# Compte rendu

# TodoList V3

Maximilien Grattepanche

Le but de ce TEA était de pouvoir sauvegarder les informations dans une base de données elle-même sauvegardée au sein d'un cache, accessible en hors ligne par l'application. Les modifications portées sur les checkbox des items doivent être sauvegardés puis transmises à l'API une fois la connexion au réseau retrouvée.

La mise en place de cette base de données SQLite au sein du fichier cache de l'application se fait grâce à la libraire Room.

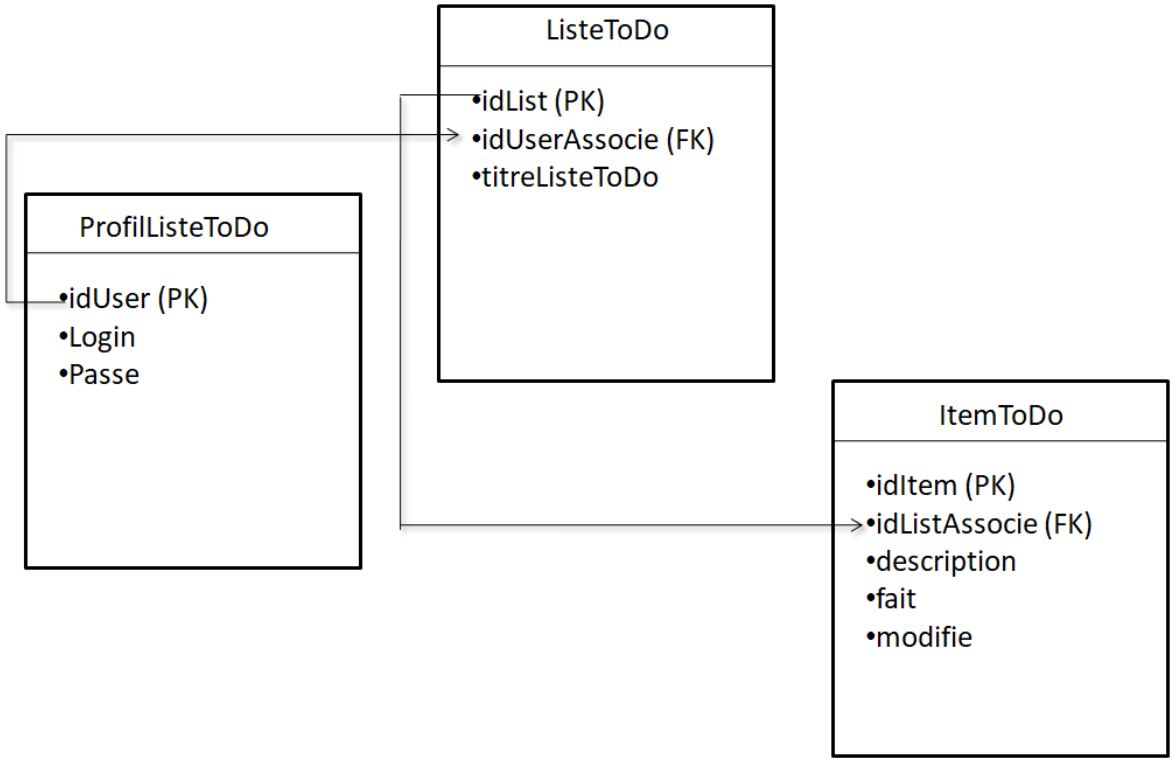
- Modification des classes afin de pouvoir mettre en place la base de données :

```
@Entity
public class ProfilListeToDo {
    @PrimaryKey
    public int idUser;
    public String login;
    private String passe;
```

