eventi, serie, fumetti e autori**

Activities con tutti i dati relativi a fumetti, serie, eventi e autori.

Notare che queste activity sono tutte legate tra loro, e che perciò consentono una navigazione dei contenuti a 360 gradi.

* **AboutActivity**

Pagina dedicata al team, con tutte le informazioni ed i contatti disponibili.

Implementata tramite uso dei menù, permette all’utente di informarsi sul team di sviluppo e di contattarci.

Progettazione e scelte implementative

# RecyclerView

Le RecyclerView sono uno strumento molto utile quando c’è bisogno di organizzare una lista di informazioni secondo uno schema preciso. Nell’applicazione, ogni Model di tipo Hero, Comics, Series, Events o Creators ha altre liste di Model a lui collegate (es i fumetti, le serie e gli eventi collegati ad un eroe) che vanno quindi mostrati all’utente.

Ogni RecyclerView è un contenitore di singole View, che possono avere uno schema più o meno complesso, in questo caso le View hanno una struttura molto semplice e sono composte da un ImageView ed una TextView. Le View devono essere fornite da un LayoutManager, che nel progetto è quasi sempre un LinearLayoutManager con opzione di scorrimento orizzontale.

Inoltre, ogni elemento della RecyclerView viene gestito da un ViewHolder, i cui metodi vengono chiamati proprio dal LayoutManager quando è necessario creare una nuova riga nella RecyclerView o bisogna aggiornarne il contenuto.

Le liste da visualizzare sono di dimensione variabile, andando da 20/30 elementi fino anche a più di 50. La RecyclerView mantiene tutti gli elementi che non possono essere mostrati per questioni di spazio nel display in una apposita cache, così da andare a caricare i nuovi quando viene eseguito uno scroll per visualizzare gli altri elementi, o ricaricare i vecchi quando si vuole tornare verso l’inizio della lista. In questo modo, la gestione della memoria è più efficiente e la lista è completamente gestita dalla RecyclerView.

Le RecyclerView sono state introdotte pressoché in ogni Activity, specialmente in quelle che mostravano il dettaglio di uno specifico Model, quindi la HeroDetailActivity, ComicsActivity, SeriesActivity, EventsActivity e CreatorsActivity.

# Adapter

Ogni elemento della RecyclerView è rappresentato mediante una View indipendente, con un preciso layout che deve essere gestito mediante appositi ViewHolder.

Ogni ViewHolder è a sua volta gestito da un Adapter, che svolge la funzione di ponte fra la UI e i ViewHolder.

L’Adapter svolge due compiti fondamentali nella gestione dei ViewHolder: assegna ad ognuno di essi una specifica posizione nella RecyclerView e vi collega i dati che dovranno essere gestiti.

Ogni Adapter gestisce una lista di Model, i cui dati saranno poi inseriti nelle View gestite dai ViewHolder.

Abbiamo scelto di avere un’apposita classe Holder per ogni Adapter, che estende la classe Recycler.ViewHolder e gestisce quindi gli elementi del layout. In questo modo, il codice è molto più mantenibile e leggibile.

Ogni volta che la RecyclerView ha bisogno di mostrare un nuovo elemento, viene invocato il metodo *onCreateViewHolder* dell’Adapter, che crea per l’appunto un nuovo oggetto di tipo Holder per la gestione della View.

Il metodo di *onBindViewHolder* nell’Adapter viene invece chiamato quando c’è bisogno di aggiornare il contenuto di una View, tale metodo ha come parametro l’Holder di cui va aggiornata la View e la posizione di quest’ultimo nella RecyclerView.

Infine, vi è un metodo *getItemCount* che ritorna la lunghezza della lista di Model che viene gestita dall’Adapter.

Ognuno degli elementi nella RecyclerView è inoltre cliccabile, infatti ogni Adapter implementa il metodo di *onClick*, andando a caricare la schermata di dettaglio per l’elemento selezionato dall’utente.

# Caching con Shared Preferences

Per poter mantenere sempre identici i suggeriti del giorno, è stato necessario aggiungere un meccanismo di salvataggio su file, per fare questo la scelta è ricaduta sulle Shared Preferences.

Dovendo salvare tre Model, ovvero un eroe, un fumetto ed una serie, costituiti da varie informazioni, i Model vengono convertiti in un Json tramite l’API Gson, per poi essere riconvertiti ogni volta che viene caricata la Home Page.

