Rapport - Développement Android

Laboratoire n°4 Architecture MVVM, utilisation d'une base de données Room et d'un RecyclerView

Auteurs


```
- Ouweis Harun
- Amir Mouti
```


1. Mise en place

Dans cette première étape, nous avons utilisé le **template de base fourni pour les laboratoires** comme point de départ.

Ajouts principaux effectués

1. **Ajout des plugins nécessaires dans `build.gradle.kts`** :
   - Les plugins pour Kotlin, Android, et KSP ont été inclus afin de prendre en charge les fonctionnalités nécessaires pour l'architecture MVVM et la base de données Room.
   - Ces plugins permettent notamment la génération automatique de code pour Room via KSP.
2. **Inclusion des dépendances pour Room** :
   - Nous avons ajouté les dépendances `room-runtime`, `room-ktx`, et `room-compiler` pour intégrer la base de données Room, essentielle pour gérer les entités et relations (comme `Note` et `Schedule`).
   - Ces dépendances facilitent les interactions avec la base de données de manière optimisée et moderne.
3. **Inclusion des dépendances pour LiveData et ViewModel** :
   - Les bibliothèques pour le cycle de vie d'Android, telles que `lifecycle-viewmodel-ktx` et `lifecycle-livedata-ktx`, ont été ajoutées pour gérer les données de manière réactive et permettre une séparation claire des responsabilités dans l'application.
4. **Ajout des fichiers vectoriels dans le dossier `drawable`** :
   - Les icônes fournies (`clock.xml`, `family.xml`, `note.xml`, etc.) ont été placées dans le dossier `res/drawable`. Elles seront utilisées ultérieurement pour représenter visuellement les différents types de Notes ou actions dans l'interface utilisateur.
5. **Intégration des fichiers modèles Kotlin** :
   - Les fichiers modèles (`Note.kt`, `NoteAndSchedule.kt`, `Schedule.kt`, `State.kt`, `Type.kt`) ont été ajoutés dans le dossier `src/main/java/ch/heigvd/iict/daa/template/models`. Ces

fichiers définissent les entités, les relations et les énumérations nécessaires à la structure de la base de données Room.

---

2. Conception du squelette de l'Activité et des Fragments

MainActivity

- La **MainActivity** a été définie comme le point d'entrée de l'application.
- Elle utilise **DataBinding** pour lier le layout `activity_main.xml` à l'activité de manière déclarative.
- Un **menu** a été ajouté à l'ActionBar, permettant :
  - Le tri des notes par date de création ou date prévue de réalisation.
  - La création d'une note aléatoire.
  - La suppression de toutes les notes.
- La logique associée aux actions de menu est gérée dans la méthode `onOptionsItemSelected`. Nous appelons directement les méthodes du **ViewModel** pour effectuer les actions correspondantes.

Layouts adaptés aux form factors

- **Smartphone** :
  - Le layout `activity_main.xml` comprend une barre d'outils (Toolbar) et un conteneur de fragment unique (`FragmentContainerView`) pour afficher la liste des notes.
- **Tablette** :
  - Un layout spécifique pour les tablettes inclut deux fragments côte à côte :
    - Un fragment pour la liste des notes.
    - Un fragment pour afficher un compteur et des boutons de contrôle (génération et suppression de notes).

Ces deux layouts garantissent une expérience utilisateur adaptée aux différents form factors.

Fragments

- **NotesFragment** :
  - Utilisé pour afficher la liste des notes à l'aide d'un **RecyclerView**.
  - La logique de tri est gérée dans le **ViewModel**, et les données sont observées via **LiveData**.
  - Implémente un mécanisme de clic sur les éléments pour naviguer vers un écran d'édition (ou remplace un fragment sur tablette).
- **ControlsFragment** (uniquement pour les tablettes) :
  - Contient des boutons pour générer des notes et en supprimer toutes.
  - Affiche un compteur dynamique des notes en temps réel grâce à l'observation du **ViewModel**.

Choix techniques importants

- **RecyclerView** : Utilisé dans le `NotesFragment` pour afficher une liste de notes avec un adaptateur flexible. Deux layouts différents seront utilisés dans une étape ultérieure pour gérer les notes avec ou sans date prévue.

- **Fragments adaptatifs** : En mode paysage ou sur tablette, l'interface utilise plusieurs fragments pour maximiser l'espace disponible.
- **AppBar et ToolBar** : Les titres et les actions globales sont gérés de manière cohérente à travers une barre d'outils unique.

Décisions et simplifications

- La logique liée à l'affichage et à la navigation est concentrée dans les fragments, tandis que la **MainActivity** agit principalement comme un conteneur.
- L'utilisation de `ViewModel` garantit que les données persistent lors des rotations ou des changements de configuration, sans avoir besoin de recharger les données depuis la base.