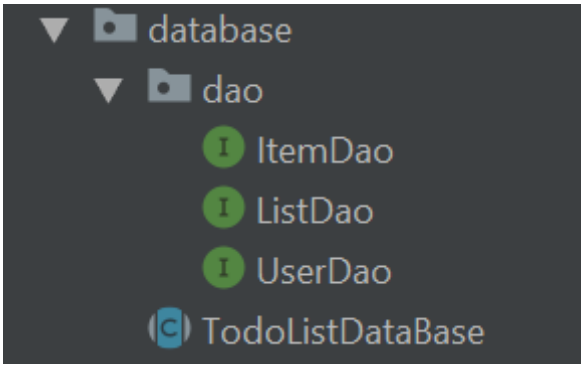
```
@Entity(foreignKeys = @ForeignKey(entity = ProfilListeToDo.class,
        parentColumns = "idUser",
        childColumns = "idUserAssocie"))
public class ListeToDo {
```

La base de données est composée de 3 tables. **ProfilListeToDo**, **ListeToDo** et **ItemToDo**. Pour chacune de ces tables, c'est les attributs des classes associées qui feront les colonnes.
Il faut ajoute @Entity avant la déclaration de la table afin d'informer de cette manipulation.
Il faut également ajouter des informations pour pouvoir correctement construire la base de données. Par exemple pour ProfilListeToDo de la première image, on indique grâce à @PrimaryKey quel sera la clef primaire de la classe. De même pour la classe ListeToDo, on informe la clef secondaire grâce à @ForeignKey.

La structure de la base de données que j'ai choisis de suivre est la suivante :




Une fois la gestion des classes pour correctement construire ma base de données, il faut mettre en place la base de données en elle-même, ainsi que 3 interfaces (une pour chaque table) qui contiendra les requêtes SQL que nous utiliserons au sein de l'application.




Pour la mise en place de TodoListDataBase, je me suis appuyé sur ce que M.Boukadir nous a présenté :

```
@Database(entities = {ProfilListeToDo.class, ListeToDo.class, ItemToDo.class}, version = 1, exportSchema = false)
public abstract class TodoListDataBase extends RoomDatabase {
    // --- SINGLETON ---
    private static volatile TodoListDataBase INSTANCE;

    // --- DAO ---
    public abstract UserDao userDao();
    public abstract ListDao listDao();
    public abstract ItemDao itemDao();

    // --- INSTANCE ---
    public static TodoListDataBase getDatabase(Context context) {
        if (INSTANCE == null) {
            synchronized (TodoListDataBase.class) {
                if (INSTANCE == null) {
                    INSTANCE = Room.databaseBuilder(context.getApplicationContext(),
                            TodoListDataBase.class, name: "MyDatabase.db")
                            .build();
                }
            }
        }
        return INSTANCE;
    }
```

Pour les interfaces, il faut indiquer que c'est un DAO (data access object) avec @Dao avant la déclaration de l'interface. On utilise @Query ou autre pour faire les requêtes souhaitées.
Je vous présente ici quelques méthodes que j'ai utilisées pour la gestion de la table ItemToDo :

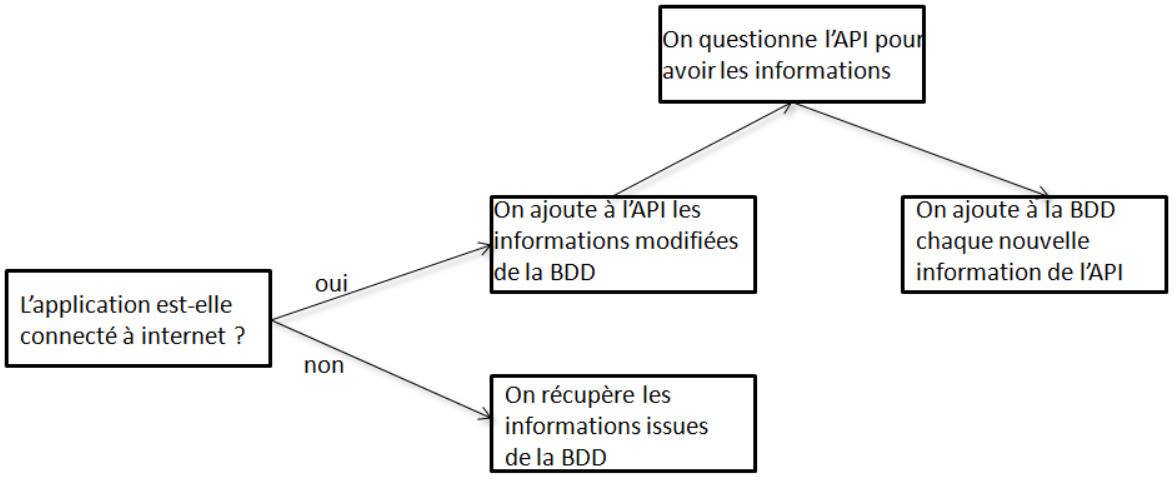
```
@Query("UPDATE ItemToDo SET fait = :fait, modifie='true' WHERE idItem= :idItem")
int updateItem(boolean fait, int idItem);

@Query("SELECT fait FROM ItemToDo WHERE idItem = :idItem")
boolean getFaitBdd(int idItem);
```

