PROGRAMMATION DE COMPOSANTS MOBILES (ANDROID)

Wieslaw Zielonka

Bases de données avec Room

ajouter dans build.gradle (Module)

Ajouter dans build.gradle (Module) :

au début du fichier, (pour avoir accès à KSP) dans la section plugins ajouter

```
plugins {

        //ligne à ajouter
      id 'com.google.devtools.ksp' version "1.7.10-1.0.6"

}
```

D'autres versions du plugin sur la page
https://search.maven.org/artifact/com.google.devtools.ksp/symbol-processing-api

Il semble que la version du plugin ksp doit correspondre à la version du plugin kotlin dans
la section plugins de build.gradle (Project). Dans mon projet c'est

```
plugins{
   ...
       id 'org.jetbrains.kotlin.android' version '1.7.10' apply false
}
```

Si vous passez au kotlin 1.7.20 il convient de changer la version de ksp.

ajouter dans build.gradle (Module)

à la fin de fichier build.gradle dans la section dependencies ajouter :

```
dependencies{
    .....
        //Room
    def room_version = "2.4.3"
    implementation "androidx.room:room-ktx:$room_version"

    // To use Kotlin Symbol Processing (KSP)
    ksp "androidx.room:room-compiler:$room_version"

}
```

ajouter dans build.gradle (Module)

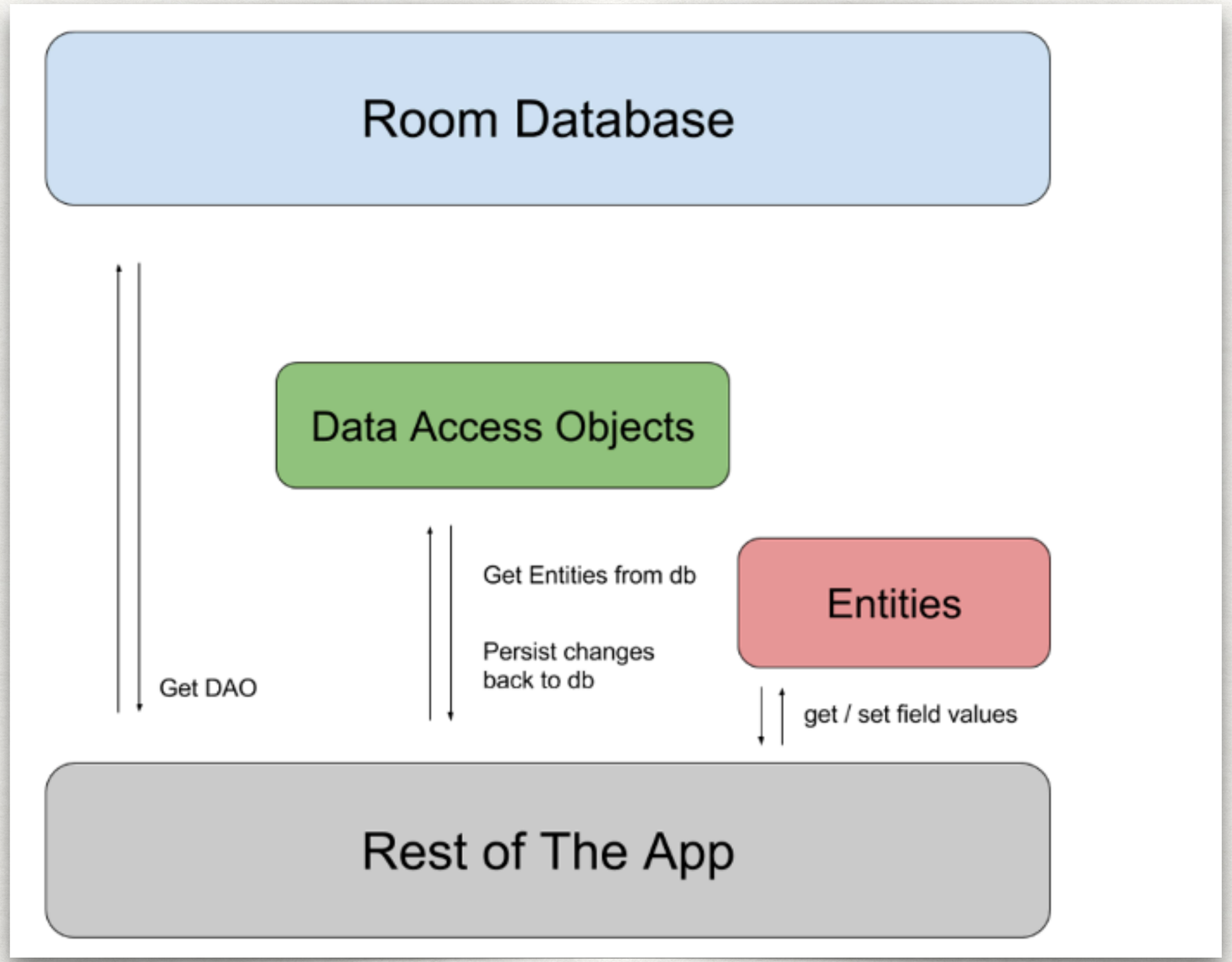
En plus dans la section dependencies ajouter les dépendances pour ViewModel et LiveData et si besoin pour RecyclerView

```
dependencies{

implementation 'androidx.recyclerview:recyclerview:1.2.1'

def lifecycle_version = '2.6.0-alpha02'
// ViewModel
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"
// LiveData
implementation "androidx.lifecycle:lifecycle-livedata-ktx:$lifecycle_version"

}
```

