DAA - Laboratoire 4

November 25, 2024

Émilie Bressoud **Sacha Butty** **Loïc Herman**

Réponses aux questions

Question 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.*

Android offre via l'extension `lifecycle` une API nommée `SharedPreferences` qui permet de sauvegarder des valeurs sous une forme clé-valeur dans un fichier géré par l'API. Cette solution est relativement simple et n'est donc pas adaptée si l'application aura besoin d'être très réactive sur les changements de préférences ou si les données à sauvegarder sont complexes. Dans ce dernier cas, il est préférable d'utiliser la base de données Room ou alors l'extension `DataStore` qui est construite sur les coroutines et les `Flow`.

Dans le cas de l'option de tri de la liste des notes, qui est une donnée très simple et qui ne change pas souvent, l'utilisation de `SharedPreferences` est donc une solution suffisante et adaptée à nos besoins.

Nous avons réalisé l'approche de sauvegarde de l'option de tri de la liste des notes en utilisant `SharedPreferences` en mettant en premier lieu en place un mécanisme qui permet d'intégrer la valeur sauvegardé dans un `LiveData` de `lifecycle` pour permettre de mettre en place une pipeline de données réactive avec les données venant de la base de données. Pour ceci, nous avons en premier lieu défini la classe abstraite `SharedPreferenceLiveData` qui permet de transformer un `SharedPreferences` en `LiveData`.

```kotlin
// lifecycle.SharedPreferenceLiveData.kt
/**
 * A mutable LiveData implementation that observes changes in a SharedPreferences object.
 * It is used to observe changes in the SharedPreferences object and update the UI accordingly.
 *
 * @param T the type of the value to observe
 * @param sharedPreferences the SharedPreferences object to observe
 * @param key the key of the value to observe
 * @param defaultValue the default value of the value to observe
 *
 * @author Emilie Bressoud
 * @author Loïc Herman
 * @author Sacha Butty
 */
abstract class SharedPreferencesLiveData<T>(
    protected val sharedPreferences: SharedPreferences,
    protected val key: String,
    protected val defaultValue: T
) : LiveData<T>() {

    /**
     * Gets the value from the SharedPreferences object.
     *
     * @return the value from the SharedPreferences object
     */
    protected abstract fun getValueFromPreferences(): T

    /**
     * Sets the value to the SharedPreferences object.
     *
     * @param value the value to set
     */
    protected abstract fun setValueToPreferences(value: T)

    final override fun onActive() {
        super.onActive()
```

```
        super.setValue(getValueFromPreferences())
        sharedPreferences.registerOnSharedPreferenceChangeListener(changeListener)
    }

    final override fun onInactive() {
        super.onInactive()
        sharedPreferences.unregisterOnSharedPreferenceChangeListener(changeListener)
    }

    public final override fun setValue(value: T) {
        setValueToPreferences(value)
    }

    private val changeListener = SharedPreferences.OnSharedPreferenceChangeListener { _, key ->
        if (key == this.key) {
            super.setValue(getValueFromPreferences())
        }
    }
}
```

Comme nous utilisons un `enum` pour représenter les options de tri, nous avons créé une version spécialisée de `SharedPreferencesLiveData` pour les `enum`, nommée `SharedPreferencesEnumeratedLiveData`, offrant l'API nécessaire pour faire fonctionner le mécanisme avec tout type d'`enum` (via le type générique `T` qui a la contrainte d'étendre `Enum<T>`).

```
// lifecycle.SharedPreferenceLiveData.kt
/**
 * [SharedPreferencesLiveData] implementation for Enum values.
 *
 * @param T the Enum type of the value to observe
 * @param sharedPreferences the SharedPreferences object to observe
 * @param key the key of the value to observe
 * @param defaultValue the default enum value of the value to observe
 *
 * @author Emilie Bressoud
 * @author Loïc Herman
 * @author Sacha Butty
 */
class SharedPreferencesEnumeratedLiveData<T : Enum<T>>(
    sharedPreferences: SharedPreferences,
    key: String,
    defaultValue: T
) : SharedPreferencesLiveData<T>(sharedPreferences, key, defaultValue) {

    private val enumClass = defaultValue::class.java

    override fun getValueFromPreferences(): T = sharedPreferences
        .getString(key, defaultValue.name)!!
        .let { enumClass.enumConstants!!.first { enum -> enum.name == it } }

    override fun setValueToPreferences(value: T) = with(sharedPreferences.edit()) {
        putString(key, value.name)
        apply()
    }
}
```

Ensuite, en utilisant l'`enum NoteViewModel.SortOrder` que nous avons défini lors de l'implémentation du menu, nous pouvons initialiser un `SharedPreferencesEnumeratedLiveData` pour l'option de tri de la liste des notes dans le `NoteViewModel`.

```
// viewmodel.NoteViewModel.kt
private const val NOTES_PREFS_CONTEXT_KEY = "notes_prefs"
private const val NOTES_PREFS_KEY_SORT_ORDER = "sort_order"

class NoteViewModel /* ... */ {
```

```
    // Mutable data binding allowing to save [SortOrder] in the shared preferences
    private val sortOrder = SharedPreferencesEnumeratedLiveData(
        application.getSharedPreferences(NOTES_PREFS_CONTEXT_KEY, Context.MODE_PRIVATE),
        NOTES_PREFS_KEY_SORT_ORDER,
        SortOrder.NONE
    )
}
```