Abbiamo usato il metodo *getPreferences* per ottenere il file su cui salvare i Json,in quanto è necessario un solo file per contenere tutto e tale file viene resettato ogni giorno con i nuovi suggeriti del giorno.

Per il salvataggio è necessario usare un oggetto di tipo SharedPreferences.Editor, i metodi di scrittura possibili sono due: *commit*, che è un metodo sincrono con valore di ritorno di tipo booleano per indicare se la scrittura è andato a buon fine ed il metodo *apply*, che viene invece eseguito su un thread e non riscontra al main se la scrittura è avvenuta con successo.

Abbiamo deciso di utilizzare il metodo di *apply*, in quanto il controllo sulla presenza del dato nel file di SharedPreferences viene effettuato solo quando l’utente ritorna sulla Home Page in un secondo momento (così da verificare anche che i dati non sono stati cancellati dall’esterno), inoltre in questo modo il salvataggio su file non pesa sull’ Home Page Fragment, non essendo una chiamata sincrona.

Il file di SharedPreferences viene resettato ogni giorno tramite il metodo *clear* e ripopolato con i nuovi suggeriti del giorno.

# Volley

La nostra applicazione è stata sviluppata con lo scopo di sfruttare al massimo le potenzialità messe a disposizione dal portale web Marvel developer, pertanto l’interazione con questo, mediante l’uso di Internet, è fondamentale per quasi ogni azione svolta dalla nostra app. È stato necessario gestire queste interazioni con la rete mediante l’utilizzo di thread, in modo da non appesantire il main thread e non causare di conseguenza blocchi dell’interfaccia grafica. Per eseguire queste operazioni, a livello implementativo, abbiamo optato per l’utilizzo di una ben nota libreria Android che adempie a tali scopi: Volley.

Volley infatti permette di gestire in modo molto semplice le richieste http/https, utilizzando per i suoi scopi thread e buffer con strutture a coda per gestire le diverse richieste (solitamente di tipo FIFO). Questa libreria inoltre fa utilizzo di strutture di cache per salvare i risultati delle ricerche e garantire maggior efficienza.

Per utilizzare volley occorre dapprima fare due cose: dichiarare il suo utilizzo nel file build.gradle ed inserire nel manifest.xml i permessi dell’app relativi agli accessi ad internet.

Dopo di che nell’applicazione sono state definite cinque classi dedicate all’uso di Volley, ed utilizzate per gestire richieste atte a reperire tutte le informazioni necessarie, relative rispettivamente a: eroi, fumetti, serie, eventi ed autori.

Queste classi sono state sviluppate seguendo la medesima struttura, e risultano pertanto classi “standardizzate”. La loro definizione infatti si apre con la dichiarazione di un attributo che definisce la coda delle richieste e di un metodo astratto di *fill* che servirà per effettuare un’operazione di callback una volta ottenuti i dati necessari.

Successivamente sono stati dichiarati i metodi che, se invocati, permettono di formulare il corretto url (generalmente prendendo in input l’id di un oggetto di interesse), e di conseguenza effettuare tutti i tipi di richieste necessarie relative alla tipologia della Volley in questione.

Questi metodi, una volta formulato il corretto url, convergono tutti nell’utilizzo di un metodo di *apiCall*, il quale inizializza le richieste e la inserisce nella coda. (ricordiamo che tutte le richieste che formuliamo sono di tipo get).

Troviamo infine le due funzioni che gestiscono la risposta alla richiesta formulata. Nel caso in cui la risposta non sia stata correttamente ricevuta infatti entra in gioco un metodo di *onErrorResponse*, il quale prima cerca di capire se l’errore è dipeso da un malfunzionamento della connessione ad Internet o dall’esaurimento del numero di query giornaliere possibili (segnalandolo eventualmente all’utente mediante un toast), e poi scrive sul log un messaggio di errore.

Nel caso in cui invece la risposta arrivata si riveli corretta, entra in funzione il metodo di *onResponse*, che va a prendere in input il file di tipo Json restituito nella risposta di get. A questo punto il metodo istanzia un oggetto di tipo GSON che, combinato con il meccanismo di reflection messo a disposizione da java, verrà utilizzato per effettuare la conversione di array Json in array di oggetti java (opportunamente definiti nel model per il corretto mapping degli attributi).

