Rapport de Projet Android : Labo 6

Auteurs

- Amir Mouti, Ouweis Harun

Introduction

Ce projet consiste en la mise en place d'une application Android communicante basée sur l'architecture MVVM. L'objectif principal est de permettre la gestion locale et la synchronisation des contacts avec un serveur REST. Ce rapport décrit brièvement les différentes étapes réalisées jusqu'à présent.

Partie 1 : Mise en place de l'interface utilisateur

Objectif

Créer une interface utilisateur permettant la visualisation, la création et l'édition des contacts dans une base de données locale.

Actions Réalisées

1. Mise en place de deux écrans principaux avec **Jetpack Compose** :
    - Un écran pour la liste des contacts.
    - Un écran pour la création et l'édition des contacts.
2. Intégration des champs obligatoires et facultatifs pour la saisie des informations d'un contact.
3. Gestion des interactions utilisateur (sauvegarde, annulation, navigation entre les écrans).
4. Utilisation de `LiveData` pour synchroniser l'état de l'interface avec le **ViewModel**.

Fonctionnalités Bonus Ajoutées

1. **Validation stricte du format suisse pour le numéro de téléphone** :
    - Mise en place d'une expression régulière pour vérifier automatiquement que le numéro respecte le format suisse.
    - Blocage de la sauvegarde en cas de format invalide.
2. **Amélioration de l'interface avec des indicateurs visuels** :
    - Placeholder dans le champ du numéro de téléphone pour aider l'utilisateur à saisir les données correctement.
    - Message d'erreur dynamique indiquant la raison de l'échec (par exemple : "Format du numéro invalide").
3. **Support complet pour le mode horizontal** :
    - Ajout d'un défilement vertical permettant d'accéder à l'intégralité du formulaire, même en mode paysage.

Résultat

L'application offre une interface fonctionnelle pour afficher et éditer des contacts. Les modifications sont reliées au ViewModel, mais la connexion avec la base de données locale et le serveur n'était pas encore finalisée.

Partie 2 : Préparation des entités pour la synchronisation

Objectif

Ajouter les champs nécessaires dans l'entité `Contact` pour permettre une synchronisation efficace avec le serveur et valider leur fonctionnement à travers des tests unitaires.

Actions Réalisées

1. **Modification de l'entité `Contact`** :
    - Ajout du champ **`serverId`** pour identifier un contact synchronisé sur le serveur.
    - Ajout du champ **`syncState`** pour suivre l'état de synchronisation local :
        - `CREATED` : Contact créé localement mais non encore synchronisé.
        - `UPDATED` : Contact modifié localement, en attente de synchronisation.
        - `DELETED` : Contact marqué pour suppression sur le serveur.
        - `SYNCED` : Contact synchronisé avec succès avec le serveur.
2. **Fonctions de conversion** :
    - Implémentation de **`toNetworkModel`** et **`toDomainModel`** pour convertir entre le modèle local (`Contact`) et le modèle distant (`ContactNetworkModel`).
3. **Mise à jour du DAO** :
    - Adaptation des requêtes pour gérer les nouveaux champs.
    - Ajout de requêtes spécifiques, comme la récupération des contacts à synchroniser.
4. **Création de tests unitaires** :
    - Développement d'un fichier de test unitaire `ContactEntityTest` pour valider les conversions entre `Contact` et `ContactNetworkModel`.
    - Mise en place des assertions pour vérifier la cohérence des données entre les deux modèles.

    Exemple des tests réalisés

    - Test de conversion d'un contact local (`Contact`) vers un modèle réseau (`ContactNetworkModel`).
    - Test de conversion inverse (`ContactNetworkModel` vers `Contact`).

    Les tests vérifient les points suivants :

    - La cohérence des données entre les deux modèles.
    - L'intégrité des champs `serverId`, `birthday`, `syncState`, et autres informations.
5. **Validation complète** :

- Tous les tests unitaires ont été exécutés avec succès, confirmant que les conversions et les champs ajoutés sont fonctionnels et cohérents.

Résultat

L'entité `Contact` est désormais prête pour la synchronisation avec le serveur. Les tests unitaires valident les ajouts, garantissant une interaction fiable entre l'application et le serveur.

Partie 3 : Implémentation de l'enrollment

Objectif

Mettre en place le mécanisme d'enrollment pour :

1. Obtenir un **UUID** unique représentant l'utilisateur et son jeu de données sur le serveur.
2. Supprimer les données locales existantes.
3. Récupérer les contacts associés à cet UUID depuis le serveur et les stocker localement.

