DAA LABORATOIRE 3

Enseignant Fabien Dutoit

Assistant Ilias Goujgali

Auteurs Damien Maier, Jean-François Pasche, Vincent Peer

Date 04.12.2022

Classe DAA-TIC-A-L2

Conception du squelette

Nous avons créé

- La mainActivity, qui détermine un layout pour le menu et détermine les différentes actions des boutons des menus, qui déclenchent des appels de méthodes se trouvant dans le ViewModel. La mainActivity a un layout principal et deux layouts alternatifs (large-land et large-port), qui permettent d'afficher automatiquement la présentation appropriée selon si on est sur un téléphone, une tablette en portrait ou une tablette en paysage
- Le fragment pour la liste des notes. Il a un accès au même ViewModel que la mainActiviy.
- Le fragment de contrôle. Il a un accès au même ViewModel que la mainActivity. Il possède des boutons déclenchant des appels à des méthodes du ViewModel, et affiche un compteur de notes qui est automatiquement mis à jour en fonction d'une liveData se trouvant dans le ViewModel.

Mise en place de la base de données Room

Nous avons créé

- Une classe CalendarConverter qui permet de faire la conversion entre les dates stockées dans la base de données sous forme d'un entier, et le type Calendar que nous utilisons
- Une interface NotesDAO qui décrit des méthodes pour ajouter des notes et des schedules, supprimer toutes les notes, supprimer tous les schedules, lire les notes et schedules, connaître le nombre de notes. Ces méthodes sont implémentées automatiquement par Room.
- Une classe NoteDatabase qui fait le lien entre les entités, le converter et la DAO. On accède à la base de données via le companion object de cette classe, qui gère un singleton. Nous utilisons une classe CreationCallback qui remplit la bdd avec du contenu lorsqu'elle est créée.
- Une classe Repository, qui sert à donner accès à la bdd au reste de l'application.
- Une classe NoteApp qui hérite de Application et qui stocke la base de données et donne accès au repository.

Conception du ViewModel

Nous avons créé une classe NoteViewModel, qui met à disposition une liveData pour la liste des notes, une liveData pour le nombre de notes, une méthode pour générer une note aléatoire et une méthode pour supprimer toutes les notes.

RecyclerView

Nous avons créé

- Un layout pour le fragment des notes, qui contient une RecyclerView
- Un layout list_item_note qui décrit l'affichage d'une note sans schedule
- Un layout list_item_note_schedule qui décrit l'affichage d'une note avec schedule
- Un adaptateur NotesListAdapter qui
    - Utilise une classe NoteDiffCallback qui nous avons implémentée pour détecter ce qui a changé lorsque la liste de notes est modifiée, ce qui permet d'avoir une animation quand on change l'ordre de tri, et évite de recréer entièrement la recyclerView
    - A des méthodes pour trier ses éléments par ordre de création et de deadline

- A une classe interne ViewHolder qui représente un élément de la liste et est responsable d'afficher l'icône, la couleur, etc en fonction du contenu de la note.

Questions complémentaires

1. Quelle est la meilleure approche pour sauver 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 œuvre votre approche.

Les *SharedPreferences* permettent de stocker des informations isolées sous forme de clé valeur qui persistent si l'application est fermée. C'est un outil qui convient pour mémoriser le choix de tri que l'utilisateur a saisi. Pour la mise en œuvre depuis notre *mainActivity*, nous accédons aux préférences à partir du *onCreateView* pour récupérer une éventuelle valeur de tri déjà sauvée. Si aucune valeur n'est disponible pour la clé demandée, alors nous trions par défaut par date de création.

Obtention d'un choix de tri existant ou tri par date sinon :

```
val sortedValue : String? = getPreferences(Context.MODE_PRIVATE).getString("sorted_choice", "CreationDate")
viewModel.sorting.postValue(NoteViewModel.Sorting.valueOf(sortedValue!!))
```

Il faut ensuite mettre à jour la préférence en ajoutant la clé/valeur correspondante lorsque l'utilisateur entre un ordre de tri. Pour ça, nous ajoutons l'information depuis la fonction *onOptionsItemSelected* qui possède des callbacks pour gérer le changement de tri, parmi d'autres gestions.

```
R.id.creation_date -> {
    viewModel.sorting.postValue(NoteViewModel.Sorting.CreationDate)
    getPreferences(Context.MODE_PRIVATE).edit().putString("sorted_choice", "CreationDate").apply()
    true
}
R.id.eta -> {
    viewModel.sorting.postValue(NoteViewModel.Sorting.Schedule)
    getPreferences(Context.MODE_PRIVATE).edit().putString("sorted_choice", "Schedule").apply()
    true
```

Ici, si le tri par date de création est choisi, nous introduisons cette donnée dans les préférences avec « CreationDate » comme valeur de la clé sorted_choice. Le principe est le même pour « Schedule ».

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 ?

La solution actuelle implique que toutes les notes sont chargées en mémoire, même celles qui ne sont pas actuellement visibles à l'écran.

Si il y a beaucoup de notes, il serait préférable que celles-ci soient chargées et déchargées de la mémoire au fur et à mesure que l'utilisateur scroll dans la recycler view. Cela peut être fait avec la librairie Paging, comme expliqué en détail ici : https://genicsblog.com/gouravkhunger/pagination-in-android-room-database-using-the-paging-3-library

Les principales modifications à faire sur notre code seraient les suivantes :

- Dans l'interface NotesDAO, prévoir une méthode getPage(minId : int, maxId : int) qui charge seulement un sous-ensemble des notes (avec une requête SELECT qui trie les notes par ID et renvoie les notes dont l'ID se trouve entre un minimum et un maximum qu'on passe en argument)

- Créer une classe qui hérite de PagingSource et qui fait des appels à notre fonction décrite au point précédent. Cette classe permettra de générer des pages de notes à la demande.
- Remplacer notre NotesListAdapter par un adapter qui hérite de PagingDataAdapter. On pourra accéder à la note à la position n en appelant getItem(n), ce qui va automatiquement charger la page contenant cette note.

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

On peut utiliser androidx.recyclerview.selection qui est documenté ici : https://developer.android.com/reference/androidx/recyclerview/selection/package-summary

L'idée générale est la suivante :

- Créer une classe qui hérite de ItemDetailsLookup et redéfinir la méthode getItemDetails qui sert à déterminer quel élément doit être sélectionné lorsque l'utilisateur appuie sur l'écran.
- Créer une classe qui hérite de ItemKeyProvider et implémenter ses méthodes getKey et getPosition pour faire le lien entre la position de l'élément et sa key
- Quand l'utilisateur sélectionne un élément, la méthode onBindViewHolder de l'adapter va être rappelée pour cet élément. Il faut donc que onBindViewHolder vérifie si l'élément est sélectionné, et le cas échéant donne le statut « activé » à la vue de l'élément sélectionné, et changer la façon dont il est affiché pour que la sélection soit visible pour l'utilisateur
- Instancier un selectionTracker qui prend dans son constructeur des instances des deux classes décrites aux points 1 et 2, et la recyclerView
- Utiliser la méthode SelectionObserver du recyclerView pour enregistrer un callback qui change l'affichage de manière appropriée en fonction de si un élément est sélectionné (par exemple, qui fait apparaître un bouton « éditer »).