Room

- Définition de tables (objets de la classe Entity)
- Définition de requêtes (insert, update, delete, select), DAO - data access objects
- Définition de la base de données
- Exécution de requêtes

L'architecture de Room



ROOM

Définir les tables

Base de données : tables

Trois tables (entités Entity)

Author :

attributs (colonnes) : nom, prénom, id : Int

clé primaire : id

Book :

attributs : title , id: Int

clé primaire : id

La relation entre Author et Book est de type plusieurs à plusieurs :
un livre peut avoir plusieurs auteurs et un auteur peut écrire plusieurs livres.
Donc il nous faut une table intermédiaire.

En Android il est possible d'indiquer que la relation est type plusieurs à plusieurs pour que l'entité (table) intermédiaire soit automatiquement générée. Mais nous allons rester proche de SQL, nous construisons explicitement une table intermédiaire :

AuthorBook :

attributs : idAuthor foreign key references Author
idBook foreign key references Book

Base de données : tables

Author

nom	prénom	id
Hugo	Victor	9

Book

id	titre
22	Les misérables

AuthorBook

idAuthor	idBook
9	22

`id` de `Author` : colonne parent de `id` de `AuthorBook`

Définition d'entités (tables)

Dans un fichier séparé Entity.kt (mais ce n'est pas nécessaire) je déclare les entités que Room transformera en table :

```
import androïde.room.*

@Entity
data class Author(
    @PrimaryKey  var idAuthor: Long,
    var name: String,
    var firstName: String
)
```

- l'annotation @Entity désigne une entité, le nom de la classe = nom de la table
- data class : une classe "spéciale" pour les données (pas important pour nous)
- l'annotation @PrimaryKey désigne la clé primaire
- les noms de propriétés correspondent aux noms de colonnes de la table (les noms d'attributs)
- dans cet exemple toutes les propriétés définies dans le constructeur

Définition d'entités (tables)

```
@Entity( indices = [Index( value = [ "name", "firstName" ], unique = true  )])
data class Author(
    @PrimaryKey(autoGenerate = true)
    val idAuthor: Long = 0,
    var name: String?,
    var firstName: String?
)
```

La même entité redéfinie.
Une annotation peut avoir des propriétés. Ici la propriété `autoGenerate=true` de `@PrimaryKey` avec la valeur true indique que la clé primaire est générée automatiquement par la BD. La valeur initiale de `idAuthor` est 0

Les attributs de l'annotation `@Entity` indiquent que
le couple d'attributs `(name, firstName)` est unique. Cela nécessite la création d'index sur ce couple.

