27 novembre 2024

Labo 4 - Architecture MVVM, utilisation d'une base de données Room et d'un RecyclerView

Vitória Oliveira **Camille Koestli**

Table des matières

1. Implémentation

1. La MainActivity joue un rôle important en gérant les fragments et en configurant les menus pour des actions comme le tri, la suppression des données et la génération de notes. Elle agit également comme un médiateur entre les fragments et les ViewModels. Elle donne la possibilité d'une bonne coordination des données et des actions utilisateur.

2. Nous avons implémenté deux versions de activity_main.xml, qui permet un support multi-écrans avec des layouts adaptatifs :
   - Une version pour les smartphones qui affiche un seul fragment à la fois.
   - Une version pour les tablettes (activity_main_large-land.xml) qui affiche côte à côte les fragments NotesFragment et ControlsFragment. Nous avons pu faire cette manipulation grâce à FragmentContainerView.

3. L'architecture MVVM permet la gestion des données NotesViewModel et centralise l'accès aux données via un DataRepository. Les méthodes sortNotesByCreationDate ou generateA-Note sont encapsulées dans le ViewModel pour assurer une séparation claire

4. Des fragments spécialisés ont été mis en place. Ils sont indépendants, mais partagent les mêmes ViewModels pour accéder aux données.
   - NotesFragment : Gère l'affichage des notes via une RecyclerView.
   - ControlsFragment : Exclusif aux tablettes, il fournit des actions comme la génération ou la suppression de notes via des boutons.

5. RecyclerView permet l'affichage dynamique des notes, Les notes sont affichées dans une liste défilante grâce à RecyclerView. DiffUtil est utilisé pour garantir des mises à jour efficaces et réactives de la liste.

6. Les actions comme le tri, la suppression ou la génération de notes sont définies dans un menu (res/menu/main_menu.xml) accessible via la MainActivity. Sur tablette, ces actions sont remplacées par des boutons dans le ControlsFragment.

7. Les dates complexes sont gérées via un Converter, une classe `@TypeConverter` pour convertir des objets Calendar en valeurs Long compatibles avec la base de données Room. Ce convertisseur est enregistré dans la base via l'annotation `@TypeConverter`.

8. NotesAdapter permet gérer l'affichage des éléments RecyclerView. L'adaptateur adapte les données des notes aux vues, en tenant compte du type de note (avec ou sans planification). L'utilisation des classes NoteItem garantit une gestion typée des notes.

2. Questions

2.1. Question 1

Quelle est la meilleure approche pour sauver, même après la fermeture de l'app, le choix de l'option de tri de la liste des notes? Vous justifierez votre réponse et l'illustrez en présentant le code mettant en oeuvre votre approche.

La manière d'implémenter cette feature est l'utilisation de SharedPreferences pour sauver l'état du tri. Ces dernières permettent de stocker des données primitives dans des paires clé-valeur, ce qui est idéal pour enregistrer des paramètres ou des préférences simples. Pour l'implémenter:

1. Création d'une classe pour gérer les préférences

```
import android.content.Context
import android.content.SharedPreferences
```

```
class SortPreferenceManager(context: Context) {
    private val preferences: SharedPreferences =
        context.getSharedPreferences("notes_preferences", Context.MODE_PRIVATE)

    companion object {
        const val SORT_KEY = "sort_option"
        const val DEFAULT_SORT = "creation_date" // Valeur par défaut
    }

    // Méthode pour sauvegarder l'option de tri
    fun saveSortOption(sortOption: String) {
        preferences.edit().putString(SORT_KEY, sortOption).apply()
    }

    // Méthode pour récupérer l'option de tri
    fun getSortOption(): String {
        return preferences.getString(SORT_KEY, DEFAULT_SORT) ?: DEFAULT_SORT
    }
}
```

