Laboratoire 6

Application communicante

Christophe Kunzli, Guillaume Trüeb, Gwendal Piemontesi

11 janvier 2026

DAA

Table des matières

1 Introduction 3
2 Implémentation de l'enrollment 4
2.1 Vue d'ensemble 4
2.2 Architecture et workflow 4
3 Opérations CRUD avec synchronisation 5
3.1 Principe de la politique « best-effort » 5
3.2 Préparation des entités 5
3.3 Création d'un contact 5
3.4 Modification d'un contact 6
3.5 Suppression d'un contact 6
3.6 Gestion des modifications successives 7
4 Synchronisation globale 8
4.1 Vue d'ensemble 8
4.2 Architecture 8
4.3 Requêtes DAO 8
4.4 Phase 1 : Synchronisation des suppressions 8
4.5 Phase 2 : Synchronisation des créations/modifications 9
4.6 Vérification de l'UUID 9
5 Utilisation de l'IA 10
6 Conclusion 11

1 Introduction

Ce laboratoire consiste au développement d'une application Android de gestion de contacts avec synchronisation REST. L'application est basée sur l'architecture MVVM et met en œuvre une base de données locale Room synchronisée avec un serveur REST via Retrofit.

L'implémentation couvre trois axes principaux : l'enrollment pour initialiser un utilisateur avec un UUID unique, les opérations CRUD individuelles avec politique de synchronisation « best-effort », et la synchronisation globale de tous les contacts « dirty ». L'application utilise Jetpack Compose pour l'interface, des Coroutines pour l'asynchronisme, et des Flow pour la réactivité des données.

2 Implémentation de l'enrollment

2.1 Vue d'ensemble

L'enrollment permet d'initialiser un nouvel utilisateur en obtenant un UUID unique du serveur et en chargeant 3 contacts par défaut. Cette fonctionnalité est accessible via le bouton :

2.2 Architecture et workflow

L'implémentation suit le pattern MVVM avec trois couches :

- **UI Compose** : Bouton dans TopAppBar + CircularProgressIndicator + Snackbar pour feedback
- **ViewModel** : Méthode `enroll()` qui orchestre via coroutines (`viewModelScope.launch`)
- **Repository** : Logique métier dans `ContactsRepository.enroll()` exécutée sur `Dispatchers.IO`

Le processus se déroule en 5 étapes séquentielles :

1. Nettoyage des données locales

Suppression de toutes les données existantes pour repartir sur une base propre :

```
contactsDao.clearAllContacts()
uuidManager.clearUuid()
```

2. Obtention de l'UUID

Appel GET au endpoint `/enroll` qui retourne un UUID en texte brut (pas JSON) :

```
@GET("enroll")
suspend fun enroll(): Response<String>
```

Pour gérer cette réponse texte, nous avons implémenté un `StringConverterFactory` personnalisé qui doit être ajouté avant le `GsonConverterFactory` dans la configuration Retrofit.

3. Sauvegarde de l'UUID

L'UUID est persisté dans SharedPreferences via la classe `UuidManager`.

Cette sauvegarde garantit la persistance de l'UUID même après redémarrage de l'application.

4. Récupération des contacts

Appel GET au endpoint `/contacts` avec l'UUID dans un header HTTP :

```
@GET("contacts")
suspend fun getAllContacts(@Header("X-UUID") uuid: String): Response<List<ContactDTO>>
```

Le serveur retourne un tableau JSON de 3 contacts. Gson désérialise les dates ISO 8601 en `ContactDTO`.

5. Stockage local

Chaque `ContactDTO` est converti en entité Room `Contact` et inséré en base :

```
contacts.forEach { dto ->
    val contact = ContactDTO.toContact(dto, syncState = SyncState.SYNCED)
    contactsDao.insert(contact)
}
```

L'état `SYNCED` indique que ces contacts proviennent directement du serveur.

