CAI-SI01 SMB116

La Persistance

Persistance (1)

- Les applications manipulent des **données en mémoire**.
- Certaines de ces données doivent **survivre à un arrêt système**: d'où la nécessité de les sauvegarder.
- Les **données des applications orientées objets sont représentées par les « objets métiers ».**
- **Persistance: mécanisme permettant de sauvegarder l'état des objets métiers** d'une application dans le but de pouvoir les retrouver ultérieurement dans ce même état.
- Les Objets métiers à sauvegarder sont dits **« Persistants »** et sont nommés **« Entités ».**

Persistance (2)

- **Le mécanisme de persistance doit donc permettre :**
  - **L' Enregistrement:** Persister des Objets
  - **La Modification:** Persister les changements d'état de ces Objets
  - **La Lecture:** Retrouver les objets persistés
  - **La Suppression:** détruire définitivement les objets qui n'existent plus

Persistance (3)

- **La persistance consiste à enregistrer les données :**
  - Dans des fichiers simples (texte, xml, binaire etc…)
  - Dans des Bases de Données (Objets, Relationnelles)
- Les **bases de données à objets** semblent **Idéales pour les applications à objets** mais sont très peu adoptées *(pas d'implémentation Android à ma connaissance)*
- Les bases de données relationnelles ont été préférées bien que pas réellement adaptées :

  « On veut faire entrer des objets (graphe) dans des tables (rectangles) »

ORM : Object Relational Mapping (1)

- Les représentations utilisées dans **les BDD relationnelles** et les **objets** manipulés par les **applications objet sont incompatibles.**

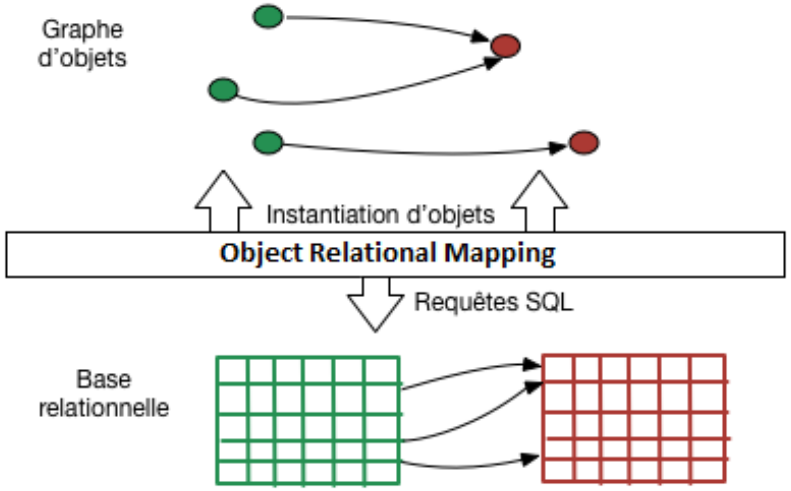
- **Il faut faire correspondre ces deux représentations :**
  «C'est le **Mapping Objet Relationnel** ou **Object Relational Mapping**

- **L' ORM assure :**
  - La traduction du graphe d'objet en tuples stockés dans des tables
  - La reconstitution du graphe d'objets à partir des données stockés dans des tables de la BDD

- **La création d'un schéma relationnel** se déduit à partir de l'ensemble des classes, de leurs associations et de leurs relations d'héritages. (Règles étudiées en NFE113)

ORM : Object Relational Mapping (2)

- Les langages objet manipulent des **Objets correspondants à des Classes**



- Le langage standard des BDD Relationnelles est **SQL pour manipuler les Tables**

SQLite

- **SQLite :**
  - **Base de données SQL open source**
  - **Stocke les données dans des fichiers texte**
  - **Supporte les instructions de base SQL pour :**
    - La description des données
    - la manipulation des données
    - La gestion de transactions
  - **Disponible sur Android**

création et ouverture d'une BDD SQLite (1)

Création d'une BDD SQLite par code :

```
SQLiteDatabase mydatabase = openOrCreateDatabase("nomBdd", MODE_PRIVATE, curSorFactory);
```

Crée la Bdd si elle n'existe pas puis l'ouvre et retourne une reference de type **`SQLiteDatabase`** pour la manipuler.

