Chapitre 4 : Composants avancés d’une application mobile

Table des matières

4. Chapitre 4 : Composants avancés d'une application mobile

4.1. Introduction

Dans ce chapitre, on introduit des éléments de listing plus avancés, plus riches et plus récents. Dans un premier lieu on présente la vue CardView pour donner un style plus beau aux icones et aux images comme l'arrondi et l'ombre. Dans un deuxième lieu, on détail l'utilisation des recycler view, comment les intégrer, les alimenter, les actualiser et la définition des actions sur ses éléments.

4.2. CardView

Le composant CardView est un conteneur de composant qui peut essentiellement être considéré comme un FrameLayout avec des coins arrondis et une ombre. un CardView enveloppe une mise en page et est souvent le conteneur utilisé dans une mise en page pour chaque élément dans un ListView ou un RecyclerView.

4.2.1. Utilisation de CardView

Pour créer une CardView, il faut d'abord l'importer de la bibliothèque AndroidX dans le fichier build.gradle du module comme montré par le code source ci-dessous.

```
dependencies {

    implementation "androidx.cardview:cardview:1.0.0"

}
```

Et le contenu de layout sera augmenté par :

```
<androidx.cardview.widget.CardView
    android:id="@+id/cv1"
    android:layout_width="@dimension”
    android:layout_height="@dimension"
    >

    <!-- Main Content View -->

</androidx.cardview.widget.CardView>
```

Le cardview possède des attributs d'arrondi et de l'ombre qui sont cardCornerRadius et cardElevation. Ils peuvent être utilisés comme suit :

```
app:cardCornerRadius="10dp"
app:cardElevation="100dp"
```

Et comme résultat, on trouve la figure ci-dessous :

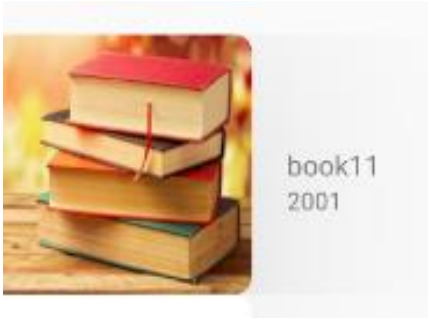

Figure 4.1. Application de l'ombre et de l'arrondi sur un CardView.

4.3. Le composant RecyclerView

Le *RecyclerView* est un *ViewGroup* qui fonctionne avec n'importe quelle vue basée sur un adaptateur. Il est le successeur de *ListView* et *GridView*. *RecyclerView* offre la possibilité d'implémenter des dispositions horizontales et verticales et souvent utilisé avec des collections de données dont les éléments changent au moment de l'exécution en fonction de l'action de l'utilisateur ou d'événements réseau.

De plus, il fournit un support d'animation pour les éléments de *RecyclerView* à chaque fois qu'ils sont ajoutés ou supprimés, ce qui était extrêmement difficile à faire avec *ListView*. *RecyclerView* fonctionne également avec le modèle *ViewHolder*, qui était déjà une pratique recommandée mais maintenant profondément intégrée à ce composant.

4.3.1. Comparaison entre RecyclerView et ListView

RecyclerView diffère de son prédécesseur *ListView* principalement en raison des fonctionnalités suivantes :

- ViewHolder - Les adaptateurs de *ListView* ne nécessitent pas l'utilisation du modèle *ViewHolder* pour améliorer les performances. En revanche, l'implémentation d'un adaptateur pour *RecyclerView* nécessite l'utilisation du modèle *ViewHolder* pour lequel il utilise *RecyclerView.Viewholder*.
- Dispositions d'éléments personnalisables – le composant *ListView* ne peut contenir que des éléments de disposition dans un arrangement linéaire vertical et cela ne peut pas être personnalisé. En revanche, le *RecyclerView* a un *RecyclerView.LayoutManager* qui permet toutes les mises en page d'éléments, y compris les listes horizontales.
- Animations d'éléments – le *ListView* ne contient aucune disposition spéciale permettant d'animer l'ajout ou la suppression d'éléments. En revanche, le

RecyclerView a la classe *RecyclerView.ItemAnimator* pour gérer les animations d'éléments.