Al termine dell’operazione, se la lista non è vuota, viene scritto nel log un messaggio con il numero degli oggetti inseriti nell’array. Invece, in caso di lista vuota, si andrà a segnalare all’utente mediante l’uso di un toast che non stati trovati risultati.

Infine, viene fatta l’operazione di callback, definita all’inizio e relativa al metodo di *fill* appartenente all’oggetto che ha istanziato la Volley e che ne ha invocato le operazioni. Tale metodo utilizzerà i dati scaricati mediante la Volley per popolare le RecyclerView presenti nella user interface.

La classe Volley si chiude così con un catch che cattura eventuali eccezioni legate all’uso del Json.

È importante dire che la nostra scelta di utilizzare Volley è anche dipesa dal fatto che ci siamo ritrovati ad operare con file di piccole dimensioni, se invece avessimo dovuto gestire file di dimensione maggiore, questa non sarebbe stata la scelta più efficiente, poiché questa libreria è ottimizzata per il download di file relativamente piccoli (<100mb). Pertanto, con file di dimensioni maggiori, sarebbe stato più corretto utilizzare il download manager.

# Menu e pagina about us

Abbiamo deciso di inserire all’interno della nostra applicazione un’intera pagina dedicata al team che ha lavorato allo sviluppo della stessa. Si tratta di una pagina molto completa, che presenta dapprima il logo della start up e una breve descrizione del team di sviluppo, e successivamente una serie di voci molto esplicative che permettono di reperire le informazioni relative al nostro team che sono disponibili in rete. L’utente può ad esempio trovare le nostre pagine Facebook, Twitter e Instagram, con le quali è in grado di mettersi in contatto con noi e seguire in tempo reale le nostre attività, ma anche la nostra pagina GitHub, che gli permette di accedere a tutto quel codice che abbiamo deciso di lasciare pubblico. È presente inoltre la possibilità di mettersi in contatto con il nostro team a scopo collaborativo-professionale, tramite la nostra mail certificata. Non abbiamo poi tralasciato nemmeno la pagina relativa alla nostra applicazione sul play store, che permetterà di accedere velocemente a tutte le caratteristiche e recensioni relative alla nostra app, oppure la nostra pagina YouTube, che riporterà video relativi ai lavori svolti presso la nostra start up. Infine, non può mancare una voce per accedere al nostro sito web, la fonte con le informazioni più complete e dettagliate sul nostro team e la nostra impresa. Precisiamo però che, non essendo il nostro gruppo un vero team di sviluppo, e non avendo (almeno per ora) una start-up, abbiamo deciso di non creare realmente per ora tutte le pagine web relative a Tiburtina Valley inc. e presentate nella nostra about page. Tuttavia, tutte le voci sono state effettivamente collegate a link e tutti i bottoni implementati, utilizzando come pagine di riferimento (YouTube, Facebook, Twitter o Instagram che siano) pagine scelte casualmente cercando il nome “tiburtina valley” nei vari siti web. L’unica pagina realmente relativa a noi è quella di GitHub. (inoltre, il link al play store porta ad una pagina vuota, non avendo noi effettuato il deploy della nostra applicazione su di esso).

Tornando invece agli aspetti implementativi, abbiamo deciso di inserire la nostra pagina *about us* come unica voce di un classico menù Android, posizionato come consuetudine in alto a destra nell’applicazione e recante l’immagine dei tipici 3 pallini disposti in verticale. Vogliamo precisare che la nostra pagina *about us* non è stata realizzata mediante l’utilizzo di un file xml, ma invece attraverso una libreria java reperita online, che permette di generare la pagina e tutto il suo contenuto in modo dinamico basandosi unicamente sul codice scritto in java e su xml esterni.

La libreria usata è reperibile su GitHub all’indirizzo: <https://github.com/medyo/android-about-page> e può essere aggiunta ad un applicazione Android semplicemente aggiungendo al file build.gradle la seguente stringa:

implementation ‘com.github.medyo:android-about-page:1.2.5'

Il funzionamento della libreria è molto semplice, tutte le caratteristiche della pagina sono definite e gestite da un *about us* activity in Java. Questa infatti ci permette di associare tutte le voci e i relativi bottoni ai link delle corrispondenti pagine web, di definire la descrizione del nostro gruppo, e di scegliere le immagini da mostrare. Tutti i link e le stringhe sono stati definiti nello strings.xml, compresa la descrizione, che è stata localizzata nelle tre lingue riportate nella corrispondente sezione di questa relazione.