(méthode accessible à partir d'un Context)

Création et ouverture d'une Bdd en utilisant le `SQLiteOpenHelper` :

```
public class DBHelper extends SQLiteOpenHelper {
   public DBHelper(){
      super(context,DATABASE_NAME,null,1);
   }
   public void onCreate(SQLiteDatabase db) {}
   public void onUpgrade(SQLiteDatabase database, int oldVersion, int newVersion) {}
}
```

L'instance de la BDD est obtenue en invoquant la méthode `getWritableDatabase()` sur le `SQLiteOpenHelper`

création et ouverture d'une BDD SQLite (2)

Implémentation d'un Helper personnalisé :

```
public class BddDemoOpenHelper extends SQLiteOpenHelper {

    public static final String NOM_BDD = "bddDemo";
    public static final int VERSION_BDD = 1;

    public BddDemoOpenHelper(Context context) {
        super(context, name, null, bddVersion);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        // Méthode appelée lorsqu'il faut créer la Bdd
        db.execSQL("DROP TABLE IF EXISTS Utilisateur;");
        db.execSQL("CREATE TABLE IF NOT EXISTS Utilisateur(Username VARCHAR,Password VARCHAR);");
    }

    @Override
    public void onUpgrade(SQLiteDatabase sqLiteDatabase, int oldVersion, int newVersion) {
        // Méthode appelée lorsque la version de la Bdd a changé
        // pour mettre à jour la structure de la Bdd
        //...
    }
}
```

Requêtes Actions : LDD (1)

- **Exécution de requêtes action en SQL :**

```
execSQL(String sql, Object[] bindArgs)
```

  - **Création des tables :**

```
mydatabase.execSQL("CREATE TABLE IF NOT EXISTS Utilisateur
                        (Username VARCHAR, Password VARCHAR);"
);
```

Requêtes Actions : Création données (2)

- **Ajout de données dans les tables :**
  - **Requête texte en dur :**

```
mydatabase.execSQL("INSERT INTO Utilisateur VALUES('toto', 'secret');");
```

  - **Requête paramétrée :**

```
mydatabase.execSQL("INSERT INTO Utilisateur VALUES(?, ?);",
                   new Object[]{'toto', 'secret'}
);
```

  - **Méthode insert :**

```
ContentValue values = new ContentValues();
values.put("username", "toto");
values.put("password", "secret");
mydatabase.insert("Utilisateur", null, values);   // null est le nullColumnHack (nom de la
                                                  // colonne à passer à null si values est vide)
```

Requêtes Actions : Modifications données (3)

- **Modification des données dans les tables :**
  - **Requête texte en dur ou paramétrée :**

```
mydatabase.execSQL("UPDATE Utilisateur SET password = 'terces'
                           WHERE username = 'toto';");
mydatabase.execSQL("UPDATE Utilisateur SET password = ?
                           WHERE username = ?;",
                           new Object[]{'terces','toto'}
);
```

  - **Méthode Update :**

```
ContentValue values = new ContentValues();
values.put("password", "terces");
mydatabase.update("Utilisateur", values, "username = ?", new Object[]{'toto'});
```

Requêtes Actions : Suppression données (3)

- **Suppression de données dans les tables :**
  - **Requête texte en dur ou paramétrée :**

```
mydatabase.execSQL("DELETE FROM Utilisateur WHERE username = 'toto';");
mydatabase.execSQL("DELETE FROM Utilisateur WHERE username = ?;", new
                                    Object[]{'toto'}
);
```

  - **Méthode Delete :**

```
mydatabase.delete("Utilisateur", " username = 'toto' ", null); // null: arguments
```

Requêtes pour obtenir des données (1)

- **Requêtes pour extraction de données en SQL :**

`rawQuery(String sql, Object[] bindArgs)`

Requête retournant un **`Cursor`** pour itérer sur un ensemble de tuples résultats

```
Cursor resultset = db.rawQuery("SELECT * FROM Utilisateur WHERE username = ? ",
                                new String[]{'toto'}
);
```

- **Le `Cursor`** est un itérateur sur un jeu de tuples
  - Si cursor *avant* premier ou *après* le dernier tuple (**`isBeforeFirst()`** ou **`isAfterLast()`**) ➔ pas de données
  - Pour faire avancer le curseur et itérer sur les tuples : **`moveNext()`**

