Les ORM

Duthoit Lucile (FI)
8 novembre 2017

Résumé

Les base de données sont très utiliser dans les applications Android pour personnaliser son contenue. Utiliser un ORM permet d'aller plus vite dans l'utilisation et la création de BDD. Cela facilite la vie aux développeurs.

Dans ce TP, nous verrons comment créer des tables, utiliser et faire des requêtes à partir de ORMLite sur ces tables.

Pré-requis

- Connaître les bases d'Android
- Comprendre un minimum les bases de données
- Connaitre Sqlite pour Android

Code source

Code source **initial** disponible à https://github.com/LucileD/TutorielOrmLiteinitial.git

Code source **final** disponible à https://github.com/LucileD/TutorielOrmLitefinal.git

Explications du TP

Mon tutoriel est sur les ORMs, dans ce tutoriel nous allons utiliser ORMLite et donc utiliser Sqlite pour la base de données. Il nous faut donc une architecture pour celle ci. J'ai décidé de partir sur une base de données pour recenser des films, pour par exemple voir ceux que l'on a déjà vu et ceux que l'on veut voir. Présentation de l'architecture :

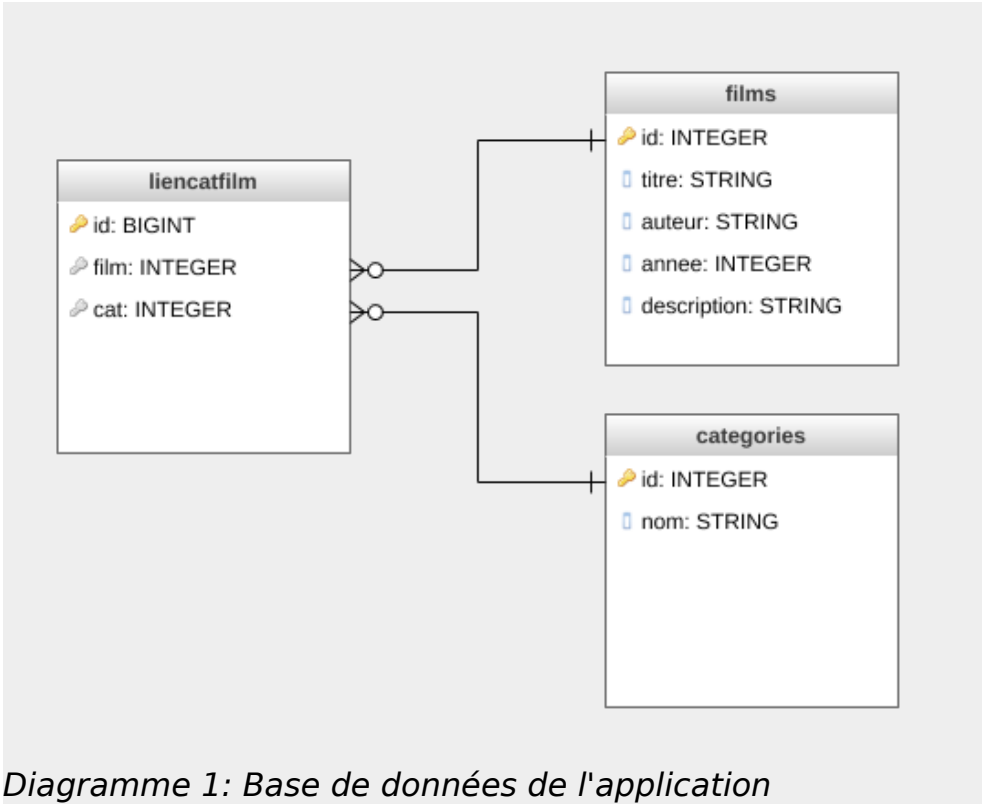


Diagramme 1: Base de données de l'application

Et maintenant c'est parti pour les étapes à réaliser.

