Rapport Laboratoire 6

Auteurs : Rachel Tranchida, Massimo Stefani, Eva Ray

Choix d'implémentation

Utilisation de Volley

Nous avons choisi d'utiliser Volley pour gérer les requêtes HTTP dans notre application, bien que l'utilisation de l'API de base java.net.URL aurait été suffisante compte tenu de la petite ampleur de notre projet. Nous avons fait ce choix car nous voulions explorer des outils modernes offrant une gestion plus simple et efficace des requêtes réseau. En effet, Volley simplifie le traitement des requêtes HTTP grâce à ses fonctionnalités intégrées, comme la mise en cache, le traitement automatique des erreurs et la gestion asynchrone, ce qui réduit le risque d'écritures complexes ou d'erreurs liées aux threads. En adoptant Volley, nous profitons d'une solution robuste tout en nous formant à des pratiques reconnues pour des projets plus ambitieux à l'avenir.

Utilisation de `kotlinx.serialization`

Pour la sérialisation et la désérialisation des données, nous avons choisi d'utiliser la bibliothèque `kotlinx.serialization.json.Json`. Initialement, nous avons envisagé d'utiliser **Gson**, mais nous avons rencontré des problèmes lors de la gestion du champ `birthday` dans nos modèles :

1. **Sérialisation** : Lorsque le champ `birthday` était `null` ou absent, Gson générait parfois des erreurs de conversion.
2. **Désérialisation** : Même après avoir configuré Gson pour ignorer les champs inconnus ou `null`, des conflits subsistaient lors de la conversion en instance `Calendar`.

Ces limitations nous ont poussés à adopter `kotlinx.serialization`, qui offre plusieurs avantages :

- **Compatibilité Kotlin-native** : Elle s'intègre parfaitement avec les classes Kotlin et gère automatiquement les types complexes.
- **Gestion simplifiée des champs optionnels** : La configuration `ignoreUnknownKeys = true` nous a permis de gérer facilement les scénarios où certains champs, comme `birthday`, pouvaient être absents ou `null`.
- **Configuration flexible** : Avec des options comme `coerceInputValues = true`, nous avons pu éviter des erreurs de conversion.

Voici un exemple de configuration de `Json` utilisée dans notre projet :

```
val json = Json {
    ignoreUnknownKeys = true
    coerceInputValues = true
    encodeDefaults = true
}
```

Problèmes avec Gson

Avec plus de temps, nous aurions pu envisager de réimplémenter la gestion des dates avec Gson en définissant un `TypeAdapter` personnalisé pour le champ `birthday`. Cependant, cela aurait nécessité un effort supplémentaire pour gérer des cas spécifiques, que `kotlinx.serialization` a pu résoudre plus simplement.

4.1 Implémentation de l'inscription (enrollment)

Un bouton est ajouté à l'UI pour permettre de lancer le processus d'inscription. Lorsqu'on clique sur ce bouton, la fonction `enroll()` de `ContactsViewModel` est appelée. Cette méthode s'occupe de supprimer toutes les données locales puis d'obtenir un nouvel uuid et de récupérer tous les contacts.
fonction `enroll()` du **ViewModel** :

```
fun enroll() {
        viewModelScope.launch {
            try {
                repository.deleteAllContacts()
                repository.enrollAndFetchContacts()
            } catch(e: Exception) {
                e.printStackTrace()
            }
        }
    }
```

La méthode enroll() repose sur deux fonctions principales du repository : `deleteAllContacts` et `enrollAndFetchContacts`, qui gèrent l'interaction avec la base de données locale et les appels réseau.

- `deleteAllContacts` : Cette fonction supprime toutes les entrées de la table des contacts dans la base de données locale. L'opération est exécutée de manière asynchrone à l'aide des coroutines Kotlin pour éviter de bloquer le thread principal. Elle utilise le contexte `Dispatchers.IO`, qui est adapté aux tâches d'entrée/sortie, comme les opérations sur la base de données. Ces tâches peuvent être longues et bloquer le thread sur lequel elles s'exécutent. En déléguant ces opérations à un pool de threads optimisé pour les I/O via `Dispatchers.IO`, on évite de bloquer le thread principal de l'application.
- `enrollAndFetchContacts` : Cette fonction commence par appeler la méthode `enroll`, qui effectue une requête HTTP vers l'endpoint /enroll pour obtenir un nouvel UUID unique. Cet UUID est ensuite stocké dans les SharedPreferences afin d'être réutilisé lors des prochaines sessions. Lors du démarrage de l'application, cet UUID est récupéré des SharedPreferences pour les requêtes ultérieures, notamment celles adressées à l'endpoint /contacts. Une fois l'uuid récupéré, `enrollAndFetchContacts` continue avec la récupération des contacts en appelant la méthode `getAllContacts`, qui fait une requête vers le endpoint \contacts, en passant l'uuid. Encore une fois, la requête est faite dans une coroutine pour ne pas bloquer le thread principal. Chaque contact est alors ajouté dans la base de donnée local en appelant la méthode `insert` du DAO.