- Source de données manuelle – le composant *ListView* avait des adaptateurs pour différentes sources telles que *ArrayAdapter* et *CursorAdapter* pour les tableaux et les résultats de base de données respectivement. En revanche, *RecyclerView.Adapter* nécessite une implémentation personnalisée pour fournir les données à l'adaptateur.
- Décoration manuelle des éléments – le composant *ListView* a la propriété *android:divider* nécessaire pour implémenter des séparateurs entre les éléments de la liste. En revanche, *RecyclerView* nécessite l'utilisation de l'objet *RecyclerView.ItemDecoration* pour configurer beaucoup plus de décorations de séparation.
- Détection de clique – le composant *ListView* a une interface *AdapterView.OnItemClickListener* pour interagir avec les événements de type clique pour les éléments individuels de la liste. En revanche, *RecyclerView* ne prend en charge que *RecyclerView.OnItemTouchListener* qui gère les événements tactiles individuels.

4.3.2. L'utilisation de RecyclerView

Pour utiliser le composant *RecyclerView*, il faut ajouter les dépendances suivantes au gradle du module.

```
dependencies {
    implementation("androidx.recyclerview:recyclerview:1.2.1")
    implementation("androidx.recyclerview:recyclerview-selection:1.1.0")
}
```

L'utilisation d'un *RecyclerView* nécessite les étapes suivantes :

1- Définir une classe du modèle à utiliser comme source de données.

2- Ajouter un RecyclerView à l'activité pour afficher les éléments.

3- Créer un fichier XML de disposition de ligne personnalisée.

4- Créer un RecyclerView.Adapter et un ViewHolder.

5- Lier l'adaptateur à la source de données pour remplir le RecyclerView.

4.3.2.1. Définir une classe de modèle

Chaque *RecyclerView* est soutenu par une source de données. Dans ce cas, on doit définir une classe item (Book dans le contexte du TP) qui représente le modèle de données affiché par le *RecyclerView*. La classe ItemBook est donnée par le code ci-dessous.

```
package dz.ubma.bookcollection_.fragments;
public class itemBook {
    // Attributs
    String title;
    String year;
    String id;
    String iconBook;
    int iconStar;
    // Setters et Getters
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getYear() {
        return year;
    }
    public void setYear(String year) {
        this.year = year;
    }
    public String getIconBook() {
        return iconBook;
    }
    public void setIconBook(String iconBook) {
        this.iconBook = iconBook;
    }
    public int getIconStar() {
        return iconStar;
    }
    public void setIconStar(int iconStar) {
        this.iconStar = iconStar;
    }

    // Constructeurs
    public itemBook(){}

    public itemBook(String title, String year, String id, String iconBook,
int iconStar) {
        this.title = title;
        this.year = year;
        this.id = id;
        this.iconBook = iconBook;
        this.iconStar = iconStar;
    }
}
```

4.3.2.2. Ajouter une RecyclerView à l'activité

Dans le fichier XML de présentation d'activité souhaité localisé dans res/layout/___.xml, on ajoute le composant RecyclerView comme montré ci-dessous:

