DAA - labo6

DAA_B_9

Bleuer Rémy
Changanaqui Yoann
Rajadurai Thirusan

10.12.2025

Introduction

Ce labo nous fait mettre en pratique les concepts vus en cours sur l'architecture MVVM, la gestion de bases de données locales avec Room, et la communication avec une API REST. Nous sommes partis sur l'implémentation "classique" au lieu de faire `Compose`.

Exercice 2

Choix d'implémentation

Utilisation d'un LiveData nullable dans le `ViewModel`

- Si le `Contact` est à null, c'est qu'il n'y a pas d'édition en cours.
- Si le `Contact` est vide, il faut créer un nouveau contacte.
- Si le `Contact` est présent, il faut l'éditer.
- Permet de survivre aux changements d'états
- Permet d'être facilement observable depuis les `Activity` et les `Fragments`.

`FragmentTransaction` dans `MainActivity`

```
private fun showEditFragment() {
    // Replace fragment
    supportFragmentManager.beginTransaction()
        .replace(R.id.main_content_fragment, EditContactFragment.newInstance())
        .addToBackStack(null) // Can go back to list
        .commit()
}
```

- `.addToBackStack(null)`, permet de revenir en arrière avec le bouton retour.
- `.replace()`, pour éviter d'empiler plusieurs instance d'un meme fragment.

Formulaire d'édition Nous avons utilisé une guideline, afin de pouvoir aligner les label et les inputs facilement. Par conséquent, on utilise aussi un `ConstraintLayout`.

Tests effectués

Test effectué	Résultat attendu	Résultat obtenu
Créer un nouveau contacte	Le nouveau contacte se trouve dans la liste	OK
Suppression d'un contacte existant	Le contacte supprimé n'est plus dans la liste	OK
Édition d'un contacte avec de nouvelles valeurs	Le contacte est bien mis à jour	OK
Clic sur le bouton cancel lors de l'édition	Aucune changement sur le contacte modifié	OK
Valider le formulaire sans le nom	Le formulaire n'est pas validé, il y a une erreur	OK
Rotation de l'écran sur la liste	Aucune perte de donnée	OK
Rotation de l'écran sur l'édition	Les données sont conservées	OK
Clic sur le bouton retour	Retour à la page précédente	OK

Exercice 3

Choix d'implémentation

3 nouveaux champs ont étés ajoutés é l'entité `Contact`. - `remoteId` : Stocke l'id du contact à l'identique du serveur. Si cet id est null, c'est qu'il est seulement dispo en local. Permet d'avoir un lien entre l'id créer par `Room` et l'id créer par le serveur. - `isModifiedLocally` : Même principe que le précédent, s'il est à true, c'est un contacte qu'il faut synchroniser avec le serveur. Évite également de synchroner les contacts déjà à jour. - `isDeletedLocally` : Pareil,

mais avec la suppression. S'il est à false, le contact est toujour valide. Permet de garder une trace du contact, pour pouvoir le supprimer par la suite, si pas de réseau sur le moment par exemple.

Tests effectués

Test effectué	Résultat attendu	Résultat obtenu
Lancer l'app avec le nouveau schéma	L'app crash	OK
Désinstaller et relancer l'app	L'app se lance	OK

Exercice 4

Choix d'implémentation

4.1 Enrollment L'enrollment est déclenché via le menu. Le processus : 1. Suppression de tous les contacts locaux (`clearAllContacts()`) 2. GET sur `/enroll` → récupération d'un nouvel UUID 3. Stockage de l'UUID dans **DataStore** (persistance entre redémarrages) 4. GET sur `/contacts` → récupération des 3 contacts initiaux 5. Insertion en local avec `remoteId` = id serveur et `dirty = false`

Pourquoi DataStore ? : API coroutine-native, évite les problèmes de SharedPreferences sur le main thread.

```
val Context.dataStore: DataStore<Preferences> by preferencesDataStore(name = "uuid")
```

4.2 CRUD avec politique "best-effort" Chaque opération suit le pattern **local-first** :

Création : 1. `insert()` local → récupère l'id local 2. POST `/contacts` avec `id = null` 3. Succès : mise à jour avec `remoteId`, `dirty = false` 4. Échec : reste `dirty = true`

Modification : 1. `update()` local avec `dirty = true` 2. PUT `/contacts/{remoteId}` (si `remoteId` existe) 3. Succès : `dirty = false` 4. Échec : reste `dirty = true`

Suppression : 1. Marquage : `isDeletedLocally = true`, `dirty = true` 2. DELETE `/contacts/{remoteId}` (si `remoteId` existe) 3. Succès : suppression définitive locale 4. Échec : reste marqué pour sync ultérieure

4.3 Synchronisation manuelle La fonction `syncDirtyRead()` récupère tous les contacts avec `dirty = true` et les synchronise selon leur état : - `id == null` → création - `isDeletedLocally` → suppression - sinon → modification

Tests effectués

Test effectué	Résultat attendu	Résultat obtenu
Enrollment et récupération des 3 contacts	3 contacts initiaux affichés	OK
Création d'un contact (online)	Contact créé et synchronisé	OK
Modification d'un contact (online)	Contact modifié sur serveur	OK
Suppression d'un contact (online)	Contact supprimé sur serveur	OK
Création offline puis sync	Contact synchronisé après bouton sync	OK
Modification offline puis sync	Modification propagée au serveur	OK
Suppression offline puis sync	Suppression propagée au serveur	OK
Persistance UUID après redémarrage	UUID conservé, pas besoin de re-enroll	OK
Mode avion pendant opération	Opération locale OK, contact reste dirty	OK

Utilisation d'outils IA

Un assistant IA a été utilisé pour la relecture du code de synchronisation et l'aide à la documentation.

Conclusion

Ce laboratoire nous a permis de mettre en pratique une architecture MVVM complète avec synchronisation REST. La gestion du mode offline via les champs `dirty`, `remoteId` et `isDeletedLocally` permet une expérience utilisateur fluide même sans connexion. L'utilisation de Ktor avec les coroutines Kotlin s'est révélée efficace pour les appels réseau asynchrones.

Améliorations possibles : indicateur visuel pour les contacts non synchronisés, synchronisation automatique en arrière-plan avec WorkManager.