On utilise le `ViewModelScope` pour exécuter la coroutine `enroll()` car les coroutines lancées dans ce scope sont automatiquement annulées lorsque le ViewModel est détruit. Cela évite les fuites de mémoire en s'assurant que les tâches asynchrones ne continuent pas inutilement une fois que le ViewModel est détruit.

En résumé, l'implémentation de l'inscription repose sur des coroutines Kotlin pour effectuer des requêtes réseau et des opérations sur la base de données de manière asynchrone, garantissant que le thread principal reste réactif. Le repository est utilisé pour centraliser l'accès aux données locales et distantes, en utilisant le contexte `Dispatchers.IO` pour les tâches d'entrée/sortie. Enfin, le ViewModelScope est utilisé pour gérer la durée de vie des coroutines, assurant leur annulation automatique lorsque le ViewModel est détruit, afin d'éviter les fuites de mémoire.

1. **Suppression des données locales**
   Lorsque l'utilisateur clique sur le bouton d'inscription, la première étape consiste à appeler `deleteAllContacts()` dans le **repository** :

   ```
   suspend fun deleteAllContacts() = withContext(Dispatchers.IO) {
       contactsDao.clearAllContacts()
       uuid = null
   }
   ```

   - On efface tous les contacts dans la base de données locale (DAO).
   - On réinitialise l'UUID stocké dans les `SharedPreferences` (via `uuid = null`), de sorte que l'application soit contrainte de demander un nouvel UUID au serveur.

2. **Obtention d'un nouvel UUID et récupération des contacts**
   La méthode `enrollAndFetchContacts()` effectue deux opérations principales :

   ```
   suspend fun enrollAndFetchContacts() = withContext(Dispatchers.IO) {
       // 1. Obtenir le nouvel UUID
       val newUuid = suspendCancellableCoroutine { continuation ->
           APIRequest.enroll(
               baseUrl,
               queue,
               onSuccess = continuation::resume,
               onError = { error ->
   continuation.resumeWithException(Exception(error)) }
           )
       }
       uuid = newUuid

       // 2. Récupérer les contacts auprès du serveur
       val contacts = suspendCancellableCoroutine { continuation ->
           APIRequest.getAllContacts(
               baseUrl,
               newUuid,
               queue,
               onSuccess = { jsonArray ->
   ```

```
                val contactsString = jsonArray.toString()
                val contacts = json.decodeFromString<List<Contact>>
(contactsString)
                continuation.resume(contacts)
            },
            onError = { error ->
continuation.resumeWithException(Exception(error)) }
        )
    }

    // 3. Mettre à jour la base de données locale
    contactsDao.clearAllContacts()
    contacts.forEach { contact ->
        contact.apply {
            remoteId = id
            state = ContactState.SYNCED
        }
        contactsDao.insert(contact)
    }
}
```

- **1. Obtenir un nouvel UUID** : L'application envoie une requête à l'endpoint `/enroll` via la méthode `APIRequest.enroll(...).` En cas de succès, le serveur renvoie un identifiant unique (`UUID`) qui est ensuite sauvegardé dans les `SharedPreferences`.
- **2. Récupérer les contacts** : Avec ce nouvel UUID, l'application demande la liste de tous les contacts au serveur (`APIRequest.getAllContacts(...)`). L'utilisation de `suspendCancellableCoroutine` permet d'intégrer harmonieusement l'appel asynchrone de Volley dans une coroutine Kotlin.
- **3. Mise à jour de la base de données** : Après avoir reçu la liste de contacts, on vide la table locale (sécurité pour éviter les doublons), puis on insère les nouveaux contacts, tout en définissant leur `state` à `SYNCED` et leur `remoteId` (identifiant issu du serveur).

3. **Utilisation de coroutines et de ViewModelScope**

- En exécutant tout cela dans `viewModelScope.launch`, on s'assure que ces opérations longues (requêtes réseau, opérations en base de données) ne bloquent pas l'UI.
- En cas de fermeture ou de destruction du `ViewModel`, la coroutine sera automatiquement annulée, évitant des potentiels risques de fuite de mémoire ou d'opérations inutiles.

Scénario d'utilisation

1. L'utilisateur clique sur le bouton d'inscription.
2. L'application supprime toutes les données de contact de la base de données locale.
3. L'application envoie une requête GET à /enroll pour obtenir un nouvel UUID.
4. L'application stocke le nouvel UUID dans les SharedPreferences.
5. L'application envoie une requête GET à /contacts avec le nouvel UUID.
6. Le serveur renvoie une liste de contacts.
7. L'application insère les contacts dans la base de données locale.