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/rvBooks"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"
    android:orientation="vertical"
    />
```

Et comme résultat, on aura la vue montrée par la figure 4.2. Une fois le RecyclerView est intégré dans le fichier de mise en page d'activité. On peut définir la mise en page pour chaque élément de la liste.

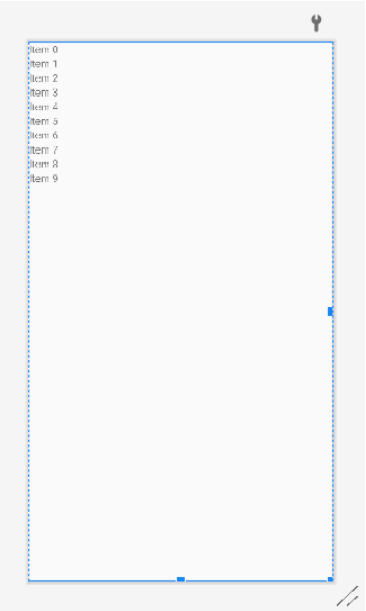

Figure 4.2. RecyclerView de base.

4.3.2.3. Créer un fichier XML de disposition de ligne personnalisée

Avant de créer l'adaptateur, on définit le fichier de mise en page XML qui sera utilisé pour chaque ligne de la liste. Cette disposition d'élément pour l'instant devrait contenir une disposition linéaire horizontale avec :

- Une vue textuelle pour le titre et l'année de livre,
- Et des vues images pour l'icône du livre et pour l'étoile qui indique si le livre est mis en favoris.

Le code XML en question est comme suit :

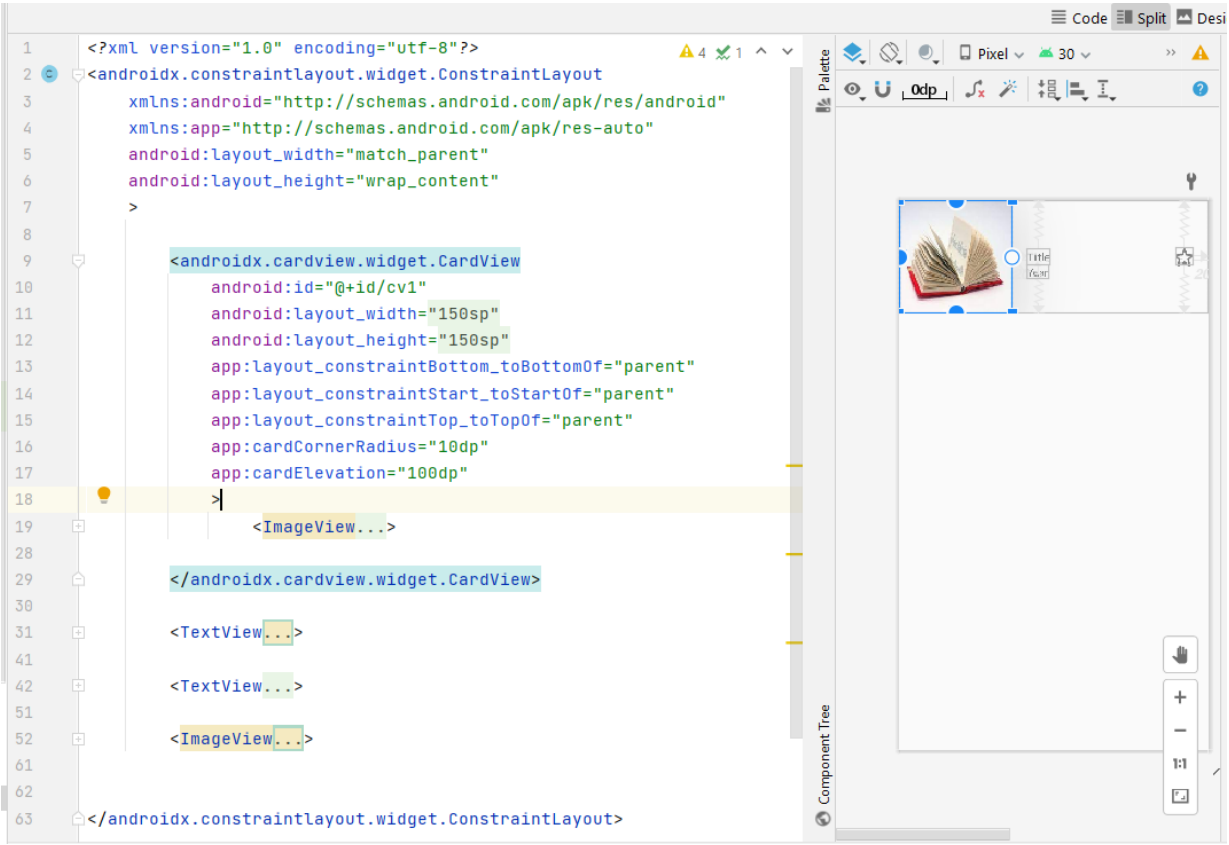

Figure 4.3. Contenu XML de disposition de ligne de RecyclerView.

4.3.2.4. Créer un RecyclerView.Adapter et un ViewHolder

Une fois la disposition de l'élément personnalisé terminée, on peut créer l'adaptateur pour remplir les données dans la vue du recycleur.

Le rôle de l'adaptateur est de convertir un objet à une position en un élément de ligne de liste. Cependant, avec un *RecyclerView*, l'adaptateur nécessite l'existence d'une classe *ViewHolder* qui décrit et donne accès à toutes les vues dans chaque ligne d'élément. On peut créer l'adaptateur et le holder ensemble dans la même classe comme montré dans l'exemple ci-dessous.