Tornando invece al menù, questo è stato definito attraverso la procedura classica, ovvero mediante la creazione di un file menu.xml presente in una sottocartella menu della directory res. Lo stile è anch’esso molto semplice, si tratta di una classica piccola sezione che compare al click del pulsante di menu (poiché per noi le scelte più semplici, spesso si rivelano anche le più efficaci). Il menu.xml poi è stato collegato all’apposito bottone di menu mediante il suo id. La semplicità di questo menu non ha reso necessario utilizzare le clausole e gli attributi che Android rende disponibili per tali situazioni.

Infine, il menu è stato collegato alla main activity definendo nella stessa un metodo *onCreateOptionsMenu*, che non fa altro che “gonfiare” il menù, rendendo l’icona visibile nella main activity. Infine, abbiamo implementato il click sul menu stesso, andando a definire sempre nella main activity il metodo *onOptionsItemSelected*, che intercetta il click dell’utente, e fa partire la relativa activity di *about us.*

# Bundle, Parcelable e Intent

Dovendo nella nostra applicazione passare da una activity all’altra e data la necessità di portare con noi i dati relativi alle activity precedenti (necessari per il corretto funzionamento delle activity nuove), siamo stati costretti a fare riferimento a due strumenti che Android mette a disposizione: i Bundle le Intent.

Infatti, essendo le activity dei thread (e perciò asincroni), la comunicazione tra essi è possibili unicamente in tale modo.

Un bundle non è altro che una classe in stile HashMap, cioè basata su coppie chiave/valore, dove la chiave è una stringa e il valore è un tipo di dato elementare oppure un oggetto, purché quest’ultimo sia di tipo Parcelable.

Una classe si dice di tipo Parcelable se implementa la relativa interfaccia Parcelable e ne fa l’override dei relativi metodi necessari.

Nei bundle il set dei dati da passare all’activity chiamata viene fatto dall’activity chiamante mediante il metodo di put,mentre il get degli stessi è fatto dall’activity chiamata attraverso il relativo metodo di get.

Activity chiamante e activity chiamata possono corrispondere nel caso in cui ci sia un cambiamento di orientazione dello schermo, poiché ciò corrisponde ad un restart dell’activity stessa e sarà quindi necessario ripristinarne lo stato.

Una Intent è invece definita come la descrizione di una activity da eseguire, ed è necessario in Android per effettuare il lancio di una nuova activity.

Avendo noi appunto più di una activity, abbiamo dovuto scegliere come utilizzare questi strumenti. Nello specifico abbiamo deciso di usare gli Intent anche per adempiere alle funzioni del bundle stesso, salvando i dati da passare al suo interno.

Questa scelta è dipesa da motivi di efficienza, infatti utilizzando un bundle e passandolo ad un Intent non viene fatto altro che un mapping tra la HashMap del bundle verso quella dell’Intent, invece utilizzando solo le Intent (e facendo perciò passaggio diretto dei dati), si evita questa faticosa operazione.

# Glide

Glide è una libreria open-source che si occupa della gestione dei contenuti multimediali, ed in particolare è un framework per il caricamento delle immagini in ambiente Android.

Il suo ruolo principale è quello di rendere trasparente al programmatore il decoding dei media, il caching su disco e memoria principale, e la gestione delle risorse attraverso l’uso di un’interfaccia semplice da utilizzare.

Nella nostra applicazione avevamo bisogno di un meccanismo semplice di caricamento e caching delle immagini offerte per gli eroi, i fumetti, gli autori e le storie dei personaggi, e la nostra scelta è ricaduta su questo particolare framework per la sua semplicità d’uso.

String urlThumbnail = comicOfTheDay

.getThumbnail().getPath().replaceFirst("http", "https")

+ "." +

comicOfTheDay.getThumbnail().getExtension();

Glide.*with*(getActivity())

.setDefaultRequestOptions(requestOptions)

.load(urlThumbnail)

.diskCacheStrategy(DiskCacheStrategy.*ALL*)

.into(ivComic);

Si può osservare dall’esempio in alto che in pochi passi è possibile implementare il prelievo delle risorse multimediali ed il caricamento nei relativi contenitori attraverso poche righe di codice, passando come parametri il *context* dell’*Activity*, il *URL* della risorsa, le modalità di *caching* da utilizzare, il riferimento al contenitore, ed un oggetto con le opzioni che possono essere condivise tra più richieste, che riguardano il ritaglio, lo zoom, e la possibilità di inserire dei *placeholder* nel contenitore prima del completamento della richiesta.