Requêtes pour obtenir des données (2)

- **Accès aux valeurs des colonnes du tuple pointé par le curseur :**

`getTTT (index)` :

- **`TTT`** est le type de la valeur de la colonne,
- **`index`** est le rang de la colonne (commence à 0)

`getString, getDouble, getFloat, getInt, getLong, getBlob`

- **Obtention de méta-informations à partir du curseur :**
  - **`getColumnCount()`** : nombre de colonnes
  - **`getColumnIndex(String columnName)`** : index de la colonne à partir de son nom
  - **`getCount()`** : nombre de lignes retournées par la requête

Persistance en manuel avec SQLite (1) – Entité

```java
/**
 * Entité Utilisateur
 */
public class Utilisateur {

    protected String username;
    protected String password;

    public Utilisateur(String username, String password) {
        this.username = username;
        this.password = password;
    }

    public String getUsername() {return username;}
    public void setUsername(String username) {this.username = username;}

    public String getPassword() {return password;}
    public void setPassword(String password) {this.password = password;}

    @Override
    public String toString() {
        return "Utilisateur{" + "username='" + username + '\'' + ", password='" + password + '\'' + '}';
    }
}
```

Persistance en manuel avec SQLite (2.1)– Dao

```
/**
 * Dao pour les entités Utilisateur
 */
public class UtilisateurDao {

    public SQLiteDatabase db;
    /**
     * Ouverture de la BDD
     * @param context
     */
    public void open(Context context) {
        db = context.openOrCreateDatabase("bddDemo", context.MODE_PRIVATE, null);
        // Création des tables
        db.execSQL("DROP TABLE IF EXISTS Utilisateur;");
        db.execSQL("CREATE TABLE IF NOT EXISTS Utilisateur(Username VARCHAR,Password VARCHAR);");
    }
    /**
     * Création d'un utilisateur (utilisation rawQuery pour créer et pouvoir lire le tuple créé ensuite
     * @param utilisateur
     * @return
     */
    public Utilisateur createUtilisateur(Utilisateur utilisateur) {

        Cursor resultset = db.rawQuery("INSERT INTO Utilisateur VALUES(?, ?)",
                                       new String[]{utilisateur.getUsername(), utilisateur.getPassword()
        });
        resultset.moveToFirst();
        return retrieveUtilisateur(utilisateur.getUserName());
    }
```

Persistance en manuel avec SQLite (2.2)– Dao

```
/**
 * Obtention d'un utilisateur par son nom
 */
public Utilisateur retrieveUtilisateur(String userName) {
    Cursor resultset = db.rawQuery("SELECT * FROM Utilisateur WHERE username = ?", new String[]{userName});
    resultset.moveToFirst();
    return cursorToUtilisateur(resultset);
}

/**
 * Mise à jour d'un utilisateur
 */
public void updateUtilisateur(Utilisateur utilisateur) {
    db.execSQL("UPDATE Utilisateur SET password = ? WHERE username = ?",
                    new String[]{utilisateur.getPassword(), utilisateur.getUsername()}
    );
}

/**
 * Suppression d'un utilisateur
 * @param utilisateur
 */
public void deleteUtilisateur(Utilisateur utilisateur) {
    db.execSQL("DELETE FROM Utilisateur WHERE username = ?",
                    new String[]{utilisateur.getUsername()}
    );
}
```

Persistance en manuel avec SQLite (2.3)– Dao

```
/**
 * Retourne la liste de tous les utilisateurs
 */
public List<Utilisateur> getAllUtilisateurs() {
    List<Utilisateur> utilisateurs = new ArrayList<>();
    Cursor resultset = db.rawQuery("SELECT * FROM Utilisateur", null);
    resultset.moveToFirst();
    while(!resultset.isAfterLast()) {
        utilisateurs.add(cursorToUtilisateur(resultset));
        resultset.moveToNext();
    }
    return utilisateurs;
}

/**
 * Fonction pour instancier une entité utilisateur
 * à partir d'un tuple pointé par un Curseur ouvert
 */
private Utilisateur cursorToUtilisateur(Cursor cursor) {
    if(cursor.isBeforeFirst() || cursor.isAfterLast() || cursor.isClosed()) {
        return null;
    } else {
        String username = cursor.getString(0);
        String password = cursor.getString(1);
        Utilisateur utilisateur = new Utilisateur(username, password);
        return  utilisateur;
    }
}
```

```
/**
 * Fermeture de la Bdd
 */
public void close() {
    db.close();
}
```