```
class bookHolder extends RecyclerView.ViewHolder {

    // Le contenu d'une ligne de recyclerView
    ImageView iCbookXML;
    ImageView iCStarXML;
    TextView txtYearXML;
    TextView txtTitleXML;

   // Setters et getters
    public ImageView getiCbookXML() {
        return iCbookXML;
    }

    public ImageView getiCStarXML() {
        return iCStarXML;
    }

    public TextView getTxtYearXML() {
        return txtYearXML;
    }

    public TextView getTxtTitleXML() {
        return txtTitleXML;
    }

  // Le Constructeur
    public bookHolder(View view) {
        super(view);

        iCbookXML = view.findViewById(R.id.ICbookXML);
        iCStarXML = view.findViewById(R.id.ICStarXML);
        txtYearXML = view.findViewById(R.id.txtYearXML);
        txtTitleXML = view.findViewById(R.id.txtTitleXML);
    }
}
```

Cependant, le contenu de l'adaptateur est comme suit :

```
1   package dz.ubma.bookcollection_.recycler;
2
3   import ...
20
21  public class booksRecyclerViewAdapter extends RecyclerView.Adapter<booksRecyclerViewAdapter.bookHolder> {
22      ArrayList<itemBook> lBooks = new ArrayList<>();
23      Context context;
24
25      public booksRecyclerViewAdapter(Context _context, int LayoutID, ArrayList<itemBook> _lBooks) {...}
30
31      @NonNull
32      @Override
33      public bookHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {...}
39
40      @Override
41      public void onBindViewHolder(@NonNull bookHolder holder, int position) {...}
60
61      @Override
62      public int getItemCount() { return lBooks.size(); }
65
66      class bookHolder extends RecyclerView.ViewHolder {...}
99  }
100
```

Figure 4.4. Le contenu de l'adaptateur.

A l'intérieur de la classe Adaptateur, on déclare ce qui suite

- **Déclaration de l'adapteur**

L'adapteur est une classe qui hérite de la classe de base *RecyclerView.Adapter* d'une collection définit entre les opérateurs < et >, cette collection est basée sur l'élément bookHolder définit plus haut. Ci-dessous la déclaration de la classe adaptateur.

```
public class booksRecyclerViewAdapter extends
RecyclerView.Adapter<booksRecyclerViewAdapter.bookHolder>
```

- **Création du constructeur**

Le constructeur sert à instancier l'objet Adapter avec le passage de quelques paramètres en option pour une utilisation ultérieure. Le constructeur est définit comme suit :

```
public booksRecyclerViewAdapter(Context _context, int LayoutID,
ArrayList<itemBook> _lBooks) {
    super();
    lBooks = _lBooks;
    context = _context;
}
```

- **Augmentation de la méthode onCreateViewHolder**

La méthode *onCreateViewHolder* ne doit pas retourner une valeur nul. Son rôle est d'injecter la vue d'une ligne dans son parent qui est de type RecyclerView.