La puissance du binding de `SharedPreferences` en `LiveData` permet maintenant de définir la liste des notes triées en fonction de l'option de tri sauvegardée dans les préférences de l'application, de façon totalement réactive en cas de changement provenant de n'importe quelle source.

```
// viewmodel.NoteViewModel.kt
class NoteViewModel /* ... */ {

    /** Reference to the observable note list */
    val allNotes = repository.notes

    /** Reference to the observable note list sorted by the saved sort order */
    val sortedNotes = sortOrder.switchMap { sort -> allNotes.map { notes -> sort.sorter(notes) } }
}
```

Pour que le binding fonctionne, il est néanmoins nécessaire que `NoteViewModel` étende `AndroidViewModel` prenant en paramètre l'application pour récupérer l'instance de `SharedPreferences` de l'application à la construction du view model. Une factory est également nécessaire pour instancier le view model avec le scope précis nécessaire pour que les opérations de lecture et d'écriture dans la base de données Room fonctionnent. Les fragments et activités devront également utiliser la factory pour instancier le view model.

On notera toutefois que cette modification était de toute façon nécessaire pour que le view model puisse récupérer le repository qui est utilisé pour les autres opérations du view model. Nous avons du coup utilisé certains utilitaires (comme `viewModelFactory`) fournis par l'extension `lifecycle` pour simplifier l'import du view model dans les fragments et activités le nécessitant. On fait également usage de la clé `APPLICATION_KEY` qui est fournie par défaut dans les `CreationExtras` par la factory pour tous les view models héritant de `AndroidViewModel`.

```
// viewmodel.NoteViewModel.kt
class NoteViewModel(application: NotesApp) : AndroidViewModel(application) {
    // ...

    companion object {

        /**
         * Factory for the [NoteViewModel].
         *
         * @author Emilie Bressoud
         * @author Loïc Herman
         * @author Sacha Butty
         */
        val Factory = viewModelFactory {
            initializer { NoteViewModel(requireNotNull(this[APPLICATION_KEY]) as NotesApp) }
        }
    }
}

// fragments
class /* ... */ : Fragment() {

    private val viewModel: NoteViewModel by activityViewModels { NoteViewModel.Factory }
}

// activité
class MainActivity : AppCompatActivity() {

    private val viewModel: NoteViewModel by viewModels { NoteViewModel.Factory }
}
```

Question 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 ont-elles ? Voyez-vous une autre approche plus adaptée ?*

L'utilisation de `LiveData` pour accéder aux notes stockées dans Room présente effectivement certaines limitations significatives. La principale contrainte réside dans son fonctionnement synchrone sur le thread principal, ce qui peut potentiellement conduire à des blocages ou des ralentissements de l'interface utilisateur lors de requêtes complexes ou volumineuses.

De plus, `LiveData` a été conçu principalement pour le cycle de vie des composants Android, ce qui peut limiter sa flexibilité dans des scénarios plus complexes de manipulation de données. Sa capacité de transformation et de combinaison de flux de données est relativement basique comparée à d'autres solutions.

Enfin, `LiveData` ne propose pas de mécanisme de backpressure, ce qui peut poser des problèmes de performance dans des cas où la source de données produit des données plus rapidement que le consommateur ne peut les traiter, par exemple lors de requêtes réseau ou de calculs intensifs. Cela empêche également la mise en place de patterns réactifs permettant de limiter ou de contrôler le flux de données.

Une approche plus moderne et flexible consisterait à utiliser Kotlin Flow ou RxJava. Ces bibliothèques réactives offrent des avantages considérables :

- Kotlin Flow s'intègre naturellement avec les coroutines Kotlin, permettant une gestion plus efficace des opérations asynchrones.
- Elles proposent des opérateurs puissants pour transformer, filtrer et combiner les flux de données.
- Leur modèle de concurrence est plus sophistiqué, permettant un meilleur contrôle sur les threads d'exécution. Détachant les calculs intensifs du thread principal.
- Elles facilitent la mise en place de patterns réactifs plus avancés comme le backpressure.

En particulier, Kotlin Flow apparaît comme une solution particulièrement adaptée dans le contexte d'une application Android moderne, offrant une intégration naturelle avec les autres composants Kotlin et une syntaxe plus concise.