---

3. Mise en place de la base de données Room

DAO (Data Access Object)

- Un **DAO** a été implémenté pour permettre les interactions avec la base de données via des requêtes SQL adaptées à notre modèle.
- Les fonctionnalités principales incluent :
  - **Lecture** : Récupérer toutes les notes et leurs plannings associés avec `getAllNotes()`.
  - **Écriture** : Ajouter des notes et des plannings avec les méthodes `insert`.
  - **Mise à jour** : Modifier une note via `updateNote`.
  - **Suppression** : Supprimer toutes les notes grâce à `deleteAllNotes`.

Convertisseurs de type

- La classe `Date` dans converter a été utilisée pour convertir les objets `Calendar` en long (millisecondes) afin de les stocker dans Room et de reconvertir ces valeurs en objets `Calendar` lors de la lecture.

Base de données

- Une classe abstraite `NotesDB` a été créée pour définir la structure de notre base Room :
  - **Entités** : `Note` et `Schedule`.
  - **TypeConverters** : Gestion des dates via `DateConverter`.
  - **Singleton** : Implémentée pour garantir une seule instance de la base de données dans l'application.
  - Une méthode `populateDatabase` est utilisée pour insérer des données de test si la base est vide lors de sa création.

Repository

- Un **Repository** centralise les opérations de lecture et écriture, en combinant les fonctionnalités du DAO avec l'utilisation de **Coroutines** pour les appels asynchrones :
  - Génération de notes aléatoires.
  - Modification de notes existantes.
  - Suppression de toutes les notes.

Application

- Une classe `App` a été surchargée pour créer une instance unique du `Repository` et de la base de données.
- Le `Repository` est initialisé au démarrage de l'application, en suivant le principe d'initialisation paresseuse (`lazy`).

Choix techniques importants

- **LiveData** : Utilisé pour observer les données en temps réel dans l'interface utilisateur.
- **Coroutines** : Garantissent que toutes les opérations sur la base de données s'exécutent en arrière-plan, sans bloquer le thread principal.
- **Conformité avec Room** : Toutes les interactions avec la base respectent les bonnes pratiques pour assurer la fiabilité et la maintenabilité du code.

Décisions et simplifications

- Une méthode générique `isEmpty` a été implémentée dans le DAO pour simplifier la vérification de l'état initial de la base.
- L'injection des dépendances via la classe `App` facilite la gestion et l'accès global aux ressources de l'application.

---

4. Conception du ViewModel et intégration à l'Activité et aux Fragments

Conception du ViewModel

- Le fichier `ViewModelNotes` centralise la logique métier et les interactions entre l'interface utilisateur et le repository :
  - **Tri des notes** : Les notes peuvent être triées par date de création ou date prévue grâce à `LiveData`.
  - **Gestion des données en temps réel** : Utilisation de `LiveData` et de `MutableLiveData` pour observer les changements des données.
  - **Persistante de l'ordre de tri** : L'ordre de tri est stocké dans les `SharedPreferences` et appliqué automatiquement lors du démarrage.

Liaison entre ViewModel et les composants

- **MainActivity** :
  - Le ViewModel est lié grâce à `ViewModelProvider` et utilisé pour déclencher les actions du menu (tri, création et suppression des notes).
- **Fragments** :
  - **NotesFragment** :
    - Utilise le ViewModel pour observer et afficher les notes triées via un `RecyclerView`.
    - Gère les clics pour ouvrir l'écran d'édition des notes.
  - **ControlsFragment** :
    - Observe le nombre de notes et met à jour l'affichage en temps réel.
    - Propose des boutons pour générer des notes aléatoires ou tout supprimer.

Choix techniques importants

- **Utilisation de LiveData** :
  - Garantit une gestion réactive et efficace des données, même en cas de rotation de l'écran ou de changements de configuration.
- **Factory personnalisée** :
  - Une factory dédiée (`ViewModelNotesFactory`) est utilisée pour injecter le `Repository` et le contexte dans le ViewModel.

Décisions et simplifications

- Les responsabilités sont bien séparées :
  - Le ViewModel est responsable des données et de la logique métier.
  - Les Fragments se concentrent sur l'affichage et les interactions utilisateur.

---

5. Conception et implémentation de la RecyclerView

Objectif

La **RecyclerView** est utilisée pour afficher la liste des notes, avec une gestion différenciée pour les notes simples et les notes associées à une date de réalisation prévue. L'objectif est de proposer un affichage efficace et fluide des données, adapté au volume et à la diversité des contenus.

