LAB 4 DAA

Auteurs

- Duruz Florian
- Ferreira Silva Sven
- Richard Aurélien

Groupe : DAA_B_12

Classe : DAA 2025-26

Date : 23/11/2025

Implémentation

Concernant la structure des layouts, trois catégories existent:

- le layout par défaut, utilisé pour les smartphones,
- le layout sw600dp-land, spécifiquement pour les tablettes en paysage,
- le layout sw600dp, utilisé pour le passage d'une valeur booléenne (nécessaire afin de configurer l'affichage pour qu'il ne plante pas sur rotation).

Idéalement, nous aurions pu bloquer la rotation (portrait pour téléphone et paysage pour tablette), mais les dernières API Android cherchent une approche d'affichage adaptatif obligatoire.

Concernant le layout par défaut, des ConstraintLayouts sont employés pour la main_activity. Un layout global et un layout nested pour le fragment de la RecyclerView des notes.
Le layout tablette utilise la même approche, à la différence que le layout nested est séparé par une guideline afin de clairement définir les zones des fragments de notes et d'actions.

Le fragment des notes est un layout relatif nous permettant sa réutilisation entre les deux layouts d'activités.
Enfin, les éléments constitutifs de la RecyclerView, les items de notes, sont divisés en deux catégories:

- la catégorie "sans schedule", les affichant avec un fond bleu clair et sans données de planification,
- la catégorie "avec schedule", affichant des données relatives à la planification de la note (deadline).

Pour la partie schedule, un petit algorithme maison (considéré comme plus pratique en termes de granularité fine, spécialement concernant les deadlines située à l'année prochaine, mais dans moins de 12 mois).
Cet algorithme affiche la plus grande, non-zero granularité de l'intervalle de temps séparant aujourd'hui à la deadline.
E.g. : deadline dans 1 an, 5 mois et 2 jours -> affiché comme étant à faire d'ici 1 an.
E.g.2 : deadline dans 5 mois, 24 jours -> affiché comme étant à faire d'ici 5mois.
E.g.4 : deadline aujourd'hui -> affiché comme étant à faire "today".
E.g.5 : deadline dans le passé -> affichage "late" et icône de planification teintée en rouge.

Les granularités supportées sont: year(s), month(s), day(s).

Enfin, le menu de tri, nommé en ressources et code "sort menu", est implémenté de la même manière.
Le menu des actions sur notes, nommé "action menu", est conditionnellement instancié comme menu ou comme fragment dépendamment du layout.

RecyclerView

La RecyclerView suit une implémentation assez classique, avec un DiffCallback et ViewAdapter sur-mesure pour le type de données concerné (Notes).

Initialisation de la base de données ROOM

La base de données `NotesDatabases` est un singleton qui est initialisé avec la méthode lazy de Kotlin, à la première création de la base de données
cette dernière se rempli de 10 éléments aléatoire cette initialisation n'est faite qu'à la création d'une nouvelle base de données et non à chaque ouverture de l'application.
la base de données contient une référence vers la DAO, ainsi notre application va utiliser ces 2 classes pour initialiser le Répertoire de notes.

Utilisation de viewmodel

Afin d'utiliser un viewmodel pour nos données, 2 fichiers furent rajoutés : `NotesViewModel` qui va contenir le viewmodel et `NotesViewModelFactory` qui va contenir la factory pour le viewmodel de notes.

NotesViewModel

Le viewmodel est simple et va implémenter les besoins demandés, càd :

- Récuperation de toutes les notes (ordonées en fct des preferences de l'utilisateur (via la sharedpreference))
- le compte des notes
- la génération d'une note random
- la suppression de toutes les notes

Nous implémentons aussi ici la gestion de l'ordre des tâches (cela centralise grâce au viewmodel)

NotesViewModelFactory

Une simple factory pour instancier le viewmodel avec un `NoteRepository`

Utilisation dans les fragments et activity

Pour l'utiliser, rien de plus simple.

Dans l'activity :

```
    private val notesViewModel: NotesViewModel by viewModels {
        NotesViewModelFactory((application as NotesApplication).noteRepository,
applicationContext)
    }
```

Et dans les fragments :

```
private val notesViewModel: NotesViewModel by activityViewModels {
        NotesViewModelFactory((requireActivity().application as
NotesApplication).noteRepository, requireActivity().application)
}
```

les fragments et l'activity utiliseront donc le même viewmodel !

Réponse 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 oeuvre votre approche.

Nous avons utilisé la fonctionnalité "SharedPreferences" pour sauveguarder l'état du tri entre les sessions. Cette fonctionnalité utilisant une valeur clé-pair permet de garder en mémoire des valeurs simples et isolées ce qui est parfait pour notre cas.
cette fonctionnalité est incluse dans la classe `NotesViewModel`.

```
class NotesViewModel(private val repository: NoteRepository, context : Context) :
ViewModel() {

    private val sharedPreferences =
context.getSharedPreferences("NotePreferences", Context.MODE_PRIVATE)

    private val _currentSortOrder = MutableLiveData<SortOrder>().apply {
        value = SortOrder.entries[sharedPreferences.getInt("sortOrder",
SortOrder.CREATION_DATE.ordinal)]
    }
    fun changeSortOrder(sortOrder: SortOrder) {
        _currentSortOrder.value = sortOrder
        with(sharedPreferences.edit()) {
            putInt("sortOrder", sortOrder.ordinal)
            apply()
        }
    }
    //reste du code
}
```

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 et quelles seraient les approches mieux adaptées ?

La Principal limitation serait l'utilisation du Thread principal qui selon le volume de données,
ou de la complexité des requêtes peut provoquer des ralentissements.
Dans le cas de notre exercice ce n'est pas un soucis mais si il fallait trouver un solution, l'approche KotlinFlow serait la plus appropriée.
KotlinFlow permet:

- Prise en charge de la gestion du Backpressure, utile pour traîter des ensembles de données importants
- Prise en charges des coroutines et gestions des opérations sur plusieurs threads.

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 ?

1. Créer un layout simple nous permettant d'éditer une note avec une classe héritant de `fragment()` pour la lier.
2. Ajouter à chaque élément du viewHolder (de la classe `NotesViewAdapter`) un event `OnClickListener` qui sera appelé lors de la séléction d'un évenement.
   quelque chose dans ce genre :

```
class ViewHolder(view: View, val context: Context) :
RecyclerView.ViewHolder(view) {
    init {
                itemView.setOnClickListener {
                    //logic here
                }
        }
}
```

3. il faudra ajouter au fragment `NotesFragment` une fonction pour ouvrir la fenêtre d'édition pour la note au moment du click
4. il faudra adapter le DAO et Repository pour prendre en compte les modification faites