Développement Mobile
2023-2024

Android - Semaine 6

Référence: Cours Mr Pierre Nerzic

Plan du chapitre

- Introduction
- SQLite
- Création et mise à jour
- Types de données pour SQLite
- Création de table avec SQLite
- Exécution de requêtes
- Récupéreration de la base
- Mise à jour
- Insertion
- Suppression
- Modification
- Sélection
- Les curseurs
- L'adaptateur pour les curseurs

Introduction

- SQLite permet de créér des bases de données légères internes aux applications Android
- SQLite ne nécessite pas de serveur pour fonctionner, ce qui signifie que son exécution se fait dans le même processus que celui de l'application.
- Par conséquent, une opération massive lancée dans la base de données aura des conséquences visibles sur les performances de votre application.
- Ainsi, il vous faudra savoir maîtriser son implémentation afin de ne pas pénaliser le restant de votre exécution.

SQLITE

- SQLite a été inclus dans le cœur même d'Android, c'est pourquoi chaque application peut avoir sa propre base.
- De manière générale, les bases de données sont stockées dans les répertoires de la forme /data/data/<package>/databases.
- Il est possible d'avoir plusieurs bases de données par application, cependant chaque fichier créé l'est selon le mode MODE_PRIVATE, par conséquent les bases ne sont accessibles qu'au sein de l'application elle-même.
- Il est préférable de faire en sorte que la clé de chaque table soit un identifiant qui s'incrémente automatiquement.

Création et mise à jour

- La solution la plus évidente est d'utiliser une classe qui nous aidera à maîtriser toutes les relations avec la base de données.
- Cette classe dérivera de **SQLiteOpenHelper**.
- A la création de la base de données, la méthode de *callback* void onCreate(SQLiteDatabase db) est automatiquement appelée, avec le paramètre db qui représentera la base.
- C'est dans cette méthode que vous devrez lancer les instructions pour créer les différentes tables et éventuellement les remplir avec des données initiales.

EXEMPLE

```
public class DatabaseOpenHelper extends SQLiteOpenHelper
{
private static final String SQLCreateTableArticles =
"CREATE TABLE Articles " + " ( ID " + " INTEGER PRIMARY KEY,"
    + " Title TEXT " + " )";
public static final int databaseVersion = 1;
public static final String databaseName = "articlesDB";
private static final String SQLDeleteTableArticles =
"DROP TABLE IF EXISTS Articles“ ;
public DatabaseOpenHelper(Context context) {
super(context, databaseName, null, databaseVersion);
}
@Override
public void onCreate(SQLiteDatabase database) {
database.execSQL(SQLCreateTableArticles);
}
}
```

Création et mise à jour

- Pour créer une table, il faut fournir son nom et ses attributs.
- Chaque attribut sera défini à l'aide d'un type de données.
- On veut par exemple créer la table Metier caractérisée par
  - ID, qui est un entier auto-incrémental pour représenter la clé ;
  - Métier, qui est le nom du métier représenté par une chaîne de caractères ;
  - Salaire, qui est un nombre réel.

Types de données pour SQLite

- Pour SQLite, c'est simple, il n'existe que cinq types de données ^_^ :
  - NULL pour les données Nulles.
  - INTEGER pour les entiers (sans virgule).
  - REAL pour les nombres réels (avec virgule).
  - TEXT pour les chaînes de caractères.
  - BLOB pour les données brutes, par exemple si vous voulez mettre une image dans votre base de données

Création de table avec SQLite

- La création de table se fait avec une syntaxe très naturelle :

```
CREATE TABLE nom_de_la_table (
nom_du_champ_1 type {contraintes},
nom_du_champ_2 type {contraintes},
...);
```

- Pour chaque attribut, on doit déclarer au moins deux informations :
  - Son nom, afin de pouvoir l'identifier ;
  - Son type de donnée.

Création de table avec SQLite

- Il est aussi possible de déclarer des contraintes pour chaque attribut à l'emplacement de {contraintes}. On trouve comme contraintes :
  - PRIMARY KEY pour désigner la clé primaire sur un attribut
  - NOT NULL pour indiquer que cet attribut ne peut valoir NULL ;
  - CHECK afin de vérifier que la valeur de cet attribut est cohérente ;
  - DEFAULT sert à préciser une valeur par défaut.
- Ce qui peut donner par exemple :

```
CREATE TABLE nom_de_la_table (
champ1 INTEGER PRIMARY KEY,
champ2 TEXT NOT NULL,
champ3 REAL NOT NULL CHECK (nom_du_champ_3 > 0),
champ4 INTEGER DEFAULT 10);
```

EXÉCUTION DE REQUÊTES

- Afin d'exécuter une requête SQL pour laquelle on ne souhaite pas de réponse ou on ignore la réponse, il suffit d'utiliser la méthode

void execSQL(String sql)

- De manière générale, on utilisera execSQL(String) dès qu'il ne s'agira pas de faire un SELECT, UPDATE, INSERT ou DELETE.