Choix d'implémentation

1. **Adapter personnalisé** :
   - Le fichier `NotesAdapter` utilise un **AsyncListDiffer** pour gérer efficacement les mises à jour de la liste et optimiser les performances.
   - L'implémentation repose sur deux layouts distincts pour les notes simples et les notes avec horaire.
2. **DiffUtil** :
   - Un **DiffUtil.ItemCallback** personnalisé compare les éléments pour mettre à jour uniquement les parties nécessaires de la liste.
3. **ViewHolder et binding** :
   - Chaque type de note est associé à un `ViewHolder` qui gère son affichage.
   - La méthode `bind` applique les données du modèle au layout correspondant, avec des méthodes spécifiques pour les notes simples et celles avec horaire.
4. **Gestion des événements de clic** :
   - Le clic sur un élément de la liste est géré via l'interface `OnClickListener`.
   - Cela permet de naviguer vers l'écran d'édition ou de modifier directement une note en fonction des besoins.
5. **Layouts des items** :

- Les fichiers XML utilisent des **ConstraintLayouts** pour garantir une disposition flexible et cohérente des éléments visuels.
- Les icônes et les textes sont adaptés dynamiquement en fonction du type et de l'état de la note.

Décisions techniques

- **Performance** : L'utilisation de **AsyncListDiffer** assure des performances élevées même avec des listes volumineuses.
- **Extensibilité** : La structure en deux types de layouts et un adaptateur générique permet d'ajouter facilement de nouveaux types de données à afficher.
- **Accessibilité** : Les icônes sont accompagnées de descriptions (`contentDescription`) pour garantir une meilleure compatibilité avec les lecteurs d'écran.

---

6. Questions complémentaires

6.1 Sauvegarde du choix de tri après fermeture de l'application

Nous sauvons le choix de l'option de tri avec des SharedPreferences, car ils sont persistent même après la fermeture de l'app et ils sont bien adaptés car il ne s'agit que de se rappeler d'une key value (tri: type de tri).

Pour ce faire, il faut avoir le contexte dans le ViewModel (le ViewModelNotesFactory en a aussi besoin pour le passer au ViewModel à sa création):

```
class ViewModelNotes(private val repository: Repository, context: Context) :
ViewModel()
```

```
class ViewModelNotesFactory(private val repository: Repository, private val
context: Context) : ViewModelProvider.Factory{

    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(ViewModelNotes::class.java)) {
            return ViewModelNotes(repository, context) as T
        }
        throw IllegalArgumentException("Classe ViewModel inconnue")
    }
}
```

On met le type de sort dans un liveData

```
    // SharedPreferences pour stocker l'ordre de tri.
    private val sharedPreferences =
context.getSharedPreferences("NotePreferences", Context.MODE_PRIVATE)

    // LiveData pour gérer les changements de l'ordre de tri.
    private val _sortOrder = MutableLiveData<Sort>().apply {
        value = Sort.values()[sharedPreferences.getInt("sortOrder",
```

```
Sort.NONE.ordinal)]
    }
    private val sortOrder: LiveData<Sort> = _sortOrder
```

Fonction pour sauvegarder le type de tri dans les SharedPreferences

```
    /**
     * Modifie l'ordre de tri et le sauvegarde dans les SharedPreferences.
     */
    fun setSortOrder(sortOrder: Sort) {
        _sortOrder.value = sortOrder
        with(sharedPreferences.edit()) {
            putInt("sortOrder", sortOrder.ordinal)
            apply()
        }
    }
```

6.2 Limites de LiveData pour l'accès à la base de données Room

Les LiveData ne sont pas adaptées aux traitements complexes ou aux données de grande taille. Les mises à jour des LiveData sont faites sur le thread principal, donc les requêtes complexes (par exemple tri ou filtrage avancé) ou les données lourdes peuvent ralentir l'application.

Les observateurs de LiveData sont liés au cycle de vie. Si une partie de votre application (par exemple, un service en arrière-plan) nécessite ces données sans interface utilisateur active, LiveData n'est pas adaptée.

alternative : Kotlin Flow, qui est mieux adapté à des données de grandes tailles, fournit des opérations pour faire les manipulations telles que filtrage/tri plus efficacement et permet de fournir des données hors du contexte du cycle de vie (par exemple, pour des services en arrière-plan).

6.3 Sélection et édition d'une Note dans la RecyclerView

Le fragment hôte du recyclerview pourrait avoir une fonction onClick définie par une interface. Lorsqu'il instancie son adapter, il lui passe une référence this à lui même. L'adapter pourra faire appel à la fonction onClick du fragment hôte pour que le fragment notifie son ViewModel des changements des notes. Dans l'adapter, le ViewHolder définit un onClickListener sur la vue des items qui va appeler la fonction onClick du fragment hôte (en passant sa position pour qu'on sache sur quel item on a cliqué).