Étape 1 : Ajouter la librairie

Pour pouvoir utiliser ORMLite, il ne faut pas oublier dans les dépendances du gradle (celui de app) de mettre :

```
    compile 'com.j256.ormlite:ormlite-android:5.0'
```

Maintenant que l'on peut utiliser ORMLite, commençons par définir une table

Étape 2 : Création d'une table

Aller dans les fichiers java plus particulièrement dans : bdd/Film.java

définition de la classe/table

Pour que la classe métier devienne aussi une table de la BDD, il suffit de rajouter l'annotation **@DatabaseTable** avec le nom voulu pour la table. Décommentez (l23) (avec alt + entrée pour les import).

```
@DatabaseTable(tableName = "films")
public class Film implements Serializable {
```

optionnel choix des nom des colonnes

On peut pour se simplifier la vie et pour être sur de ne pas se tromper dans les noms définir des variables statiques pour les noms des colonnes de la futur table. Pour notre table films on a donc :

```
public static final String FILMS_TITRE = "titre";
public static final String FILMS_AUTEUR = "auteur";
public static final String FILMS_ID = "id";
public static final String FILMS_ANNEE =  "annee";
public static final String FILMS_DESCRIPTION =  "description";
```

création des colonnes

- Colonne de base et id

Pour créer les colonnes de la table il n'y a pas grande différence par rapport à créer les attributs de la classe métier. Il faut juste avant chaque attribut, rajouter une annotation **@DatabaseField** pour renseigner le nom de la colonne dans la table **columnName** et **generatedId** (de base à false) pour que l'id de l'objet soit généré automatiquement. Décommentez (l23,26,29,32,35) la définition pour la colonne de l'id et les autres attributs.

```
@DatabaseField(columnName = FILMS_ID,generatedId =  true)
private Integer mid;
```

- Colonne avec clé étrangère

Pour ce point aller dans bdd/LienCatFilm.java

Cette fois ci dans l'annotation **@DatabaseField,** mettre toujours columnName et ajouter les éléments **foreign** et **foreignAutoRefresh** en les passant à true, ça précise que c'est une clé étrangère et qu'elle doit se mettre à jour automatiquement. Décommentez (l27).

```
@DatabaseField(columnName = LIENCATFILM_FILM,
foreign=true,foreignAutoRefresh = true)
private Film film;
```

On reviens à la classe bdd/Film.java

constructeur

Comme notre table fait aussi office de déclaration de table il faut obligatoirement mettre un **constructeur vide** dedans sinon l'application plante à l'exécution. Décommentez(l38-40).

```
    public Film () {
    //TODO !!!!!!!!!! obligatoire !!!!!!!!!!
    }
```

Si ça vous intéresse vous pouvait regarder les autres classes/tables créées mais le système est le même.
Maintenant que l'on a nos classes, on va créer une classe pour nous aider à accéder aux base de données.

Étape 3 : DatabaseHelper

(dans bdd/DatabaseHelper.java)

La classe DatabaseHelper étends la classe **`OrmLiteSqliteOpenHelper`**.

Les attributs

Dans les attributs on y met le nom de notre fichier de base de données, ainsi que la version de notre base de données, ça c'est comme pour quand vous utiliser Sqlite. Ici la différence c'est que l'on y met aussi les **Dao** de nos différentes tables, donc à ne pas oublier dès qu'on ajoute une nouvelle table à notre base de données. Décomentez(l25-27).

```
    //nom du fichier de base de données
    public static final String DATABASE_NAME = "base.db";
    //Dao de toutes les tables de la base de données
    private Dao<Film, Integer> filmDao;
    private Dao<Categorie, Integer> categorieDao;
    private Dao<LienCatFilm, Integer> liencatfilmDao;
    private static final int DATABASE_VERSION = 1;
```

Le constructeur

Tout ce qu'il y a à faire dans le constructeur, c'est d'appeler le super. On y renseigne le contexte que l'on passe en paramètre, le nom du fichier de base de donnée, un curseur que l'on peut mettre à « null » et la version de la base de donnée.

```
    public DatabaseHelper(Context context) {
     super(context, DATABASE_NAME, null, DATABASE_VERSION );
    }
```

Les fonctions de la super classe

Il y a deux fonctions à compléter, **onCreate** et **onUpgrade**. La manière la plus simple de les remplir est que dans le onCreate on y met la création des tables et dans le onUpgrade, on supprime les tables si elles existent et on fait un onCreate derrière. Donc ne pas oublier à chaque fois que l'on crée une **nouvelle table de mettre les lignes associées** dans ces fonctions

Dans onCreate. Décommentez(l37-45).

```
    TableUtils.createTable(connectionSource, Film.class);
    TableUtils.createTable(connectionSource, Categorie.class);
    TableUtils.createTable(connectionSource, LienCatFilm.class);
```

Dans onUpgrade. Décommentez(51-60).

```
    TableUtils.dropTable(connectionSource, LienCatFilm.class, true);
    TableUtils.dropTable(connectionSource, Film.class, true);
    TableUtils.dropTable(connectionSource, Categorie.class, true);
    onCreate(db,connectionSource);
```

