PROF : M. Eric Youl

Sujet : Création d'une application Agenda étudiant

Contexte

Vous êtes chargée de concevoir une application mobile Android destinée aux étudiants pour la

gestion personnelle de leur emploi du temps. L'objectif est d'offrir aux utilisateurs la possibilité de

consulter, ajouter, modifier et supprimer les cours et activités de leur semaine universitaire.

1. Introduction et Choix Technologiques Clés

L'application Agenda Étudiant a été développée dans le respect des bonnes pratiques modernes de développement Android, en utilisant l'architecture **MVVM (Model-View-ViewModel)**. Cette approche garantit la maintenabilité, la testabilité et la robustesse du code.

1.1. Architecture MVVM (Model-View-ViewModel)

L'architecture MVVM a été choisie pour séparer clairement les responsabilités :

- **View (Activité & Adapters)** : Gère l'affichage des données et les interactions utilisateur (`MainActivity`, `AddEditCourseActivity`).
- **ViewModel** : Contient la logique d'affichage, expose les données à la `View` via **LiveData** et survit aux changements de configuration (rotation de l'écran).
- **Model/Repository** : Gère la logique métier et l'accès aux données, servant de source unique pour le `ViewModel`.

Voici la documentation technique complète de l'application **Agenda Étudiant**, décrivant sa structure, les choix technologiques et les mécanismes de fonctionnement.

1. Introduction et Choix Technologiques Clés

L'application Agenda Étudiant a été développée dans le respect des bonnes pratiques modernes de développement Android, en utilisant l'architecture **MVVM (Model-View-ViewModel)**. Cette approche garantit la maintenabilité, la testabilité et la robustesse du code.

1.1. Architecture MVVM (Model-View-ViewModel)

L'architecture MVVM a été choisie pour séparer clairement les responsabilités :

- **View (Activité & Adapters)** : Gère l'affichage des données et les interactions utilisateur (`MainActivity`, `AddEditCourseActivity`).
- **ViewModel** : Contient la logique d'affichage, expose les données à la `View` via **LiveData** et survit aux changements de configuration (rotation de l'écran).
- **Model/Repository** : Gère la logique métier et l'accès aux données, servant de source unique pour le `ViewModel`.

1.2. Justification des Outils

Outil/Bibliothèque	Rôle	Justification
Room Persistence Library	Couche d'abstraction pour la base de données SQLite.	Permet une manipulation simple et sécurisée des données locales, validation du SQL à la compilation.
LiveData	Conteneur de données observable et **conscient du cycle de vie**.	Assure que l'interface utilisateur se met à jour automatiquement et en temps réel, sans risque de fuite mémoire.
ViewModel	Gestion du cycle de vie des données de l'UI.	Les données persistent lors de la rotation de l'écran ou d'autres changements de configuration.
Material Design	Composants UI (`CardView`, FAB, `TextInputLayout`).	Fournit une ergonomie soignée et une expérience utilisateur moderne (objectif 7 du projet).
AlarmManager	Planification des événements récurrents.	Permet de déclencher les notifications de rappel de manière fiable et hebdomadaire.

2. Structure de l'Application

Le projet est organisé en plusieurs paquets (packages) pour isoler les différentes couches de l'architecture MVVM :

`model`

Contient les classes de données brutes.

- **`Course.java`** : L'**Entité Room** représentant un cours dans la base de données. Elle inclut les champs (`name`, `professor`, `dayOfWeek`, `startTime`, etc.) et la clé primaire.

db (Data Layer - Abstraction)

Contient les composants Room.

- **`AppDatabase.java`** : La classe singleton étendant `RoomDatabase`. Elle est responsable de la création de la base de données locale.
- **`CourseDao.java`** : L'interface **Data Access Object** qui définit les méthodes d'interaction avec la base (requêtes SQL) : `insert()`, `update()`, `delete()`, `getAllCourses()`, et `searchCourses()`.

`repository`

Sert de pont entre les sources de données (ici, Room uniquement) et le `ViewModel`.