3 Opérations CRUD avec synchronisation

3.1 Principe de la politique « best-effort »

Chaque opération CRUD (Create, Read, Update, Delete) suit un processus en deux phases :

1. **Phase locale** : Opération appliquée immédiatement en base Room → visible instantanément dans l'UI
2. **Phase serveur** : Tentative de synchronisation en arrière-plan

Si la synchronisation échoue (pas de réseau, timeout), le contact reste en état « dirty » (TO_SYNC ou TO_DELETE) pour être synchronisé plus tard. Cette approche garantit une UI réactive même en mode hors-ligne.

3.2 Préparation des entités

L'entité `Contact` a été enrichie de trois propriétés pour la synchronisation :

```
@Entity
data class Contact(
    @PrimaryKey(autoGenerate = true) var id: Long? = null,
    var remoteId: Long? = null,
    // ...
    var syncState: SyncState = SyncState.SYNCED
)
```

- **id** : Clé primaire locale auto-générée par Room
- **remoteId** : ID du contact côté serveur (null si jamais synchronisé)
- **syncState** : Enum avec 3 états
  - `SYNCED` : Parfaitement synchronisé
  - `TO_SYNC` : Créé ou modifié localement, pas encore sync
  - `TO_DELETE` : Marqué pour suppression (soft delete)

Un `TypeConverter` Room stocke l'enum comme String en base de données.

3.3 Création d'un contact

Phase 1 - Insertion locale

```
contact.syncState = SyncState.TO_SYNC
val localId = contactsDao.insert(contact)
```

Le contact est immédiatement visible dans l'UI via le Flow Room.

Phase 2 - Synchronisation serveur

```
val dto = ContactDTO.fromContact(contact)
val response = apiService.createContact(uuid, dto)

if (response.isSuccessful && response.body() != null) {
    val createdDto = response.body()!!
    val updatedContact = contact.copy(
        id = localId,
        remoteId = createdDto.id,
        syncState = SyncState.SYNCED
    )
    contactsDao.update(updatedContact)
}
```

Si le POST réussit (201 CREATED), le contact local est mis à jour avec le `remoteId` reçu et passe en état `SYNCED`. En cas d'échec réseau (exception capturée), le contact reste en `TO_SYNC` sans erreur affichée à l'utilisateur.

3.4 Modification d'un contact

Phase 1 - Mise à jour locale

```
contact.syncState = SyncState.TO_SYNC
contactsDao.update(contact)
```

Phase 2 - Synchronisation serveur

```
if (uuid != null && contact.remoteId != null) {
    val dto = ContactDTO.fromContact(contact)
    val response = apiService.updateContact(uuid, contact.remoteId!!, dto)

    if (response.isSuccessful) {
        contact.syncState = SyncState.SYNCED
        contactsDao.update(contact)
    }
}
```

Un PUT est effectué seulement si le contact possède un `remoteId`. Si `remoteId == null`, la modif reste locale et sera créée sur le serveur lors du sync global.

3.5 Suppression d'un contact

Phase 1 - Soft delete

```
@Query("UPDATE Contact SET syncState = 'TO_DELETE' WHERE id = :id")
suspend fun markAsDeleted(id: Long)
```

Le contact est marqué `TO_DELETE` mais pas supprimé physiquement. Le DAO filtre ces contacts de la requête principale :

```
@Query("SELECT * FROM Contact WHERE syncState != 'TO_DELETE'")
fun getAllContacts(): Flow<List<Contact>>
```

Ainsi le contact disparaît visuellement tout en restant en base pour permettre sa suppression serveur.

Phase 2 - Suppression serveur

```
if (uuid != null && contact.remoteId != null) {
    val response = apiService.deleteContact(uuid, contact.remoteId!!)

    if (response.isSuccessful) {
        contactsDao.hardDelete(contact.id!!)
    }
}
```

Si `remoteId == null`, le contact est directement supprimé localement (n'existe pas sur serveur).

3.6 Gestion des modifications successives

Exemple de scénario hors-ligne :

1. Création d'un contact (TO_SYNC, remoteId=null)
2. Mode avion activé
3. Modification du nom
4. Modification du numéro

Chaque modification appelle `updateContact()` qui sauvegarde l'état actuel en local. Lors du sync global, un seul POST sera envoyé avec l'état final du contact. C'est une approche « last-write-wins » qui fonctionne car nous avons un seul client et le serveur ne modifie jamais les données.

