L6 - Rapport

Auteurs : Bugna, Slimani & Steiner

Introduction

L'application de gestion des contacts permet à l'utilisateur de créer, modifier, supprimer et synchroniser des contacts via une base de données locale et un serveur distant. L'architecture adoptée est le modèle **MVVM** (Model-View-ViewModel), facilitant la séparation des responsabilités et la gestion des données. L'application utilise **Room** pour la gestion des contacts en base locale et une **API REST** pour la synchronisation avec un serveur distant.

Architecture de l'application

L'application repose sur l'architecture **MVVM** avec les composants suivants :

1. **Model** : La classe `Contact` qui représente un contact.
2. **ViewModel** : Le `ContactsViewModel` qui gère la logique de l'application, notamment la gestion des données des contacts via le repository.
3. **View** : L'interface utilisateur qui inclut `MainActivity`, `ListFragment` et `EditFragment` pour interagir avec l'utilisateur.

1. Model

Le modèle de données est constitué de la classe `Contact`, qui contient des informations telles que le nom, le prénom, l'email, l'UUID, l'indicateur de synchronisation (`isDirty`), et la date de modification (`lastModified`). Exemple de la classe `Contact` :

```
@Entity
data class Contact(@PrimaryKey(autoGenerate = true)
                   var id: Long? = null,
                   var uuid: String? = null, // Identifiant unique
                   var name: String,
                   var firstname: String?,
                   var birthday : Calendar?,
                   var email: String?,
                   var address: String?,
                   var zip: String?,
                   var city: String?,
                   var type: PhoneType?,
                   var phoneNumber: String?,
                   var isDirty: Boolean = false,
                   var lastModified: Long = System.currentTimeMillis())
```

2. ViewModel

Le ContactsViewModel est responsable de la gestion des contacts. Il interagit avec le repository pour récupérer les contacts depuis la base de données locale ou effectuer des opérations CRUD (création, mise à jour, suppression) via l'API REST. Il gère aussi la logique de synchronisation avec le serveur distant.

3. View

L'interface utilisateur est gérée dans les fragments et activités. La MainActivity sert de point d'entrée et permet à l'utilisateur d'interagir avec l'application via un bouton pour créer un nouveau contact, ou via un menu pour synchroniser les contacts et peupler la base de données avec les données du serveur.

Fonctionnalités de l'application

1. Gestion des Contacts

L'application permet de créer, modifier et supprimer des contacts. Ces opérations sont effectuées localement dans la base de données Room. Lorsqu'un contact est ajouté, modifié ou supprimé, ces changements sont aussi synchronisés avec le serveur via une API REST.

- Création de contact : Un contact est ajouté localement et une tentative de synchronisation est effectuée avec le serveur.
- Mise à jour de contact : Si un contact est modifié, il est mis à jour localement et la modification est synchronisée avec le serveur.
- Suppression de contact : Un contact est supprimé localement et, si possible, cette suppression est aussi effectuée sur le serveur.

2. Synchronisation avec le Serveur

Les contacts marqués comme "dirty" (c'est-à-dire non synchronisés) sont envoyés au serveur pour une mise à jour ou une création. Un UUID unique est utilisé pour identifier l'utilisateur, et les contacts sont envoyés ou récupérés en fonction de ce UUID.

- Synchronisation manuelle : Les contacts marqués comme "dirty" sont synchronisés avec le serveur via l'API REST.
- Inscription : Lors de l'inscription (enroll), un nouvel UUID est récupéré du serveur, et tous les contacts locaux sont effacés pour être remplacés par ceux du serveur.

Détail du Code

1. ContactsRepository.kt

Le ContactsRepository gère toutes les opérations relatives aux contacts. Il interagit avec le DAO de Room pour manipuler la base de données locale et avec l'API pour la gestion des contacts à distance. Méthodes importantes :

- `insert(contact: Contact)` : Insère un contact localement et tente de le synchroniser avec le serveur.
- `update(contact: Contact)` : Met à jour un contact localement et tente de synchroniser les modifications avec le serveur.
- `delete(contact: Contact)` : Supprime un contact localement et tente de le supprimer sur le serveur.
- `synchronizeDirtyContacts()` : Synchronise les contacts marqués comme "dirty" avec le serveur.
- `enroll()` : Récupère un UUID du serveur et synchronise les contacts.