La fonctionnalité des méthodes updateItem ou bien getFaitBdd se lit à travers la requête SQL sur la ligne juste au dessus. L'utilisation de ces méthodes se fait dans la classe DataProvider. Certes l'appel des méthodes est redondant mais cela permet de mieux structurer le code :

```
public int updateItem(boolean fait, int idItem){
    return (int) itemDao.updateItem(fait,idItem);
}
public boolean getFaitBdd(int idItem){
    return (boolean) itemDao.getFaitBdd(idItem);
}
```

La mise en place de la base de données est donc correctement faite. Pour pouvoir y accéder, ou bien la modifier, tout se fait via la classe DataProvider : à travers un asynctask d'une activité, on appelle une fonction de DataProvider qui ira chercher une fonction du Dao, qui elle-même ira chercher dans la base de données.

Ci-après la logique suivit pour la gestion des données, pour cette 3ème version de l'application :




La gestion de la connexion se fait au sein de la MainActivity. Selon l'état de la connexion, on ouvre un AsyncTask particulier. Lors du doInBackGround de l'AsyncTask, c'est la que l'on va questionner la BDD afin de ne pas autoriser le MainThread à faire les requêtes SQL comme demandé dans l'énoncé.
Prenons comme exemple l'asynctask lié au questionnement de la BDD dans la MainActivity, afin de vérifier que l'utilisateur fait bien parti de la BDD :

```
@Override protected List<ProfilListeToDo> doInBackground(Object... objects) {
    return  dataProvider.loadUsers();
}

@Override protected void onPostExecute(List<ProfilListeToDo> listUsers) {
    super.onPostExecute(listUsers);
    for (ProfilListeToDo user :  listUsers){
        if(user.getLogin().equals(pseudo) && user.getPasse().equals(password)){
            data.putInt("idUser", user.getIdUser());
            //On lance l'activité suivante
            toChoixListActivity.putExtras(data);
            findViewById(R.id.progess).setVisibility(View.GONE);
            startActivity(toChoixListActivity);
        }
    }
    findViewById(R.id.progess).setVisibility(View.GONE);
}
```

Dans le doInBackground, on utilise le passage par un objet dataProvider de la classe éponyme afin de récupérer la liste des utilisateurs :

```
public List<ProfilListeToDo> loadUsers() {
    return (List<ProfilListeToDo>) userDao.getUser();
}
```

Qui fait appel à l'interface UserDao :

```
@Query("SELECT * FROM ProfilListeToDo")
List<ProfilListeToDo> getUser();
```

Une fois que l'on a la liste des utilisateurs, on la parcours pour vérifier que le pseudo et le mot de passe existe dans la BDD. Si c'est le cas, on retient l'identifiant de l'utilisateur pour le donner à l'activité suivante, puis on change d'activité. Le schéma est le même pour les deux classes suivantes. Il fallait cependant faire bien attention à la manière de récupérer et transmettre les identifiants liés aux listes et aux items. En effet j'ai choisi d'avoir comme identifiant dans la BDD les mêmes identifiants que ceux sauver dans l'API, afin de ne pas avoir de problème de correspondance.