2. Intégration dans le ViewModel

```
import android.app.Application
import androidx.lifecycle.AndroidViewModel

class NotesViewModel(application: Application) : AndroidViewModel(application) {
    private val sortPreferenceManager = SortPreferenceManager(application)
    private var currentSortOption: String = sortPreferenceManager.getSortOption()

    // Récupérer la liste triée en fonction de l'option actuelle
    fun getSortedNotes(): LiveData<List<NoteAndSchedule>> {
        return when (currentSortOption) {
            "creation_date" -> repository.getNotesSortedByCreationDate()
            "schedule_date" -> repository.getNotesSortedByScheduleDate()
            else -> repository.getNotesSortedByCreationDate()
        }
    }

    // Mettre à jour l'option de tri et sauvegarder
    fun updateSortOption(sortOption: String) {
        currentSortOption = sortOption
        sortPreferenceManager.saveSortOption(sortOption)
    }
}
```

3. Application de l'option de tri dans l'UI

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
        R.id.main_menu_sort_creation -> {
            notesViewModel.updateSortOption("creation_date")
            true
        }
        R.id.main_menu_sort_schedule -> {
            notesViewModel.updateSortOption("schedule_date")
            true
        }
```

```
        else -> super.onOptionsItemSelected(item)
    }
}
```

2.2. Question 2

L'accès à la liste des notes issues de la base de données Room se fait avec une LiveData. Est-ce que cette solution présente des limites? Si oui, quelles sont-elles? Voyez-vous une autre approche plus adaptée?

LiveData est une solution pratique et efficace dans beaucoup de cas, mais elle montre ses limites lorsqu'il s'agit de gérer de grandes quantités de données ou des requêtes complexes. Comme ses mises à jour se font sur le thread principal, cela peut entraîner des ralentissements et diminuer les performances si les données sont trop volumineuses.
En plus, LiveData ne gère pas la régulation du débit des données (le backpressure) entre la source et leur traitement (le consommateur), ce qui peut surcharger ce dernier.
En autre, LiveData ne propose pas d'outils intégrés pour transformer, filtrer ou combiner les données, ce qui rend les manipulations plus complexes plus difficiles à mettre en œuvre.

Une alternative est d'utiliser Flow, une API réactive de Kotlin. Il prend en charge la gestion du backpressure, ce qui le rend idéal pour traiter des données volumineuses ou complexes. Il propose également de nombreux opérateurs comme map, filter ou combine, qui permettent de manipuler les flux de données de manière fluide.
Flow peut être utilisé avec Room et peut être collecté directement dans l'UI ou converti en LiveData. C'est une solution idéalie pour des traitements plus complexes.

2.3. Question 3

Les notes affichées dans la RecyclerView ne sont pas sélectionnables ni cliquables. Comment procéderiez-vous si vous souhaitez proposer une interface permettant de sélectionner une autre note pour l'éditer?

1. Création du layout d'édition en utilisant une ScrollView afin d'éditer un élément NoteAndSchedule. Ex:

```
<ScrollView
  ...

    <LinearLayout
      ...

        <TextView
            android:id="@+id/note_title"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Title"
            android:layout_marginBottom="8dp"
            android:inputType="text" />
        <EditText
            android:id="@+id/note_title_edit"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Title"
            android:layout_marginBottom="8dp"
            android:inputType="text" />

      ...
```

2. Création d'une interface onClickListener qui permet d'informer le fragment/activité lors d'un click sur une note. On y déclare une fonction onNoteClick qui prend une note en paramètre.
3. Cette interface sera implementée dans le fragment qui contient le RecyclerView. C'est aussi à ce moment là qu'on défini la fonction onNoteClick: elle contiendra la logique de navigation vers le fragment qui gère l'édition d'une note.
4. Création du fragment d'édition de note: la fonction onViewCreated de ce fragment affichera les informations actuelles de la note cliquée via le NoteViewModel et l'affiche via le layout d'édition.
5. Mise en place du lien entre le RecyclerView et l'interface dans le NoteAdapter. Il faudrait passer l'interface onClickListener à l'Adapteur mais aussi créer une méthode onClickListener dans le ViewHolder qui appelera la méthode de l'interface onClickListener du point précédant.
6. Il faut une méthode updateNote dans le NoteViewModel pour récupérer les nouvelles informations et modifier la note cliquée. Par conséquent, il faut implémenter la même chose dans le Repository ainsi que le Dao puisque ces méthodes s'appellent entre elles (VM -> Repo -> Dao). Dans le Dao plus précisement, il faut préciser une requête SQL « UPDATE » qui met à jour la note dont l'id est le même que la note cliquée avec les valeurs des attributs de cette dernière.