Question 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 ?*

La première modification à faire sera de rendre les éléments de la liste cliquables. Pour cela, il suffit d'ajouter un `OnClickListener` sur chaque élément de la liste. Dans le cas de notre application, nous devrons modifier l'`Adapter` de la `RecyclerView` et sa classe interne `ViewHolder` pour ajouter en paramètre un `noteClickListener` qui sera ensuite défini come étant le `OnClickListener` de l'élément de la liste.

```kotlin
// fragment.NoteListFragment.kt
private class NotesRecyclerAdapter(
    private val noteClickListener: (NoteAndSchedule) -> Unit,
    initNotes: List<NoteAndSchedule> = emptyList()
) : RecyclerView.Adapter<NotesRecyclerAdapter.ViewHolder>() {

    // [...]

    inner class ViewHolder(
        private val binding: ViewBinding
    ) : RecyclerView.ViewHolder(binding.root) {

        fun bind(note: NoteAndSchedule) {
            binding.root.setOnClickListener { noteClickListener(note) }
            when (binding) {
                is ListNoteViewItemBinding -> bind(binding, note)
                is ListScheduledNoteViewItemBinding -> bind(binding, note)
            }
        }
    }
}
```

Il faudra ensuite modifier la gestion du fragment `NotesListFragment` pour y ajouter la définition du `noteClickListener` qui ouvrira l'écran d'édition de la note sélectionnée. Pour gérer cette navigation, il est possible d'utiliser le `supportFragmentManager` pour remplacer le fragment actuel par un nouveau fragment d'édition de note en ajoutant la transition dans le backstack pour prendre en charge le retour en arrière. Il est également possible, et probablement de manière plus simple et maintenable, d'utiliser la librairie `navigation` de Jetpack pour gérer cette navigation.

Un fragment `NoteEditFragment` devra être créé pour afficher le formulaire d'édition de la note. Il viendra remplacer le fragment `NoteListFragment` dans la vue principale de l'application lors de la sélection d'une note. Ce fragment devra être initialisé avec l'identifiant de la note à éditer, et devra charger les données de la note depuis la base de données Room pour les afficher dans le formulaire d'édition en utilisant le même viewmodel.

Pour rendre l'utilisation sur tablette plus agréable, il est aussi possible de remplacer la navigation spécifiquement pour les tablettes en remplaçant le fragment des contrôles par le fragment d'édition de note au lieu de remplacer la liste.

Une fois que l'utilisateur aura terminé l'édition de la note, il pourra appuyer sur un bouton de sauvegarde qui enverra les modifications à la base de données Room. Une fois les modifications sauvegardées, le fragment d'édition de note pourra être retiré de la vue principale pour revenir à la liste des notes, via une navigation du backstack ou `navigateUp` de la librairie `navigation`.

Certaines modifications seront nécessaire dans le viewmodel tout comme le repository et le DAO pour permettre de mettre à jour une note existante dans la base de données Room avec les nouvelles valeurs.

Choix d'implémentation

Titre et contenu des notes avec horaire
Pour éviter que le texte du titre et du contenu d'une note déborde sur l'icône et le texte de l'horaire, nous avons ajouté une contrainte de largeur à ces deux éléments et activé le mode `ellipsize` pour que le texte soit tronqué s'il dépasse trop.

Affichage de la deadline
Pour nous simplifier la vie, nous utilisons `DateUtils.getRelativeTimeSpanString` pour obtenir le texte d'affichage de la date qui sera automatiquement traduit selon la locale du système. Cet affichage diffère légèrement de celui indiqué en donnée car il affichera la date jusqu'à ce qu'elle soit à moins d'une semaine où un format relatif sera utilisé.

Mode tablette paysage
La densité de pixels d'une tablette est définie comme ayant la largeur la plus courte à `600dp`, nous utilisons donc le sélecteur `sw600dp` pour la version du layout des tablettes. Le layout avec les actions s'applique aussi uniquement aux tablettes en mode paysage, via le sélecteur `land`.

Ces layouts utilisent des fragments qui contiennent le contenu donc le rajout de dispositions peut être fait facilement.

Initialisation de la base de données Room
Avec notre définition spécifique de `Application`, nous instancions via l'extension Kotlin `lazy` une instance de `NotesDatabase` qui est la base de données Room via le pattern singleton. Le créateur de l'instance va également enregistrer un `Callback` avec la méthode `onCreate` qui s'assurera de remplir la base de données lors de la première création avec 10 notes aléatoires. (Cette création ne se fait qu'une seule fois, pour relancer la création il faut supprimer le fichier SQLite via l'option clear storage de Android)

La base de donnée comntient la référence vers le DAO et notre instance d'Application va utiliser ces deux objets pour initialiser le repository de notes en donnant à tous un scope pour les coroutines Kotlin qui sera utilisé pour les opérations sur la base de données.

Diffing asynchrone des notes pour le recycler
Pour permettre la gestion de l'affichage des notes dans la `RecyclerView` de façon plus fluide, nous avons utilisé un `AsyncListDiffer` avec un `NoteDiffCallback` qui fait emploi de l'égalité entre data class fournie par Kotlin. `AsyncListDiffer` décharge le travail de comparaison des notes à un thread exécuté en background permettant ainsi de ne modifier que les vues qui ont été modifiées lors de la réception d'une mise à jour de la part de la base de données.

Cela permet aussi de mieux prendre en charge le cas où il y aurait une grande quantité de données en base et que l'utilisation de `Flow` avec des paramètres de backpressure ajustés ne puisse pas être considérée.