Exécution de requêtes

- Par exemple, pour notre table Metier :

```
public class DatabaseHandler extends SQLiteOpenHelper {
  public static final String METIER_KEY = "id";
  public static final String METIER_INTITULE = "intitule";
  public static final String METIER_SALAIRE = "salaire";

  public static final String METIER_TABLE_NAME = "Metier";
  public static final String METIER_TABLE_CREATE =
  "CREATE TABLE " + METIER_TABLE_NAME + " (" +
     METIER_KEY + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
     METIER_INTITULE + " TEXT, " +
     METIER_SALAIRE + " REAL);";
  public DatabaseHandler(Context context, String name, CursorFactory
     factory, int version) {
   super(context, name, factory, version);
  }
  @Override
  public void onCreate(SQLiteDatabase db) {
   db.execSQL(METIER_TABLE_CREATE);    }}
```

Exécution de requêtes

Le problème du code précédent, c'est qu'il ne fonctionnera pas, et ce pour une raison très simple : il faut aussi implémenter la méthode onUpgrade qui est déclenchée à chaque fois que l'utilisateur met à jour son application.

void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)

- oldVersion est le numéro de l'ancienne version de la base de données que l'application utilisait, alors que newVersion est le numéro de la nouvelle version.

- En fait, Android rajoute automatiquement dans la base une table qui contient la dernière valeur connue de la base.
- À chaque lancement, Android vérifiera la dernière version de la base par rapport à la version actuelle dans le code.
- Si le numéro de la version actuelle est supérieur à celui de la dernière version, alors cette méthode est lancée.
- En général, le contenu de cette méthode est assez constant puisqu'on se contente de supprimer les tables déjà existantes pour les reconstruire suivant le nouveau schéma

EXEMPLE

```
public static final String
    METIER_TABLE_DROP =
    "DROP TABLE IF EXISTS " +
    METIER_TABLE_NAME + ";";
@Override
public void
    onUpgrade(SQLiteDatabas
    e db, int oldVersion, int
    newVersion) {
db.execSQL(METIER_TABLE_DRO
P);
onCreate(db);
}
```

RÉCUPÉRERATION DE LA BASE

- Si vous voulez accéder à la base de données n'importe où dans votre code, il vous suffit de construire une instance de votre SQLiteOpenHelper avec le constructeur

  SQLiteOpenHelper(Context context, String name, SQLiteDatabase.CursorFactory factory, int version)

  - name est le nom de la base,
  - factory est un paramètre dont on explique après, il accepte la valeur null
  - Version est la version voulue de la base de données.

Mise à jour

- Mise à jour de la base de données lors d'un changement de version:

```
public class DatabaseOpenHelper extends SQLiteOpenHelper {
public static final int databaseVersion = 1;
@Override
public void onUpgrade(SQLiteDatabase database, int oldVersion,
int newVersion) {
database.execSQL(SQLDeleteTableArticles);
onCreate(database);
}
@Override
public void onDowngrade(SQLiteDatabase database, int oldVersion,
int newVersion) {
onUpgrade(database, oldVersion, newVersion); } }
```

Insertion

- Pour ajouter une entrée dans la table, on utilise la syntaxe suivante :

```
INSERT INTO nom_de_la_table
(colonne1, colonne2, …) VALUES (valeur1, valeur2, …)
```

- La partie (colonne1, colonne2, …) permet d'associer une valeur à une colonne précise à l'aide de la partie (valeur1, valeur2, …).

Exemple

```
INSERT INTO Metier (Salaire, Metier) VALUES (50.2, "Prof")
```

Suppression

- La méthode utilisée pour supprimer est:

int delete(String table, String whereClause, String[] whereArgs)

- L'entier renvoyé est le nombre de lignes supprimées. Avec:
  - table est le nom de la table.
  - whereClause correspond au WHERE en SQL. Par exemple, pour sélectionner la première valeur dans la table Metier, on mettra pour whereClause la chaîne « id = 1 ». En pratique, on préférera utiliser la chaîne « id = ? » .
  - whereArgs est un tableau des valeurs qui remplaceront les « ? » dans whereClause. Ainsi, si whereClause vaut « LIKE ? AND salaire > ? » et qu'on cherche les métiers qui ressemblent à « ingénieur avec un salaire supérieur à 1000 dinars », il suffit d'insérer dans whereArgs un String[] du genre {"ingenieur", "1000"}.