Les fonctions d'accès aux Dao des tables

On crée une fonction par table, ces fonctions permettent de remplir les constructeurs associé s'ils sont vides et de renvoyer ces **Dao** pour permettre d'**interagir sur les tables** de la base de données. Décommentez(l64-86).

```
public Dao<LienCatFilm, Integer> getLiencatfilmDao() throws SQLException {
  if (liencatfilmDao == null ){
    liencatfilmDao = getDao(LienCatFilm.class);
  }
  return liencatfilmDao;
}

//récupère le dao de la table films
public Dao<Film, Integer> getFilmDao() throws SQLException {
  if (filmDao == null ){
    filmDao = getDao(Film.class);
  }
  return filmDao;
}

//récupère le dao de la table categories
public Dao<Categorie, Integer> getCategorieDao() throws SQLException {
```

```
  if (categorieDao == null ){
    categorieDao = getDao(Categorie.class);
  }
  return categorieDao;
}
```

Maintenant que l'on peut accéder facilement à notre base de données, il faudrait peut être remplir un peut tout ça.

Étape 4 : injecter des données

Rendons nous dans Accueil.java

Récupérer le DatabaseHelper

On définit au début de la classe, l'attribut pour stocker l'appelle au DatabaseHelper et la fonction permettant de le récupérer. Décommentez(l30-38).

```
private DatabaseHelper databaseHelper = null;

//récupère l'objet pour appeler les tables
private DatabaseHelper getHelper () {
  if ( databaseHelper == null) {
    databaseHelper =
OpenHelperManager.getHelper(this,DatabaseHelper.class);
  }
  return databaseHelper;
}
```

Création des données

Dans genererBDD

On commence par **récupérer les Dao** des tables que l'on va utiliser. Décommentez (l110-112).

```
Dao<Film,Integer> filmDao = getHelper().getFilmDao();
Dao<Categorie,Integer> categorieDao = getHelper().getCategorieDao();
Dao<LienCatFilm,Integer> lienCatfilmsDao = getHelper().getLiencatfilmDao();
```

Ensuite, on **crée les éléments à rentrer** dans la table. Décommentez(l114-126).

```
//on crée les catégories
Categorie cat1 = new Categorie("vu");
Categorie cat2 = new Categorie("à voir");

//on crée un film
String titre = "Blade runner 2049";
```

```
String auteur = "Denis Villeneuve";
int annee = 2017;
String description = "En 2049, la société est fragilisée par les nombreuses tensions entre les humains et leurs esclaves créés par bioingénierie. L'officier K est un Blade Runner : il fait partie d'une force d'intervention d'élite chargée de trouver et d'éliminer ceux qui n'obéissent pas aux ordres des humains.";
Film film = new Film(titre,auteur,annee,description);

//on crée un lien entre un film et une catégorie
LienCatFilm lien = new LienCatFilm(film,cat2);
```

Enfin, il n'y a plus qu'à **rentrer les éléments** dans la base à partir des Dao. Décommentez(l128-136).

```
//on ajoute les catégories dans la BDD
categorieDao.create(cat1);
categorieDao.create(cat2);

//on ajoute le film dans la BDD
filmDao.create(film);

//on ajoute le lien dans la BDD
lienCatfilmsDao.create(lien);
```

Si vous voulez rentrer plus de données, parce qu'un seul film ce n'est pas énorme, décommentez (l141-230).

Donc on a bien rentré les données dans notre base de données, maintenant, on voudrait bien pouvoir les récupérer.

Étape 5 : Récupérer les données d'une table

Récupérer toutes les données

Toujours dans la même classe, dans le onResume

Pour récupérer toutes les données d'une table ce n'est pas très compliqué.

On commence par **récupérer le Dao** de la table comme à chaque fois. Décommentez (l55). Et comme la fonction peut déclencher une exception on met tout ça dans un try, décommentez (l53,l104-106).

```
Dao<Film,Integer> filmDao = getHelper().getFilmDao();
```

Et pour **récupérer la liste de tous les objets** de la table, il suffit d'appeler la fonction **queryForAll** sur le Dao :(Décommentez (l59) et commentez (l60))

```
List<Film> films = filmDao.queryForAll();
```

Et l'**on obtient la liste de tous les éléments** de la table, ici tous nos films.