4 Synchronisation globale

4.1 Vue d'ensemble

La synchronisation globale rattrape tous les contacts « dirty » après une période hors-ligne prolongée. Accessible via le bouton :

Elle tente de synchroniser tous les contacts TO_SYNC et TO_DELETE en une seule opération batch.

4.2 Architecture

Le ViewModel déclenche `refresh()` qui appelle `repository.syncAll()`.

Le booléen retourné indique si **toutes** les opérations ont réussi (même un seul échec retourne `false`).

4.3 Requêtes DAO

Deux requêtes récupèrent les contacts nécessitant une synchronisation :

```
@Query("SELECT * FROM Contact WHERE syncState = 'TO_SYNC'")
suspend fun getContactsToSync(): List<Contact>

@Query("SELECT * FROM Contact WHERE syncState = 'TO_DELETE'")
suspend fun getContactsToDelete(): List<Contact>
```

4.4 Phase 1 : Synchronisation des suppressions

L'ordre est important : les suppressions **d'abord** pour éviter les conflits.

```
val contactsToDelete = contactsDao.getContactsToDelete()

contactsToDelete.forEach { contact ->
    if (contact.remoteId != null) {
        try {
            val response = apiService.deleteContact(uuid, contact.remoteId!!)
            if (response.isSuccessful) {
                contactsDao.hardDelete(contact.id!!)
                successCount++
            } else {
                failCount++
            }
        } catch (e: Exception) {
            failCount++
        }
    } else {
        // Pas d'ID serveur → suppression locale directe
        contactsDao.hardDelete(contact.id!!)
        successCount++
    }
}
```

Chaque contact est traité indépendamment avec son propre try-catch, permettant de continuer même si un DELETE échoue.

4.5 Phase 2 : Synchronisation des créations/modifications

```
val contactsToSync = contactsDao.getContactsToSync()

contactsToSync.forEach { contact ->
    try {
        val dto = ContactDTO.fromContact(contact)

        if (contact.remoteId == null) {
            // Nouveau contact → POST
            val response = apiService.createContact(uuid, dto)
            if (response.isSuccessful && response.body() != null) {
                val createdDto = response.body()!!
                val updatedContact = contact.copy(
                    remoteId = createdDto.id,
                    syncState = SyncState.SYNCED
                )
                contactsDao.update(updatedContact)
                successCount++
            } else {
                failCount++
            }
        } else {
            // Contact existant → PUT
            val response = apiService.updateContact(uuid, contact.remoteId!!, dto)
            if (response.isSuccessful) {
                contact.syncState = SyncState.SYNCED
                contactsDao.update(contact)
                successCount++
            } else {
                failCount++
            }
        }
    } catch (e: Exception) {
        failCount++
    }
}
```

La méthode retourne `true` uniquement si tous les contacts ont été synchronisés.

4.6 Vérification de l'UUID

Avant toute tentative :

```
val uuid = uuidManager.getUuid()
if (uuid == null) {
    Log.e(TAG, "Cannot sync - no UUID")
    return false
}
```

Sans UUID, impossible de communiquer avec le serveur.

5 Utilisation de l'IA

L'intégration de l'intelligence artificielle dans le processus de développement a permis d'optimiser plusieurs aspects du projet : la refactorisation du code source, la documentation par génération automatique de commentaires, l'accompagnement technique pour les implémentations Android et la clarification de concepts techniques. Ces interventions ont significativement amélioré la qualité structurelle du code et son intelligibilité, tout en accélérant le cycle de développement.

6 Conclusion

Ce laboratoire a permis de développer une application Android complète intégrant communication REST et synchronisation de données. L'architecture MVVM avec Jetpack Compose, Room et Retrofit assure une séparation claire des responsabilités et une maintenabilité optimale.

Ce travail a consolidé notre compréhension des communications réseau asynchrones, de la gestion d'états avec Flow/StateFlow, et des stratégies de synchronisation.