# Menu contestuale

Per permettere all’utente di eliminare dalla lista dei preferiti uno o più eroi salvati, senza dover aprire l’*Activity* di dettaglio, è stato implementato un menù contestuale con la singola voce “elimina”, gestito all’interno del FavouriteFragment.

Per l’attivazione del menù contestuale è stato utilizzato il meccanismo dell’*onLongClick* sugli elementi contenitori *view* degli eroi preferiti, implementato negli *Holder* che sono gestiti dal *FragmentAdapter*.

Ogni volta che viene effettuato un *long press* su uno degli eroi preferiti, appare il menù contestuale dove l’utente può selezionare o deselezionare altri elementi con un semplice tocco ed eliminarli tramite la voce del menù, che si chiude subito dopo l’eliminazione.

L’implementazione si basa sul meccanismo di *callback*, infatti, ogni elemento *view*, associato ad un eroe salvato nei preferiti, fa una chiamata tramite il riferimento *smListener* per iniziare la *ActionMode* ed attivare il menù contestuale. Osserviamo infatti che nella *onLongClick* che viene impostata su tutti i contenitori *view*, abbiamo:

if (smListener != null) {  
 smListener.onSelect(selectedHeroesList.size()); //callback verso l'Activity

Il parametro passato rappresenta il numero di elementi che sono stati selezionati, così da permettere la disattivazione del menù, quando questo numero arriva a 0.

L’attributo *smListener* viene impostato durante l’istanziazione di *FavouriteAdapter*, poiché viene passata l’istanza creata nella *FavouriteFragment*, dove la classe viene definita come:

public class SelectModeListener implements HeroSelectMode {  
 @Override  
 public void onSelect(int size) {  
 if (mActionMode != null) {  
 if (size == 0) {  
 mActionMode.finish();  
 mActionMode = null;  
 }  
 } else  
 mActionMode = ((AppCompatActivity)

Objects.*requireNonNull*(getActivity()))

.startSupportActionMode(hActionModeCallback);  
 }  
}

Si osserva da qui che il ciclo di vita del menù contestuale inizia e termina attraverso le chiamate di *callback* sul metodo *onSelect*. Tuttavia, può terminare anche nel caso in cui venga premuto il pulsante di *back*.

In tal caso, viene direttamente effettuata una chiamata al metodo *onDestroyActionMode* dell’attributo della classe *FavouriteFragment* che ha il nome di *hActionModeCallback*.

Quest’ultimo rappresenta l’istanza della classe astratta *ActionMode.Callback*, e la cui implementazione viene definita sempre all’interno della classe *FavouriteFragment*, che permette di definire le operazioni di creazione e disattivazione della barra in alto e l’eliminazione degli eroi dalla sezione dei Preferiti.

# SectionedRecyclerView

La libreria SectionedRecyclerView permette di dividere facilmente gli elementi nella *RecyclerView*, la classe a cui vengono estese le funzionalità, in sezioni espandibili comprese del relativo *header*.

L’obiettivo era organizzare gli eroi preferiti in modo da facilitare la ricerca del personaggio filtrando per la prima lettera del nome.

I componenti principali offerti dalla libreria sono la classe *SectionedRecyclerViewAdapter*, ed i metodi astratti offerti da essa, che sono *onPlaceSubheaderBetweenItems*, *onBindItemViewHolder* e *onBindSubheaderViewHolder*.

Il primo metodo permette di determinare quando due sezioni vadano separate, e quindi quando è necessario inserire un *subheader* tra due elementi, nel nostro caso bastava confrontare le prime lettere dell’eroe dell’elemento corrente e del successivo e verificare se fossero diverse.

public boolean onPlaceSubheaderBetweenItems(int position) {  
 final char heroNameFirstCharacter = heroes.get(position).getName().charAt(0);  
 final char nextHeroNameFirstCharacter =

heroes.get(position + 1).getName().charAt(0);

return heroNameFirstCharacter != nextHeroNameFirstCharacter;  
}

Il secondo metodo permette di impostare i valori contenuti nell’elemento *view* dell’eroe, mentre il terzo riguarda ciò che deve essere visualizzato all’interno del *subheader*.