Persistance en manuel avec SQLite (3)
Utilisation par une activité

```
private void demoPersistance() {
    utilisateurDao = new UtilisateurDao();
    utilisateurDao.open(this);

    System.out.println("*** Création ***");
    for(int i = 0; i < 10; i++) {
        utilisateurDao.createUtilisateur(new Utilisateur("User" + i, "Secret" + i));
    }

    List<Utilisateur> utilisateurs = utilisateurDao.getAllUtilisateurs();
    listerUtilisateurs(utilisateurs);

    System.out.println("*** Suppression ***");
    utilisateurDao.deleteUtilisateur(utilisateurs.get(0));

    utilisateurs = utilisateurDao.getAllUtilisateurs();
    listerUtilisateurs(utilisateurs);

    System.out.println("*** Update ***");
    utilisateurs.get(0).setPassword("terces");
    utilisateurDao.updateUtilisateur(utilisateurs.get(0));

    utilisateurs = utilisateurDao.getAllUtilisateurs();
    listerUtilisateurs(utilisateurs);
}
```

Persistance en manuel avec SQLite

- **Discussion :**
  - **Interêt ?**
  - **Avantages / Inconvénients**
  - **Complexité de mise en oeuvre …**
  - **Persistance complète ?**
  - **Fonctionnalités indispensables non abordées …**

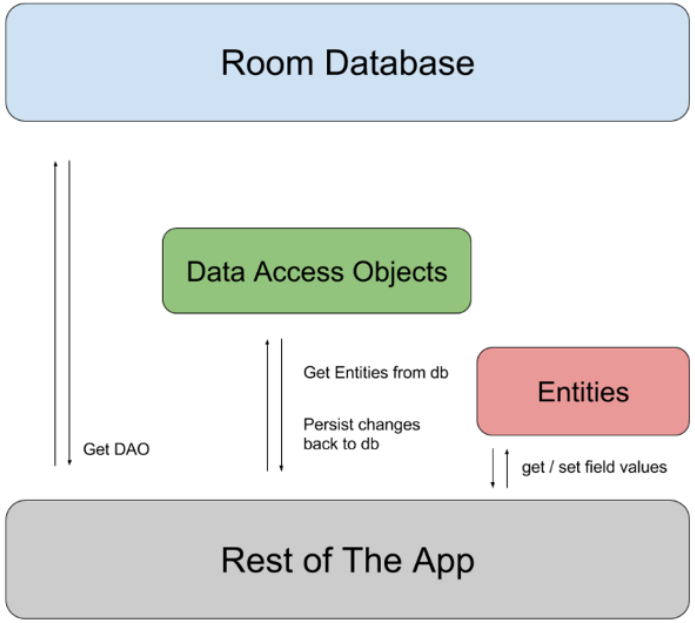
Pour plus de details :
https://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html

Room

- **Room Persistence Library :** fournit une couche d'abstraction au-dessus de SQLite et assure le Mapping Objet Relationnel.
- Room se configure avec des **annotations**
- Les codes permettant d'implanter les **fonctionnalités les plus classiques du MOR (CRUD) sont générées au moment du build**
- Les **requêtes spécifiques** à implanter en DAO sont définies dans **des Interfaces ou Classes abstraites**

Architecture Room

- **Room Database :** fournit l'accès à la base de données SQLite.
- **Entities :** Classes des Objets métiers à persister. Pour room chaque entité correspond à une table relationelle dans la BDD.
- **Data Access Objects :** contient toutes les méthodes pour persister ou lire les objets persistés.
- **Rest of The App :** L'application obtient auprès de l'instance de la Room Database les DAO nécessaires pour réaliser les cas d'utilisation.



Room : Mise en œuvre – config de gradle

- Dans le fichier **build.gradle** du module il faut ajouter les dépendances de Room :

```
dependencies {
    ...
    implementation  "android.arch.persistence.room:runtime:1.0.0"
    annotationProcessor  "android.arch.persistence.room:compiler:1.0.0"
    ...
}
```

- Dans le fichier **build.gradle** du projet il faut ajouter le repository google()

```
allprojects {
    repositories {
        google()
        jcenter()
    }
}
```

- Pour obtenir les dépendances lancer un Build une fois le gradle configuré.