Vous pouvez **lancer l'application** pour tester, n'oubliez pas pour générer la base de donnée avec les éléments que nous avons rentré tout à l'heure, il faut appuyer sur le bouton « **CRÉER BDD** ». Normalement, on peut constater la présence de tous les films rentrés.

Récupérer une partie des données

Dans activity/CategorieActivity.java, fonction onResume

Encore une fois on **récupère le Dao** de la table liencatfilm (décommentez(l54,53,91-93)) :

```
Dao<LienCatFilm, Integer> liencatfilmDao = getHelper().getLiencatfilmDao();
```

Puis on précise que l'on va faire une requête et l'on récupère un **queryBuilder.** Décommentez(l57)

```
QueryBuilder<LienCatFilm, Integer> queryBuilder = liencatfilmDao.queryBuilder();
```

Après on **écrit la requête**, ici on a juste un where, où l'on regarde si la catégorie présente dans le lien est bien celle que l'on veut afficher (décommentez(l61)) :

```
queryBuilder.where().eq(LienCatFilm.LIENCATFILM_CAT, idd);
```

Remarque : idd correspond à l'id de la catégorie à afficher que l'on a récupérer de l'autre activité. Étant donné que dans la table lien on a précisé que la colonne cat était une clé étrangère vers la table categories, la requête arrive à faire une corrélation entre l'id de la catégorie et l'objet qui se trouve dans la table lien.

Ensuite, on dit au queryBuilder que l'on a **finit de préparer la requête** et cela nous renvoi un objet **PreparedQuery** (décommentez(l64)):

```
PreparedQuery<LienCatFilm> preparedQuery = queryBuilder.prepare();
```

Enfin pour récupérer un objet Itérable sur les **résultats de la requête**, il suffit de faire (décomentez (l72-74,78-81,85)) :

```
liencatfilmDao.query(preparedQuery)
```

pour pouvoir **tester** l'application décommentez aussi dans la classe activity/CategoriesAtivity.java les lignes 53 à 82. Pour pemettre d'accès au différentes catégories et voir que d'une une catégorie on a bien que les films qui appartiennent à cette catégorie.

Étape 6 : Suppression d'élément dans une table

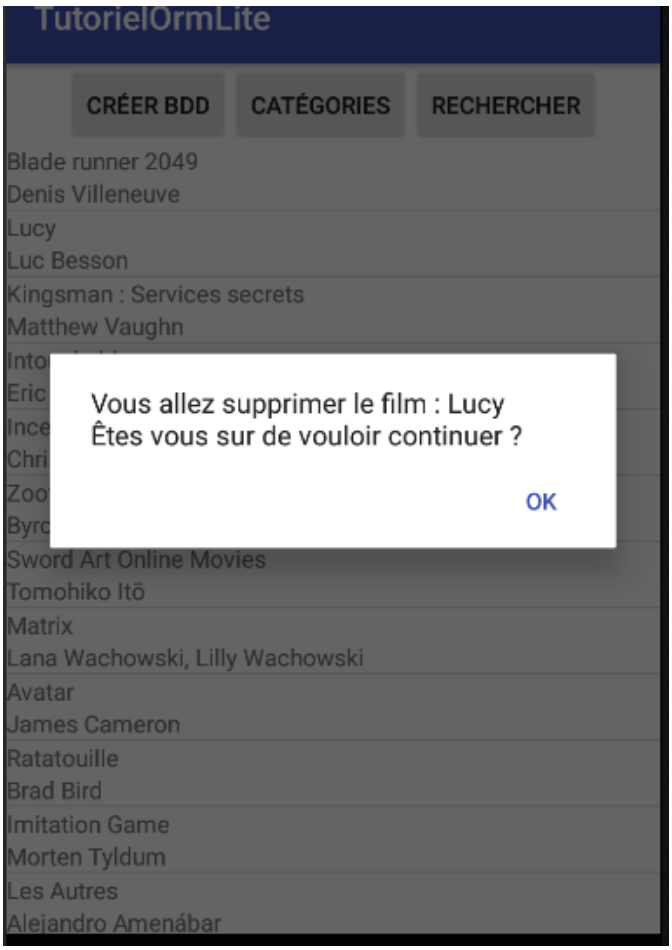
On retourner dans Accueil.java (la fin du onResume)