8. L'application affiche les contacts à l'utilisateur.
9. Au prochain lancement de l'application, l'UUID est récupéré des SharedPreferences et utilisé pour les requêtes vers /contacts.

4.2 Création, modification et suppression de contacts

Les opérations CRUD (Create, Read, Update, Delete) sont gérées via des méthodes dédiées du **repository** (`insert`, `update`, `delete`). Le mécanisme se base sur un **état** local (`ContactState`) pour indiquer si le contact est déjà synchronisé ou non.

Gestion des états des contacts

```
enum class ContactState {
    SYNCED,   // Le contact est synchronisé avec le serveur
    CREATED,  // Le contact a été créé localement mais pas encore synchronisé
    UPDATED,  // Le contact a été modifié localement mais pas encore
synchronisé
    DELETED   // Le contact est marqué pour suppression (soft delete)
}
```

- **SYNCED** : le contact est parfaitement synchronisé avec le serveur. *(Affichage vert)*
- **CREATED** : le contact a été créé localement et attend d'être envoyé sur le serveur. *(Bleu)*
- **UPDATED** : le contact existe sur le serveur mais a été modifié localement. *(Orange)*
- **DELETED** : le contact est marqué localement pour être supprimé sur le serveur. *(Rouge)*

Dans l'**adapter** (non montré ici en détail), chaque contact est coloré en fonction de son `state`, pour indiquer visuellement à l'utilisateur quelle est la situation de chaque enregistrement.

Création de contact

L'ajout d'un contact utilise la méthode `insert(contact)` :

```
suspend fun insert(contact: Contact) = withContext(Dispatchers.IO) {
    val currentUuid = uuid ?: throw Exception("No UUID available")
    val jsonContact = json.encodeToString(Contact.serializer(), contact)

    try {
        // On tente l'insertion sur le serveur
        val jsonResponse = suspendCancellableCoroutine { continuation ->
            APIRequest.addContact(
                baseUrl,
                currentUuid,
                JSONObject(jsonContact),
                queue,
                onSuccess = continuation::resume,
                onError = { error ->
continuation.resumeWithException(Exception(error)) }
            )
```

```
        }
        // Si l'insertion serveur réussit, on met à jour l'objet
        val createdContact =
            json.decodeFromString(Contact.serializer(),
jsonResponse.toString()).apply {
                remoteId = id
                state = ContactState.SYNCED
            }
        // Mise à jour ou insertion dans la base de données locale
        if (contact.id != null) {
            contactsDao.update(createdContact)
        } else {
            contactsDao.insert(createdContact)
        }
    } catch (e: Exception) {
        // Si l'opération échoue côté serveur, le contact est marqué comme
CREATED
        contact.state = ContactState.CREATED
        contactsDao.insert(contact)
    }
}
```

1. On encode le contact au format JSON (`json.encodeToString(...)`) avant de l'envoyer.
2. Si le serveur répond positivement, on récupère le contact créé (avec son `id`), on le marque comme `SYNCED` et on l'insère ou met à jour localement.
3. En cas d'erreur serveur ou réseau, le contact reste dans l'application avec l'état `CREATED`. Un appel ultérieur à la synchronisation tentera de l'envoyer à nouveau.

Modification de contact

La modification d'un contact se fait via `update(contact)` :

```
suspend fun update(contact: Contact) = withContext(Dispatchers.IO) {
    val currentUuid = uuid ?: throw Exception("No UUID available")
    var contactToUpdate = contact

    try {
        if (contact.remoteId != null) {
            val jsonContact = json.encodeToString(Contact.serializer(),
contact)
            val jsonResponse = suspendCancellableCoroutine<JSONObject> {
continuation ->
                APIRequest.updateContact(
                    baseUrl,
                    currentUuid,
                    contact.remoteId!!.toInt(),
                    JSONObject(jsonContact),
                    queue,
                    onSuccess = continuation::resume,
                    onError = { error ->
```

```
continuation.resumeWithException(Exception(error)) }
                )
            }
            // Mise à jour réussie sur le serveur
            val updatedContact =
                json.decodeFromString(Contact.serializer(),
jsonResponse.toString()).apply {
                    remoteId = this.id
                    state = ContactState.SYNCED
                }
            contactToUpdate = updatedContact
        } else {
            // Si le contact n'existe pas encore sur le serveur,
            // on le marque seulement comme UPDATED (sauf s'il était CREATED).
            if (contact.state != ContactState.CREATED) {
                contact.state = ContactState.UPDATED
            }
        }
    } catch (e: Exception) {
        // En cas d'échec, on force l'état UPDATED si le contact était SYNCED
        if (contact.state == ContactState.SYNCED) {
            contact.state = ContactState.UPDATED
        }
    }
    // Mise à jour dans la base de données locale
    contactsDao.update(contactToUpdate)
}
```

- Si `remoteId` est défini, on tente la mise à jour sur le serveur.
- Si tout se passe bien, le contact est ramené en local avec l'état `SYNCED`.
- Si le contact n'a pas encore de `remoteId`, il est *uniquement* mis à jour en local (état `UPDATED`), jusqu'à la prochaine synchronisation.