Actions Réalisées

1. Création d'une API `EnrollmentApiService` avec Retrofit pour interagir avec les endpoints REST du serveur.
   - Endpoint `GET /enroll` pour obtenir un UUID.
   - Endpoint `GET /contacts` pour récupérer les contacts.
2. Ajout de la méthode `enroll` dans le **Repository** pour :
   - Envoyer une requête au serveur pour obtenir un UUID.
   - Effacer tous les contacts locaux.
   - Récupérer et insérer les contacts depuis le serveur dans la base de données locale.
3. Intégration avec le **ViewModel** pour permettre à l'utilisateur de déclencher l'enrollment via un bouton.
4. Ajout de logs pour suivre chaque étape du processus.

Résultat

L'enrollment fonctionne correctement. Les logs confirment les étapes suivantes :

1. Obtention d'un UUID.
2. Effacement des données locales.
3. Récupération et insertion des contacts distants.

Partie 4 : Implémentation des opérations CRUD avec synchronisation partielle

Objectif

Développer les fonctionnalités de création, modification et suppression de contacts avec une synchronisation partielle et une gestion des états locaux pour les contacts.

Approche et mise en œuvre

1. **Création d’un contact :**
    - **Étape locale :**
        - Un contact est inséré dans la base locale avec l'état `CREATED`.
    - **Tentative de synchronisation :**
        - Une requête `POST` est envoyée au serveur pour créer le contact.
        - Si la requête réussit, le `serverId` est mis à jour, et l'état du contact passe à `SYNCED`.
        - En cas d'échec, l'état reste `CREATED` pour permettre une synchronisation ultérieure.
2. **Modification d’un contact :**
    - **Étape locale :**
        - Un contact est mis à jour localement avec l'état `UPDATED`.
    - **Tentative de synchronisation :**
        - Une requête `PUT` est envoyée au serveur pour mettre à jour le contact.
        - Si la synchronisation réussit, l'état passe à `SYNCED`.
        - En cas d'échec, l'état reste `UPDATED`.
3. **Suppression d’un contact :**
    - **Étape locale :**
        - Le contact est marqué comme `DELETED` dans la base locale.
    - **Tentative de synchronisation :**
        - Une requête `DELETE` est envoyée au serveur pour supprimer le contact.
        - Si la requête réussit, le contact est supprimé de la base locale.
        - En cas d'échec, le contact reste avec l'état `DELETED`.
4. **Gestion des erreurs :**
    - Si la synchronisation échoue (par exemple, en cas de déconnexion réseau), le contact conserve son état actuel (`CREATED`, `UPDATED`, `DELETED`).
    - Ces états permettent une synchronisation future lorsque les conditions sont rétablies.

Résultat

Les opérations CRUD fonctionnent correctement avec une gestion des états locale et des tentatives de synchronisation réseau. Les logs permettent de suivre précisément chaque étape et de détecter les éventuels problèmes.

Partie 5 : Synchronisation complète des contacts

Objectif

Mettre en place une fonctionnalité de synchronisation globale pour garantir que toutes les modifications locales soient reflétées sur le serveur, et inversement.

Approche et mise en œuvre

1. **Synchronisation des contacts locaux :**

- La méthode `syncAllContacts` parcourt tous les contacts en état `CREATED`, `UPDATED`, ou `DELETED` :
  - **Pour les contacts `CREATED` et `UPDATED` :**
    - Une tentative de sauvegarde (`saveContact`) est effectuée via les endpoints REST (`POST` ou `PUT`).
  - **Pour les contacts `DELETED` :**
    - Une tentative de suppression (`deleteContact`) est effectuée via le endpoint REST (`DELETE`).

2. **Récupération des données distantes :**
   - Une requête `GET` est envoyée au serveur pour récupérer tous les contacts associés à l'UUID de l'utilisateur.
   - Les contacts récupérés sont insérés ou mis à jour localement avec l'état `SYNCED`.
3. **Gestion des conflits :**
   - Si un contact local est en état `DELETED` mais existe toujours sur le serveur, il n'est pas récupéré.
   - Les contacts en état `SYNCED` sont ignorés pour éviter des opérations inutiles.
4. **Résilience face aux pannes :**
   - En cas de perte de connexion, la synchronisation partielle est utilisée comme mécanisme de secours.
   - Les logs permettent de détecter les erreurs et de planifier des actions correctives.

Résultat

La synchronisation complète garantit que toutes les modifications locales sont répercutées sur le serveur, et que les données distantes sont correctement reflétées en local. Le système est robuste face aux pannes temporaires de connexion.