Assez facile et logique, on appelle toujours le **Dao** et on fait un **delete** dessus pour supprimer l'élément voulu. Ici pour supprimer un film de la table films, j'ai fait une alerte pour confirmer ou non la suppression (on supprime un film quand on clique sur lui dans la listview des films de l'accueil). Décommentez (l78-102).

```
listView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
  @Override
  public void onItemClick(AdapterView<?> parent, View view, final
int position, long id) {
    final Film film = films.get(position);
    AlertDialog.Builder builder = new
AlertDialog.Builder(Accueil.this);
    builder.setMessage("Vous allez supprimer le film, êtes vous sur
de vouloir continuer ?");
    builder.setPositiveButton("Ok", new
DialogInterface.OnClickListener() {
      @Override
      public void onClick(DialogInterface dialog, int which) {
        // faire suppression*/
        try {
          filmDao.delete(film);
          Intent intent = new
Intent(getApplicationContext(),Accueil.class);
```

```
            finish();
            startActivity(intent);
          } catch (SQLException ee) {
            ee.printStackTrace();
          }
        }
      });
      builder.create().show();
    }
});
```

Exécutez le code, supprimez le film « Lucy », allez dans les catégories et dans la catégorie « vu », remarquer que cela fait planter l'application.



En effet supprimer un film de la table films, **ne supprime pas les liens** qu'ils sont associé avec lui, ça met juste sa référence à null.

Allons donc dans activity/CategorieActivity.java dans le onResume.

Pour vérifier qu'un lien existe toujours, on verifie au moment de la récupération des liens que le film associé au lien ne vaut pas null, sinon on supprime le lien et on ne l'affiche donc pas. Décommentez (l75-77,82)

```
Film film = lien.getFilm();
if (film == null) {
  liencatfilmDao.delete(lien);
}else {
  HashMap<String, String> map = new HashMap<String, String>();
  map.put("titre", film.getTitre());
  map.put("auteur", film.getAuteur());
  mylist.add(map);
}
```

Vous pouvez retester(juste aller dans la catégorie) et normalement cela fonctionne maintenant.

Étape 7 : Les tries

Dans Accueil.java, onResume

Pour trier les résultats, il suffit juste de lancer une requête avec la fonction orderBy. Décommentez (l62-68) et commentez (l59).

```
  //List<Film> films = new ArrayList<>() ;
  //trie
  //on initialise une requête
  QueryBuilder<Film, Integer> queryBuilder = filmDao.queryBuilder();
  //on fait juste un orderBy avec la colonne de référence et un
booléen, true pour ordre croissant, false pour décroissant
  queryBuilder.orderBy(Film.FILMS_TITRE,true);
  //on dit que l'on a fini
  PreparedQuery<Film> preparedQuery = queryBuilder.prepare();
  final List<Film> films = filmDao.query(preparedQuery);
```

Exécutez le code et vous devriez voir que les films de l'accueil sont rangés dans l'ordre alphabétique.

Informations complémentaires

Webographie :

- http://ormlite.com/ : documentation officiel mais pas super compréhensible

- http://www.lifl.fr/~dumoulin/enseign/pje/cours/3.DbEtOrm/dbAndroid-2017.pdf : utile pour débuter avec ORMLite (dans les dernières diapos)

- https://blog.jayway.com/2016/03/15/android-ormlite/ : plus complet et assez compréhensible

Pour aller un peu plus loin

Pour les joins

Sans clé étrangère (imaginons que la table lien stockait les id des films et catégories sans clé étrangère):

```
//on récupère les Dao
Dao<Film,Integer> filmDao = getHelper().getFilmDao();
Dao<Categorie,Integer> catDao = getHelper().getCategorieDao();
//initialise la requête voulu sur la table films
QueryBuilder<Film, Integer> queryBuilder = filmDao.queryBuilder();
//écrit la requête, ici équivalent à : select * from films where titre like "%"+titre.getText()+"%"
queryBuilder.where().like(Film.FILMS_TITRE,"%"+titre.getText()+"%");

//initialise la requête voulu sur la table categories
QueryBuilder<Categorie, Integer> queryBuilderc = catDao.queryBuilder();
//écrit la requête, ici équivalent à : select * from categories where nom=description.getText()+"%"
queryBuilderc.where().like(Liencatfilm.LIENCATFILM.IDCAT,1);
//on lance la première requête
queryBuilderc.prepare();
//on fait un join en les deux requêtes, le résultat est l'intersection des deux requête, pour l'union utiliser joinOr, (on précise les colonnes pour la jointure)
queryBuilder.join(Film.FILMS_ID,Liencatfilm.LIENCATFILM.IDFILM);
PreparedQuery<Film> preparedQuery = queryBuilder.prepare();
```