- **`CourseRepository.java`** : Contient des méthodes qui appellent le `CourseDao`. Il est responsable de l'exécution des opérations de base de données (CRUD) sur des **threads d'arrière-plan** (`ExecutorService`) pour éviter de bloquer l'interface utilisateur.

`viewmodel`

Contient la logique d'affichage.

- **`CourseViewModel.java`** : Expose les données (`LiveData<List<Course>>`) à la `MainActivity`. Il reçoit les requêtes de la `View` et les transfère au `Repository`.

`view`

Contient les éléments de l'interface utilisateur.

- **`MainActivity.java`** : La **View** principale, elle observe la `LiveData` de `CourseViewModel` pour afficher la liste via le `RecyclerView`. Elle gère aussi la barre de recherche et les résultats des formulaires (ajout/modification/suppression).
- **`AddEditCourseActivity.java`** : Le formulaire de saisie. Il gère la validation des champs et la planification des **notifications**.
- **`CourseAdapter.java`** : L'adaptateur du `RecyclerView` qui lie les données de chaque `Course` au layout `course_item.xml`.

`utils`

Contient les utilitaires système.

- **`AlarmReceiver.java`** : Un `BroadcastReceiver` qui est déclenché par l'`AlarmManager` et qui construit la **notification locale** à afficher à l'utilisateur.

3. Explication Détaillée du Fonctionnement

3.1. Affichage Réactif (LiveData)

1. **Observation** : Dans `MainActivity`, l'appel `courseViewModel.getAllCourses().observe(this, ...)` établit une connexion constante avec la base de données.
2. **Flux** : La `CourseDao` renvoie une `LiveData<List<Course>>`. Lorsque des données sont modifiées dans la base (via `insert`, `update`, `delete`), Room met à jour la `LiveData`.
3. **Mise à jour UI** : L'observateur dans `MainActivity` est immédiatement notifié et appelle `adapter.setCourses()`, ce qui rafraîchit le `RecyclerView`. Le tri des cours (par jour, puis par heure de début) est géré directement par la requête SQL dans `CourseDao`.

3.2. Cycle CRUD (Création, Lecture, Modification, Suppression)

Le flux passe par le pattern **StartActivityForResult** :

- **Ajout/Modification/Suppression** : Sont initiés dans `AddEditCourseActivity`, mais les opérations d'enregistrement/suppression sont effectuées dans `MainActivity.onActivityResult` après la fermeture du formulaire.
- **Suppression** : `AddEditCourseActivity` utilise un code de résultat spécifique (`DELETE_COURSE_REQUEST_CODE`) et renvoie l'ID du cours. `MainActivity` intercepte ce code, crée un objet `Course` minimaliste avec l'ID, et appelle `courseViewModel.delete()`

3.3. Filtrage et Recherche (Recherche Dynamique)

1. **Déclenchement** : `MainActivity` implémente `SearchView.OnQueryTextListener`. Dès que l'utilisateur entre du texte (`onQueryTextChange`), la méthode `performSearch()` est appelée.
2. **Requête** : `performSearch` appelle `courseViewModel.searchCourses("%" + query + "%")`.

3.4. Système de Notifications de Rappel (Bonus)

1. **Planification** : Dans `AddEditCourseActivity.saveCourse()`, la fonction `scheduleNotification()` est appelée. Elle :
    a. Convertit le jour de la semaine et l'heure de début en une date `Calendar`.
    b. **Décrémente l'heure de 10 minutes** (rappel précoce).
    c. Utilise l'`AlarmManager` avec la méthode **`setRepeating`** et un intervalle de `AlarmManager.INTERVAL_DAY * 7` pour assurer une **répétition hebdomadaire**.
2. **Déclenchement** : À l'heure planifiée chaque semaine, l'`AlarmManager` envoie un *broadcast* à l'`AlarmReceiver`.
3. **Affichage** : L'`AlarmReceiver` reçoit l'intent, extrait les détails du cours (nom, heure) et utilise le `NotificationManager` pour afficher la notification.
4. **Suppression/Modification** : Lorsque le cours est modifié ou supprimé, la fonction **`cancelNotification()`** est appelée en utilisant l'ID du cours. Cela supprime l'alarme récurrente, évitant les rappels obsolètes.