# Permessi, controlli e Toast

Naturalmente per interfacciarsi con il database messo a disposizione dal sito della Marvel occorre una continua disponibilità della connessione ad Internet. Questo infatti, insieme alla possibilità di controllare lo stato della connessione, sono stati gli unici permessi che si sono resi necessari nello sviluppare la nostra app.

Essendo entrambi permessi normali, in quanto il rischio per la privacy dell’utente o per il funzionamento delle altre app è minimo, essi vengono automaticamente concessi dal sistema operativo e l’utente non può revocarli. Perciò non è necessario né chiedere all’utente l’autorizzazione esplicita e né tantomeno inserire nel codice controlli sulla loro effettiva autorizzazione da parte dell’utente.

Al fine di verificare tale connessione abbiamo aggiunto un controllo prima di effettuare ogni query che appunto invocasse il sistema operativo per ottenere una risposta a riguardo. Nel caso la connessione non risultasse disponibile abbiamo scelto di avvertire l’utente facendo comparire un toast sullo schermo che lo invitasse a verificare la sua connessione. La scelta del toast è stata fatta in quanto era la più semplice e meno invasiva per l’utente poiché l’errore, seppure di notevole rilevanza, può essere semplicemente risolto dall’utente stesso abilitando la connessione dati o il Wi-Fi, senza richiedere un riavvio dell’applicazione o altro.

Un altro controllo, anche questo gestito tramite un toast da mostrare all’utente, si è reso necessario nel momento in cui le 3000 query giornaliere a disposizione verso il server della Marvel dovessero esaurirsi.

Dato il frequente ricorso ai toast abbiamo deciso di creare una classe apposta che si occupasse di mostrare questi messaggi sullo schermo, avendo quindi attenzione di passargli di volta in volta il messaggio corretto e il *Context* appropriato riferito all’activity in esecuzione in quel momento.

# Loading

Data la frequente attesa di dati, dovendo quest’ultimi essere reperiti real time da un database esterno, abbiamo deciso di implementare una animazione che si occupasse di rendere noto all’utente il loading dei dati. L’animazione di per sé è molto semplice, richiedendo semplicemente che ogni schermata in cui risultasse necessaria, abbia una View chiamata ProgressBar. Abbiamo deciso quindi che all’apertura di una nuova activity e quindi il caricamento di un nuovo layout, questo presentasse solo la ProgressBar come View visibile. A seguito del reperimento dei dati e del loro corretto caricamento nelle varie RecyclerView, si incrementa un contatore. Quando questo contatore raggiunge un certo valore, che dipende dal numero di RecyclerView per quella schermata, si attiva un metodo che imposta la visibilità della ProgressBar a GONE e quella del resto della schermata in VISIBLE.

# Localizzazione e stringhe

Per evitare valori hard coded nel codice e per far sì che una modifica in un determinato layout possa essere semplicemente riportata anche in tutti gli altri, si è deciso di fare un uso massiccio dei files messi a disposizione nella directory delle resources. Ciò permette anche di sfruttare i vari automatismi messi a disposizione dal sistema operativo Android.

Inn particolare, per far sì che anche utenti che non conoscano l’inglese possano comunque interfacciarsi con relativa semplicità con la nostra applicazione ed aumentare così la base di clienti che ne potrebbero essere interessati, si è deciso di localizzarla in almeno altre due lingue, ossia italiano e tedesco.

Per far ciò abbiamo inserito tutte le stringhe che comparivano nelle Views dei vari layout e nei toasts nel file string.xml. In questo modo, associando ad ogni stringa traducibile, la relativa traduzione nelle due lingue sopra citate (ogni lingua corrisponde ad un diverso file string.xml da riempire), si delega direttamente al sistema operativo il compito di presentare all’utente la traduzione corretta in base alla lingua di preferenza selezionata sul suo dispositivo.

Purtroppo, per ovvi motivi, non è stato possibile localizzare le descrizioni dei vari eroi, fumetti e simili, essendo queste stringhe prese direttamente da terzi e che non presentavano traduzioni in altre lingue oltre all’inglese.

Sfruttando sempre un meccanismo del sistema operativo, basato sui qualificatori, abbiamo anche creato più versioni delle stesse immagini utilizzate all’interno dell’applicazione (a diverse risoluzioni) per far sì che esso mostri quella più adatta ai ddp del dispositivo sul quale l’applicazione sta girando. Naturalmente ognuna di queste immagini è stata inserita nella cartella opportuna.

