DAA-L4-MVVM

Auteurs : Bugna, Slimani & Steiner

Réponses aux questions

6.1

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 œuvre votre approche.

La meilleure approche pour sauvegarder le choix de l'option de tri, même après la fermeture de l'application, est d'utiliser **SharedPreferences**. C'est une solution simple, efficace, et adaptée à la sauvegarde de petites données persistantes comme un choix d'option.

Pourquoi **SharedPreferences** ?

- Persistant : Les données restent disponibles même après la fermeture de l'application.
- Facile à implémenter : Aucune configuration complexe n'est requise.
- Efficace pour les données légères : Idéal pour stocker des options utilisateur ou des valeurs simples comme des chaînes de caractères, des booléens ou des entiers.

Sauvegarder l'option de tri : Lorsqu'une option est sélectionnée, on la stock dans SharedPreferences :

```
fun saveSortPreference(context: Context, sortOption: String) {
    val sharedPreferences = context.getSharedPreferences("preferences",
Context.MODE_PRIVATE)
    sharedPreferences.edit().putString("sort_option", sortOption).apply()
}
```

Récupérer l'option sauvegardée : Au démarrage de l'application, l'option sauvegardée est récupérée pour initialiser le tri :

```
fun getSortPreference(context: Context): String {
    val sharedPreferences = context.getSharedPreferences("preferences",
Context.MODE_PRIVATE)
    return sharedPreferences.getString("sort_option", "BY_DATE") ?: "BY_DATE"
}
```

Dans `NoteViewModel` on ajoute : un champ pour stoquer le context initialiser par un passage en parametre (en plus du `NoteRepository`), il faut donc modifier les appel au constructeur de `NoteViewModel` depuis la factory ainsi que les appels aux constructeurs de la factory depuis les fragments et `mainActivity`.

On ajoute aussi :

```
    init {
        val savedSort = getSortPreference(context)
        _sortedNotes.value = when (savedSort) {
            "BY_DATE" -> SortType.BY_DATE
            "BY_SCHEDULE" -> SortType.BY_SCHEDULE
            else -> SortType.BY_DATE
        }
    }
```

```
fun updateSortType(sortType: SortType) {
    _sortedNotes.value = sortType
    saveSortPreference(context, sortType.name)
}
```

Il rest donc a appeler `saveSortPreference()` aux endroits où l'on modifie les options de tri.

6.2

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 ?

Limites de LiveData :

- Pas de gestion fine des erreurs :
  - LiveData ne permet pas de capturer les erreurs issues des requêtes Room.
  - Si une requête échoue (par exemple, une requête complexe mal formée), il n'est pas possible de signaler cette erreur directement avec LiveData.
- Chargement initial :
  - LiveData charge immédiatement les données disponibles dans la base de données au moment de la souscription, ce qui peut être problématique si des conditions doivent être vérifiées avant.
- Manque de flexibilité :
  - Les transformations complexes ou asynchrones nécessitent des wrappers comme MediatorLiveData.

Alternative : utiliser Kotlin Flow, cela offre une approche plus moderne et flexible :

- Gestion des erreurs : Flow capture et gère facilement les erreurs.
- Asynchrone : Il est conçu pour travailler nativement avec les coroutines.
- Opérations complexes : Les opérateurs comme map, filter, et combine facilitent les transformations.

6.3

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 ?

Pour permettre de sélectionner une note dans une RecyclerView afin de l'éditer, il est nécessaire d'apporter quelques modifications à l'adaptateur et de prévoir une activité ou un fragment dédié à l'édition. Tout d'abord, l'adaptateur doit être configuré pour détecter les clics sur chaque élément de la liste. Cela se fait en ajoutant un OnClickListener à la vue correspondant à chaque note dans le ViewHolder. Plutôt que de gérer directement l'action à réaliser dans l'adaptateur, il est préférable d'utiliser un système de callback ou une interface pour notifier l'activité ou le fragment qu'une note a été sélectionnée.

Une fois le clic détecté, l'activité ou le fragment qui héberge la RecyclerView peut réagir en lançant une activité d'édition. Cette nouvelle activité doit recevoir les informations nécessaires sur la note, par exemple via un Intent contenant l'identifiant ou d'autres données de la note. L'interface d'édition permettra alors à l'utilisateur de modifier les champs de la note et de valider ses changements.

Lorsque l'édition est terminée, il est important de revenir à la liste des notes avec une mise à jour pour refléter les modifications apportées. Cela peut se faire en rechargeant les données depuis la base ou en notifiant directement l'adaptateur d'un changement. Cette approche garantit une expérience utilisateur fluide tout en gardant le code bien organisé.