Ci-après, la gestion de l'évènement Click sur une checkbox dans ShowListActivity.

```
@Override
protected String doInBackground(String... strings) {
    if(!dataProvider.getFaitBdd(idItem)){
        dataProvider.updateItem( fait: true,idItem);
    }
    else {
        dataProvider.updateItem( fait: false,idItem);
    }
    return "";
}
```

Enfin, lorsque l'utilisateur relance l'application une fois que la connexion a été établie, on lance une asynctask pour mettre à jour l'API selon les modifications qui ont été apportées dans la BDD.

```
@Override protected List<ItemToDo> doInBackground(Object... objects) {
    mHash=dataProvider.getHash(myUrl, pseudo: "tom", motDePasse: "web", methode: "POST");
    listItemModifie = dataProvider.getItemModifie();
    for(ItemToDo item : listItemModifie){
        dataProvider.requete( qs: myUrl + "/lists/"+ item.getIdListAssocie()
                +"/items/"+ item.getIdItem()
                +"?check=" + item.getFait()
                + "+&hash="+mHash, methode: "PUT");
    }
    return  listItemModifie;
}
```

Il faut un hash valide pour pouvoir apporter des modifications, cependant l'utilisateur n'est pas sensé être déjà connecté, on choisit donc d'utiliser l'utilisateur «créateur» pour nous connecter. (tom/web) L'ajout de l'attribut « modifie » dans la classe ItemToDo, nous permet de facilement retrouver tous les items qui ont subit une modification depuis la dernière connexion à l'API. La requête PUT permet de tous déposer sur l'API.

## Conclusion :

Il apparait alors un problème de sécurisation de l'application. En effet lorsque l'application retrouve une connexion et qu'il faut mettre à jour l'API, un hash est nécessaire. Afin d'obtenir ce hash, il faut écrire quelque part un pseudo et un mot de passe. Un utilisateur qui trouverait le code de l'application aurait donc trouvé un identifiant et un mot de passe fonctionnel pour pouvoir avoir un lien avec l'API.
Une solution serait d'attendre que l'utilisateur se connecte effectivement à l'API, pour récupérer son hash et ainsi mettre à jour l'API. De plus, la base de donnée a en mémoire le mot de passe entré par l'utilisateur sans qu'il soit scripté, il faudrait donc scripter le mot de passe afin d'assurer la sécurisation.

La construction de l'implémentation d'information dans la BDD est faite de telle manière que c'est à chaque réponse de l'API qu'on enregistrer des informations dans la BDD. En effet, chaque activité a son Asynctask qui sert de « mise en mémoire » de l'information qui vient d'être perçue. Cela implique que lorsque l'on passe en mode hors ligne, une liste qui n'aurait pas été visitée par l'utilisateur n'est pas mise en mémoire de la BDD. Je me suis donc demandé s'il n'était pas meilleur de sauvegarder la totalité du profil de l'utilisateur (càd ensemble de liste + ensemble d'items associés à l'utilisateur) dès que l'utilisateur se connecte. Je n'ai pas su trouver de réponse à cette interrogation. Cependant, malgré un UX peut-être légèrement détériorée, il me semble que le cache n'est pas fait pour sauver une grande quantité d'information. Si le profil associé à l'utilisateur représente une grande quantité d'information, alors tout est sauvegardé sans que l'utilisateur en ait conscience. Au vu de cette dernière réflexion j'ai choisi de ne sauver que les données « parcourues ».

**Bibliographie :**

Tutoriel pour implémenter une base de données dans une application https://openclassrooms.com/fr/courses/4568746-gerez-vos-donnees-localement-pour-avoir-une-application-100-hors-ligne