# Fragments

Per permettere all’applicazione di essere dotata di un menu di navigazione, è stato necessario implementare i fragment.

Per ciascun fragment abbiamo dichiarato il relativo layout (es*. fragment\_search.xml*) e la classe java di riferimento (es SearchFragment.java).

Per assegnare alla classe java il relativo layout utilizziamo il metodo *Inflater.inflate*. Questi layout vengono iniettati nell’activity\_mail.xml attraverso il FrameLayout avente come id fragment\_container.

A questo punto, nella MainActivity gestiamo il listener relativo all’elemento BottomNavigationView, e mediante un costrutto switch-case andiamo ad attivare il fragment desiderato.

Per attivare il fragment bisogna iniziare una FragmentTransaction, e con il metodo *fragmentTransaction.replace* andiamo ad impostare nella transazione di sostituire il fragment attuale contenuto nella fragment\_container con quello selezionato. Con un *commit* confermiamo l’operazione.

# Animazioni

Le animazioni sono dichiarate in file xml e sono contenute nella cartella res/anim.

Il tag apertura di una animazione è <set>, contenente l’attributo duration (in millisecondi), e al suo interno può contenere tag relative ai possibili attributi di un elemento per cui possiamo modificarne il valore. Alcuni esempi sono translate (posizione nello spazio), alpha (trasparenza), scale (dimensioni). Ogni tag contiene elementi che definiscono il valore iniziale, il valore finale e il tipo di interpolazione.

Un esempio pratico di animazione è quello di *anim\_zoom\_in.xml*, in cui animiamo l’effetto con cui i risultati della ricerca sono presentati all’utente.

In questa animazione combiniamo gli effetti dati da translate, alpha e scale, ottenendo la seguente animazione: l’elemento, rispetto al suo stato finale, comincia la sua animazione con una posizione del 20% più in basso lungo l’asse Y, con una trasparenza impostata a 0.5 e una dimensione del 2% più piccola. Successivamente, attraverso una interpolazione di tipo decelerate, completa la sua animazione in 200 millisecondi posizionando il suo offset lungo y a 0, la sua opacità come completa e la sua dimensione al 100%.

Il codice java utilizzato per riprodurre l’animazione è

holder.itemView.setAnimation(AnimationUtils.*loadAnimation*(appContext,R.anim.*anim\_zoom\_in*));

Presa la view desiderata, ne settiamo l’animazione con il metodo *AnimationUtils.loadAnimation*, e viene riprodotta esattamente in quel momento.

# Room

Per poter salvare in locale i dati dell’applicazione, abbiamo utilizzato un database implementato utilizzando la libreria Room.

Il primo passo è definire una classe astratta che estenda RoomDatabase, da noi chiamata AppDatabase. Questa classe utilizzerà il pattern singleton per consentire un unico punto di accesso al database, istanziato mediante il metodo *Room.databaseBuilder*.

Questa classe verrà utilizzata successivamente per chiamare i diversi Dao che abbiamo realizzato, qui dichiarati in maniera astratta.

Il secondo passo è definire le entità. Per ogni entità necessaria creiamo una classe Entity (ossia model), come ad esempio HeroEntity. In questa classe definiamo il mapping tra i campi del database e gli attributi java mediante l’annotazione @*ColumnInfo* da aggiungere in ogni attributo. Definiamo inoltre il mapping tra nome della classe e nome della tabella con @*Entity*, e la chiave primaria con @*PrimaryKey*. Non è necessario nel nostro caso avere una chiave di tipo AutoIncrement, per cui non aggiungiamo ulteriori annotazioni.

Fatto ciò, abbiamo definito i Dao per le entità, che ci consentono di accedere alle tabelle mediante interrogazioni. I Dao sono delle interfacce annotate da @*Dao*, e contengono solamente la firma del metodo. Sarà compito di Room implementare queste operazioni con opportuni metodi. Ogni operazione deve avere una annotazione che può essere @*Insert*, @*Update* e @*Delete* in caso di inserimenti, aggiornamenti e cancellazioni, o @*Query* (seguite dal testo della query) in caso di query personalizzate.

Infine, il codice per eseguire una query sarà del seguente tipo, nel caso di un inserimento:

*AppDatabase.getInstance(getActivity().getApplicationContext()).heroDao().insertHero(hero);*