Modification

```
int update(String table, ContentValues values, String
    whereClause, String[] whereArgs).
```

- On ajoute juste le paramètre values pour représenter les changements à effectuer dans le ou les enregistrements cibles. Donc, si je veux mettre à jour le salaire d'un métier, il me suffit de mettre à jour l'objet associé et d'insérer la nouvelle valeur dans un ContentValues comme suit:

```
ContentValues value = new ContentValues();
value.put(SALAIRE, m.getSalaire());
mDb.update(TABLE_NAME, value, KEY + " = ?", new
    String[] {String.valueOf(m.getId())});
```

SELECTION

- 3 formes différentes en fonction des paramètres qu'on veut lui passer. La première forme est celle-ci

Cursor query (boolean distinct, String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, String limit)

- La deuxième forme s'utilise sans l'attribut limit et la troisième sans les attributs limit*et*distinct.
- distinct, si vous ne voulez pas de résultats en double.
- table est l'identifiant de la table.
- columns est utilisé pour préciser les colonnes à afficher.
- selection est l'équivalent du whereClause précédent.
- selectionArgs est l'équivalent du whereArgs précédent.
- group by permet de grouper les résultats.
- having est utile pour filtrer parmi les groupes.
- order by permet de trier les résultats. Mettre ASC pour trier dans l'ordre croissant et DESC pour l'ordre décroissant.
- limit pour fixer un nombre maximal de résultats voulus.

Les curseurs

- Les curseurs permettent de parcourir le résultat d'une requête :
  - int getCount();
  - boolean moveToFirst();
  - boolean moveToNext();
  - boolean moveToPosition(int position);
  - int getColumnIndex(String columnName);
  - String getString(int columnIndex)
  - double getDouble(int columnIndex)
  - ...
  - void close();
  - ...

Les curseurs

- Ainsi, pour parcourir les résultats d'une requête, il faut procéder ligner par ligne. Pour naviguer parmi les lignes, on peut utiliser les méthodes suivantes :
  - boolean moveToFirst() pour aller à la première ligne.
  - boolean moveToLast() pour aller à la dernière.
  - boolean moveToPosition(int position) pour aller à la position voulue, sachant que vous pouvez savoir le nombre de lignes avec la méthode int getCount().
- Un Cursor est capable de retenir la position du dernier élément que l'utilisateur a consulté, il est donc possible de naviguer d'avant en arrière parmi les lignes grâce aux méthodes suivantes :
  - boolean moveToNext() pour aller à la ligne suivante. Par défaut on commence à la ligne -1, donc, en utilisant un moveToNext() sur un tout nouveau Cursor, on passe à la première ligne.
  - boolean moveToPrevious() pour aller à l'entrée précédente.
- Toutes ces méthodes renvoient des booléens.
- Pour récupérer la position actuelle, on utilise int getPosition(). Vous pouvez aussi savoir si vous êtes après la dernière ligne avec boolean isAfterLast().
- Par exemple, pour naviguer entre toutes les lignes d'un curseur, on fait :

```
while (cursor.moveToNext()) {
  // Faire quelque chose
}
cursor.close();
```

Les curseurs

- Pour naviguer entre les colonnes de la table métier dont la première contient un entier, la deuxième, une chaîne de caractères, et la troisième, un réel. Pour récupérer le contenu d'une de ces colonnes, il suffit d'utiliser une méthode du style X getX(int columnIndex) avec X le typage de la valeur à récupérer et columnIndex la colonne dans laquelle se trouve cette valeur.
- On peut par exemple récupérer un Metier complet avec :

```
long id = cursor.getLong(0);
String intitule = cursor.getString(1);
double salaire = cursor.getDouble(2);
Metier m = new Metier (id, intitule, salaire);
```

L'adaptateur pour les curseurs

- Comme n'importe quel adaptateur, un CursorAdapter fera la transition entre des données et un AdapterView.
- Cependant, comme on trouve rarement *une seule* information dans un curseur, on préférera utiliser un SimpleCursorAdapter, qui est un équivalent au SimpleAdapter que nous avons déjà étudié.
- Pour construire ce type d'adaptateur, on utilisera le constructeur suivant :
- **SimpleCursorAdapter (Context context, int layout, Cursor c, String[] from, int[] to)**
- avec:
  - layout est l'identifiant de la mise en page des vues dans l'AdapterView.
  - c est le curseur.
  - from indique une liste de noms des colonnes afin de lier les données au layout.
  - to contient les TextView qui afficheront les colonnes contenues dans from.

RÉFÉRENCES

- https://openclassrooms.com/courses/creez-des-applications-pour-android/les-bases-de-donnees-5