Room : Mise en œuvre – Les entités (1)

- Une **Entité** Room est une simple classe représentant un objet métier qui :
  - Est annotée **`@Entity`**
  - Possède une propriété identifiante annotée **`@PrimaryKey`**
  - Fournit des accesseurs et modifieurs pour toutes propriétés
  - Expose Un constructeur par défaut

```
import android.arch.persistence.room.Entity;
import android.arch.persistence.room.PrimaryKey;
import android.support.annotation.NonNull;

@Entity(tableName = "table_utilisateur")
public class Utilisateur {
    @PrimaryKey
    @NonNull
    protected String username;
    protected String password;

    public Utilisateur() {}

    //...
}
```

Room : Mise en œuvre – Les entités (2)

- Pour représenter les associations ou relations entre entités :

  - Annotation **`@Embedded :`** pour embarquer une structure dans une entité ou le résultat d'une requête. *Les colonnes pour représenter la structure seront ajoutées à la table ou au résultat de la requête retournant « L'embarqueur »*

  https://developer.android.com/reference/androidx/room/Embedded

  - Annotation **`@Relation :`** pour déclarer le chargement d'un Set ou d'une List

```
@Relation(parentColumn = NomColPK_Cible, entityColumn = NomColFK, entity = ClasseCible)
```

  https://developer.android.com/reference/androidx/room/Relation

  - Article sur la manière de gérer les associations avec Room
  https://android.jlelse.eu/android-architecture-components-room-relationships-bf473510c14a

Room : Mise en œuvre – Les DAO

- Un **DAO** Room est spécifié par une Interface ou une classe Abstraite qui :
  - Est annotée **`@Dao`**
  - Possède des méthodes abstraites annotées pour le CRUD: **`@Insert, @Update, @Delete`**
  - Possède des méthodes requêtes spécifiques annotées avec `@Query` qui retournent des Entités

```
@Dao
public abstract class UtilisateurDao {
    @Insert
    public abstract void create(Utilisateur... utilisateurs);

    @Update
    public abstract void update(Utilisateur... utilisateurs);

    @Delete
    public abstract void delete(Utilisateur... utilisateurs);

    @Query("SELECT * FROM utilisateur WHERE username = :parmUsername")
    public abstract Utilisateur getUtilisateurs(String parmUsername);

    @Query("SELECT * FROM utilisateur")
    public abstract List<Utilisateur> getAllUtilisateurs();
}
```

Room : Mise en œuvre – La Room Database

- La Bdd Room est spécifiée par une classe Abstraite qui :
  - Hérite de **`RoomDatabase`**
  - Est annotée avec l'annotation **`@Database`** qui comporte en propriété la liste de toutes les classes d'entités et d'autres options de configuration
  - Déclare des accesseurs abstraits à tous les DAO

```
import android.arch.persistence.room.Database;

@Database(entities = {Utilisateur.class}, version = 1)
public abstract class Bdd extends RoomDatabase {
    public abstract UtilisateurDao getUtilisateurDao();
}
```

Room : Mise en œuvre
Utilisation par l'application

- L'utilisation se fait dans l'application en accédant aux DAO à partir de la BDD

```
private void demoPersistanceRoom() {
    // Supprime la Bdd (Pour Test)
    this.deleteDatabase(Bdd.DB_NAME);

    // Obtention de la Bdd (création éventuelle)
    Bdd database = Room.databaseBuilder(this, Bdd.class, Bdd.DB_NAME)
            .allowMainThreadQueries() // En Production on se l'interdit
            .build();
    // Obtention du DAO
    UtilisateurDao utilisateurDao = database.getUtilisateurDao();

    // Tout est prêt pour exploiter ROOM et utiliser la Persistance
    // ...

}
```

Persistance avec Room

- **Discussion :**
  - **Interêt ?**
  - **Avantages / Inconvénients**
  - **Complexité de mise en oeuvre …**
  - **Persistance complète ?**
  - **Fonctionnalités indispensables non abordées …**

Pour plus de details :

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

https://www.techiediaries.com/android-room-tutorial/