1.1 Synchronisation

La synchronisation des contacts non synchronisés (dirty) suit le flux suivant :

1. **Vérification préalable**
   - Récupération de l'UUID de l'utilisateur
   - Si aucun UUID n'est trouvé, la synchronisation est abandonnée
2. **Identification des contacts à synchroniser**
   - Récupération de tous les contacts marqués comme "dirty"
   - Ces contacts sont traités un par un dans une boucle
3. **Processus de synchronisation pour chaque contact**
   - **Pour un contact existant** (id et uuid non null)
     - Copie du contact avec isDirty = false
     - Mise à jour via PUT vers le serveur
     - En cas de succès, mise à jour du statut dirty en local
   - **Pour un nouveau contact**
     - Création d'une copie sans id et uuid
     - Envoi via POST vers le serveur
     - En cas de succès, mise à jour de l'uuid local avec celui reçu du serveur
4. **Gestion des erreurs**
   - Les erreurs sont gérées individuellement pour chaque contact
   - Un échec de synchronisation d'un contact n'affecte pas les autres
   - Les contacts non synchronisés restent marqués comme "dirty"

Exemple simplifié du flux :

```
```

```
suspend fun synchronizeDirtyContacts() = withContext(Dispatchers.IO) {
    // 1. Vérification de l'UUID
    val uuid = getUuid() ?: return@withContext

    // 2. Récupération des contacts dirty
    val dirtyContacts = getDirtyContacts()

    // 3. Synchronisation de chaque contact
    dirtyContacts.forEach { contact ->
        try {
            if (contact.id != null && contact.uuid != null) {
                // Contact existant -> PUT
                apiService.updateContact(uuid, contact.id, contact.copy(isDirty =
false))
            } else {
                // Nouveau contact -> POST
                val response = apiService.createContact(uuid, contact)
                contact.uuid = response.uuid
            }
            // Marquer comme synchronisé
            contact.isDirty = false
            contactsDao.update(contact)
        } catch (e: Exception) {
            // Le contact reste dirty pour une future synchronisation
            Log.e(TAG, "Erreur de synchronisation", e)
        }
    }
}
```

1.2 Enrollement

L'inscription se déroule en trois étapes principales :

1. Obtention d'un nouvel UUID du serveur
2. Suppression des données locales existantes
3. Récupération et stockage des contacts du serveur

Exemple simplifié du flux :

```
suspend fun enroll() = withContext(Dispatchers.IO) {
    // 1. Obtention et stockage de l'UUID
    val uuid = apiService.enroll().string().trim()
    saveUuid(uuid)  // Stockage dans SharedPreferences

    // 2. Nettoyage local
    deleteAllContacts()

    // 3. Synchronisation avec le serveur
    val serverContacts = apiService.getContacts(uuid)
    serverContacts.forEach { contact ->
                val localContact = contact.copy(
                    id = null,
                    isDirty = false,
```

```
                    lastModified = System.currentTimeMillis()
                )
                contactsDao.insert(localContact)
            }
}
```

Choix des Dispatchers

Les fonctions `synchronizeDirtyContacts()` et `enroll()` utilisent le `Dispatchers.IO` pour effectuer des opérations de lecture et d'écriture sur des ressources de type I/O (comme la base de données ou les requêtes réseau). Ce choix permet de décharger le thread principal (UI) de ces tâches longues et de les exécuter sur un thread dédié aux opérations d'entrée/sortie, garantissant ainsi une bonne réactivité de l'application.

Persistance de l'UUID

L'UUID est essentiel pour l'identification de l'utilisateur et pour la synchronisation des données. Il est stocké dans les **SharedPreferences** afin de survivre aux redémarrages de l'application. Ce stockage persistant permet à l'application de récupérer l'UUID après une fermeture et de maintenir la synchronisation avec le serveur sans nécessiter une nouvelle inscription. Voici un exemple d'utilisation de SharedPreferences pour stocker l'UUID :

```
    private fun saveUuid(uuid: String) {
        sharedPreferences.edit().putString("uuid_key", uuid).apply()
    }

    private fun getUuid(): String? {
        return sharedPreferences.getString("uuid_key", null)
    }
```

Gestion des états "dirty"

1. **Marquage d'un contact comme "dirty"**
   Un contact est marqué comme "dirty" lorsqu'il est créé ou modifié mais n'a pas encore été synchronisé avec le serveur. Cela se fait en mettant à jour le champ `isDirty` du contact à `true`. Par exemple, lors de la modification d'un contact localement, son état "dirty" est activé.
2. **Modifications sur un contact non synchronisé**
   Lorsqu'un contact est modifié localement et marqué comme "dirty", il peut être modifié à nouveau sans que cette modification affecte la synchronisation. Cependant, une fois la synchronisation effectuée, le champ `isDirty` est réinitialisé à `false`.

Interface utilisateur

1. **Implémentation des boutons du menu pour l'enrollment et la synchronisation**
   Les boutons du menu permettent à l'utilisateur de s'inscrire (enroll) et de synchroniser les contacts. L'enrollment récupère un UUID unique du serveur, et la synchronisation permet de synchroniser les contacts locaux marqués comme "dirty" avec le serveur. Ces actions sont accessibles via un menu en haut de l'interface utilisateur.
2. **Réflexion des modifications dans l'UI**
   L'interface utilisateur est mise à jour en temps réel pour refléter les modifications effectuées sur les contacts. Par exemple, lorsqu'un contact est modifié, l'UI est mise à jour immédiatement via un **LiveData** ou un **State** qui observe les changements dans la base de données. Cela permet de garantir que l'utilisateur voit les informations les plus récentes sans avoir besoin de rafraîchir manuellement l'application.