Suppression de contact (Soft Delete)

La méthode `delete(contact)` fonctionne selon que le contact est déjà connu du serveur ou non :

```
suspend fun delete(contact: Contact) = withContext(Dispatchers.IO) {
    val currentUuid = uuid ?: throw Exception("No UUID available")

    try {
        if (contact.remoteId != null) {
            // Tenter la suppression côté serveur
            try {
                suspendCancellableCoroutine { continuation ->
                    APIRequest.deleteContact(
                        baseUrl,
                        currentUuid,
                        contact.remoteId!!.toInt(),
                        queue,
```

```
                        onSuccess = { continuation.resume(Unit) },
                        onError = { error ->
continuation.resumeWithException(Exception(error)) }
                    )
                }
                // Si succès, on supprime le contact localement
                contactsDao.delete(contact)
            } catch (e: Exception) {
                // Soft delete : on marque comme DELETED en cas d'échec de la
requête
                contact.state = ContactState.DELETED
                contactsDao.update(contact)
            }
        } else {
            // Le contact n'existe pas (encore) sur le serveur
            when (contact.state) {
                ContactState.CREATED -> contactsDao.delete(contact) // jamais
envoyé, on le retire
                else -> {
                    // Si déjà marqué UPDATED ou SYNCED, on applique un soft
delete
                    contact.state = ContactState.DELETED
                    contactsDao.update(contact)
                }
            }
        }
    } catch (e: Exception) {
        contact.state = ContactState.DELETED
        contactsDao.update(contact)
    }
}
```

- **Soft delete** : Tant que la suppression n'a pas abouti côté serveur, on garde le contact en base locale avec l'état `DELETED`. L'utilisateur le voit en rouge, et la prochaine synchronisation réessaiera la suppression.

4.3 Synchronisation de tous les contacts

Nous avons introduit une méthode `refresh()` dans `ContactBiewModel`

```
 fun refresh() {
        viewModelScope.launch {
            try {
                repository.synchronizeAllContacts()
            } catch(e: Exception) {

                e.printStackTrace()
            }
        }
    }
```

Lorsque l'on appuie sur le bouton de syncronisation cette fonction est appelée. Elle lance une coroutine qui va appeler la fonction du repository permettant de syncroniser les contacts. La fonction `syncronizeAllContacts()` est une fonction supsensive qui s'exécute dans le contexte Dispatchers.IO car c'est le contexte adapté aux opérations IO et va permettre de ne pas bloquer l'UI-thread.

```
suspend fun synchronizeAllContacts() = withContext(Dispatchers.IO) {
        val currentUuid = uuid ?: throw Exception("No UUID available")
        val unsyncedContacts = contactsDao.getAllContacts().filter {
            it.state != ContactState.SYNCED
        }

        unsyncedContacts.forEach { contact ->
            try {
                when (contact.state) {
                    ContactState.CREATED -> insert(contact)
                    ContactState.UPDATED -> update(contact)
                    ContactState.DELETED -> delete(contact)
                    else -> { /* Ignore SYNCED contacts */ }
                }
            } catch (e: Exception) {
                e.printStackTrace()
            }
        }
    }
```

La fonction va syncroniser uniquement les chagements qui n'ont pas été syncronisés en effectuant les insert, update et delete nécessaires. En cas d'un erreur lors de insert, update or delete, elle catche l'exception et continue à syncroniser le reste des contacts non syncronisés.

1. On récupère tous les contacts qui ne sont pas **SYNCED** : `CREATED`, `UPDATED` ou `DELETED`.
2. Pour chaque contact, on appelle la méthode appropriée (`insert`, `update`, ou `delete`) afin de faire correspondre l'état local avec l'état du serveur.
3. **Gestion des exceptions** : en cas de problème (réseau, serveur indisponible, etc.), l'opération échoue pour ce contact précis, mais la synchronisation continue pour les autres. Le contact restera dans son état non synchronisé, prêt à être réessayé ultérieurement.

Grâce à cette approche :

- Les contacts créés localement sont finalement envoyés vers le serveur.
- Les modifications sont répercutées.
- Les suppressions sont finalisées.

Le tout est effectué dans un **contexte IO** (`Dispatchers.IO`) pour ne pas bloquer le fil d'exécution principal, gardant ainsi l'interface utilisateur fluide. Les coroutines permettent également d'annuler proprement la synchronisation si l'utilisateur quitte l'écran ou si le **ViewModel** est détruit, évitant des fuites de ressources.