DAA - labo6

> Auteur: Bleuer Rémy, Changanaqui Yoann, Rajadurai Thirusan
> Date: 11.01.2026

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 à 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.
- `dirty` : Il indique que le contacte doit être synchroniser avec le serveur *(à l'aide de KTOR)* a échoué.
- `isDeletedLocally` : Pareil, mais avec la suppression. Si la suppression n'a pas pu avoir lieu sur le serveur alors ce flag est à true. Il faut alors le synchroniser.

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 des contacts récupéré côté serveur en local avec `remoteId` = id serveur et `dirty = true`

Pourquoi DataStore ? : API coroutine-native, évite les problèmes de SharedPreferences sur le main thread vu en cours *(callback)*.

```
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`
   - Succès : mise à jour du contact en local avec `remoteId`, `dirty = false`
   - Échec : reste `dirty = true`

Modification :

1. `update()` local avec `dirty = true`
2. PUT `/contacts/{remoteId}` (si `remoteId` existe)
   - Succès : `dirty = false`
   - 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

> **Note**
>
> Note: `isDeletedLocally` peut-être vu comme optionel car le flag `dirty` est activé. Cependant lors de mauvaise suppression nous pourrions envisager une gestion particulière. C'est pourquoi nous le laissons dans l'entité `Contact`.
>
> A noter également que la sauvegarde en BDD locale s'exécute deux fois. Une au début et une après celle du serveur. Ceci provient de notre gestion de "`dirty` Contact". Ainsi si la communication au serveur échoue, le contact est sauvegardé en locale.

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 *(si traitement particulier éventuel)*
- 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

> **Tldr**
>
> ## 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. Ou synchronisation de contact spécifique.