```
@NonNull
@Override
public bookHolder onCreateViewHolder(@NonNull ViewGroup parent, int
viewType) {
    View v = LayoutInflater
            .from(parent.getContext())
            .inflate(R.layout.item_book, parent, false);
    return new bookHolder(v);
}
```

- **Augmentation de la méthode onBindViewHolder**

L'objectif de la méthode *onBindViewHolder* est de récupérer la position de l'élément à afficher, récupérer l'élément à la position indiquée à partir d'une source de données, et binder les composantes du holder avec l'élément récupéré. Ci-dessous un exemple du contenu de la méthode onBindViewHolder.

```
@Override
public void onBindViewHolder(@NonNull bookHolder holder, int position) {
    itemBook it = lBooks.get(position);

Glide.with(context).load(Uri.parse(it.getIconBook())).into(holder.iCbookXML
);
    int imageStar = it.getIconStar() == 0 ? R.drawable.ic_star :
R.drawable.ic_f_start;
    holder.iCStarXML.setImageResource(imageStar);
    holder.txtTitleXML.setText(it.getTitle());
    holder.txtYearXML.setText(it.getYear());

    holder.iCStarXML.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            it.setIconStar(it.getIconStar() == 0 ? 1 : 0);
            int imageStar = it.getIconStar() == 0 ? R.drawable.ic_star :
R.drawable.ic_f_start;
            holder.iCStarXML.setImageResource(imageStar);
            bookControler bControler = bookControler.getInstance();
            bControler.updateData(it.getId(),it.getIconStar());
        }
    });
}
```

- **Augmentation de la méthode getItemCount**

La méthode *getItemCount* permet de retourner le nombre d'éléments à afficher dans le recyclerView.

```
@Override
public int getItemCount() {
    return lBooks.size();
}
```

4.3.2.5. Lier l'adaptateur à la source de données pour remplir le RecyclerView

A l'aide de la méthode setAdapter, on peut affecter l'adapter déclaré plus haut au composant Recyclerview récupéré par la méthode findViewById comme montré par le code ci-dessous.

```
RecyclerView recyclerView = view.findViewById(R.id.rvBooks);

recyclerView.setAdapter(new booksRecyclerViewAdapter(context
        , R.layout.item_book
        , lBook));
```

Un exemple de recyclerView implémenté dans le contexte du TP est :

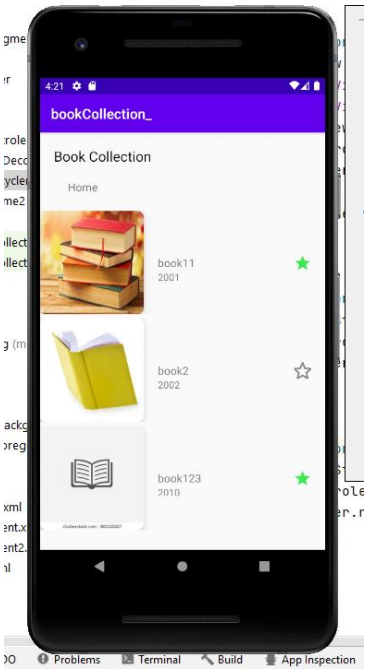

Figure 4.5. un exemple de RecyclerView.

4.4. Conclusion

Dans ce chapitre, on a utilisé des éléments plus avancés pour la création des listes d'objets. Le recyclerView est indispensable aux étudiants pour créer des listes verticales et aussi horizontales. A la fin de ce chapitre, l'étudiant améliore sa compréhension du système de fonctionnement des vues en général notamment la définition imbriquée de ces vues, et comment les données sont bindées et affichées. De plus il apprend les méthodes à implémenter et à réutiliser pour la réalisation de sa liste d'objets.