Compte rendu PMR séquence 3

Introduction :

L'application ToDoListe permet à plusieurs utilisateurs de stocker des listes personnelles de tâches à effectuer. Dans le TP précédent, nous faisions appel à une base de données distantes via une API. Si le réseau vient à manquer, l'application devient inutilisable. Nous allons donc implémenter un cache qui permet de sauvegarder temporairement les données en local. J'ai fait le choix de faire appel à la dépendance Room qui fait partie d'Android Jetpack. Cette dépendance permet de sauvegarder des données dans le cache sous forme de base de données (BdD).

Analyse :

Chaque partie est à lire en complément avec le code commenté.

Détection du réseau :

Pour savoir si l'on travaille sur le cache, on a besoin de vérifier constamment si le réseau est disponible. Si cela n'est plus le cas, on travaille avec le cache et l'on masque les boutons d'ajouts.

J'ai choisi d'implémenter un Broadcast Receiver dans chaque Activité pour cela. Je pense qu'il existe une autre façon de faire plus propre mais je n'ai pas trouvé laquelle.

Gestion des Threads :

Les appels à la BdD demandant un certain temps d'exécution, il est recommandé d'éviter de les faire dans le MainThread. Pour cela, j'ai choisi d'utiliser des executors pour éviter de faire les appels à la BdD sur le MainThread. Les appels à l'API se font déjà sur un autre Thread par la fonction .enqueue() de la dépendance Retrofit.

Gestion de la persistance de données :

Puisque lors du dernier TP, j'avais choisi de ne pas créer de classe spécifique pour l'API, j'ai été forcé d'en recréer pour la BdD. En effet il n'est pas possible de placer des List<> dans la BdD. Dans l'idéal, il aurait fallu créer 3 classes distinctes : une pour le modèle de l'application, une pour l'API et une pour la BdD.

C'est ainsi que j'ai créée les classes ProfilToDoDb (table "profils), ListeToDoDb (table "listes") et temToDoDb (table "items"), simples restranscriptions de ProfilToDo, ListeToDo et ItemToDo, pour rendre les données compatibles avec une base de donnée. De plus, j'ai réutilisé le hash de ProfilToDo comme PrimaryKey pour ProfilToDoDb, et j'ai rajouté l'attribut idListe pour ItemToDoDb qui permet de savoir à quel liste appartient tel Item.

Enfin, pour faire la liaison entre les classes Standard et les classes Db, j'ai créé une classe Converter qui permet de convertir dans les deux classes.

ProfilToDoDAO() :
Cette interface permet d'effectuer les requêtes vers la BdD qui concernent la table "profils". Deux requêtes sont implémentées, l'une qui permet d'appeler un profil qui a les bons pseudos et passwords, l'autre qui permet de sauvegarder les profils.

ListeToDoDAO() :
Cette interface permet d'effectuer les requêtes vers la BdD qui concernent la table "listes". Deux requêtes sont écrites : pour récupérer toutes les listes, et pour sauvegarder les listes dans la base de donnée.

ItemToDoDAO() :
Cette interface permet d'effectuer les requêtes vers la BdD qui concernent la table "items". Ainsi on peut demander tous les items (getAll), ceux qui ont un certain idListe (getAllItems), on peut sauvegarder une ArrayList ou un élément d'Item (save), et on peut modifier l'état d'un item (update).

RoomListeToDoDb :
La classe de la base de données de lecture. Si cela n'a pas été déjà effectuée, la base de donnée est créée. On le détecte au moyen d'un singleton. De plus, elle instancie les interfaces citées précédemment.

RoomListeModifieeDb :
La classe de la base de données d'écriture. Quasiment identique à RoomListeToDoDb, cette base de donnée enregistre tous les changements effectués dans le cache. Lorsque l'on reçoit de nouveau le réseau, on lit cette base de donnée pour effectuer toutes les requêtes POST/PUT vers l'API, puis une fois que toutes les requêtes ont réussies, on vide cette base de donnée.

Modification de l'API à partir du cache :

J'ai choisi de créer une classe Synchron qui permettra à terme d'envoyer toutes les informations de RoomListeModifieeDb. Pour l'instant, on n'utilise qu'une méthode de Synchron, SyncItemsToAPI, utilisée dans le cas où des Items sont cochés. À terme, il sera possible d'utiliser SyncAllToAPI pour synchroniser des nouvelles listes/items ou des changements de noms ou des suppressions…

Au sein des activités :

SyncFromAPI() : permet de récupérer les données de l'API et d'afficher les informations.

SyncFromBDD() : permet de récupérer les données de la BdD et d'afficher les informations.

SyncToBDD() : permet de sauvegarder les données dans le cache. À chaque fois que SyncFromApi() est appelée, cette méthode est donc appelée.

Conclusion :

L’application implémente désormais un mode hors connexion qui permet de travailler avec un cache. Pour l’instant, il n’est pas possible d’ajouter d’éléments sans connexion. Cependant, cette implémentation serait assez simple, la méthode étant la même que la modification d’état d’un Item : on ajoute les nouveaux éléments dans la BdD RoomListeModifieeDb, puis on effectue les requêtes une fois reconnecté et on vide la BdD grêce à la classe Synchron.

Je n’ai pas beaucoup eu le temps de modifier voire d’améliorer mon code entre chaque séance. Je n’ai donc pas solutionné les problèmes de la bonne manière ou utilisé les bonnes pratiques. Cela est dommage car cela m’a beaucoup ralenti. En 3 TP, le code de mon application est devenu très peu lisible, et ressemble de plus en plus à un sac de nœud. J’ai ainsi passé beaucoup de temps à terminer ce TP. Ainsi, ce TP m’aura montré l’importance des bonnes pratiques.

Perspectives :

Refactor le code dans le but de le simplifier/segmenter. Chaque activité fait au moins 300 lignes ce qui parait trop long.

Implémenter la possibilité de supprimer ou renommer des éléments lors d’un appui long.

Finir l’implémentation du cache avec l’ajout d’éléments.

Bibliographie :

Utilisation de Room : https://developer.android.com/training/data-storage/room/index.html

Utilisation d’Executors : http://codetheory.in/android-java-executor-framework/