Création d'entités (tables)

```
@Entity
data class Book(val titre: String) {
    @PrimaryKey(autoGenerate = true)
    var idBook: Long = 0
}
```

L'entité Book

Définition d'entités (tables)

Si la clé primaire peut être composée de plusieurs attributs.
Dans ce cas elle sera définie par un attribut de l'annotation `@Entity`.

```
@Entity(primaryKeys = ["firstName", "lastName"] )
data class User(
    val firstName: String?,
    val lastName: String?
)
```

Création d'entités : table avec des clés étrangères

```
@Entity(
    primaryKeys = ["idAuthor", "idBook"],

    indices = [Index( value = [   "idBook"    ] ) ,

    foreignKeys = [ ForeignKey(
            entity = Book::class,
            parentColumns = ["idBook"],
            childColumns = [ "idBook" ],
            deferred = true,
            onDelete = ForeignKey.CASCADE
        ),
        ForeignKey(
            entity = Author::class,
            parentColumns = [ "id" ],
            childColumns = [ "idAuthor" ],
            deferred = true,
            onDelete = ForeignKey.CASCADE
        )]    )

)
data class AuthorBook(
    var idAuthor: Long,
    var idBook: Long
)
```

clé primaire composé de deux attributs

index sur la colonne idBook

L'attribut idBook de AuthorBook est une clé étrangère qui référence l'attribut idBook de Author.

L'attribut idBook peut seulement prendre des valeurs qui apparaissent dans la colonne idBook de Book.

Avec `onDelete` on peut spécifier l'action à prendre e
de suppression de valeur id dans la table Book,

`onDelete` prend une des valeurs :
ForeignKey.CASCADE ForeigKey.NONE,
ForeingKey.NO_ACTION, ForeignKey.SET_DEFAULT
et spécifie l'action en cas de suppression de l'entité parent

ROOM

Définir les raquettes

DAO : Data Access Object

Création de DAO (Data Access Object)

DAO est une interface où

- chaque fonction est annotée par une de quatre annotations : `@Insert, @Delete, @Update, @Query`
- l'interface lui même est annotée par `@Dao`

```
@Dao
interface MyDao {

    @Insert(onConflict = OnConflictStrategy.ABORT)
    fun insertBooks(vararg book: Book) : List<Long>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertAuthors(vararg author: Author) : List<Long>

    @Insert(onConflict = OnConflictStrategy.ABORT)
    fun insertAuthorsBooks(vararg ab: AuthorBook) : List<Long>
}
```

Création de DAO (Data Access 0bject)

```
@Dao
interface MyDao {

    @Insert(onConflict = OnConflictStrategy.ABORT)
    fun insertBooks(vararg book: Book) : List<Long>

}
```

- `vararg` pour une fonction à nombre variable d'arguments.
- L'attribut `onConflict` indique ce qu'il faut faire en cas de conflit avec les entités qui sont déjà dans la table (une entité avec la même valeur de clé primaire). Valeurs possibles :
  - **`OnConflictStrategy.ABORT,`**
  - **`OnConflictStrategy.IGNORE,`**
  - **`OnConflictStrategy.REPLACE.`**

  Si ABORT alors en cas de conflit une exception.
- insert retourne la liste d'ids de éléments insérés. Pour chaque élément qui n'a pas pu être inséré la liste contient la "clé" -1. Par exemple si on insère trois éléments mais seulement le deuxième est insérées mais pas les autres à cause de conflit, la fonction retournera la liste [-1, 25, -1] (à la place de 25 aaa clé réelle). 5identifiant ce n'est pas toujours la clé primaire, SQLite maintient les identifiants qu'il attribue aux enregistrements dans la table.

Création de DAO (data access object)

Dao pour les insertions d'éléments partiellement définis :

On définit une classe qui contient certains propriété sélectionnées de Author .Par exemple pour Author on supprime la clé:

```
data class AuthorName(
    val name : String,
    val firstName : String
)
```

et dans Dao :

La table dans laquelle on insère

```
@Dao
interface MyDao {

    @Insert(entity = Author::class,   onConflict = OnConflictStrategy.IGNORE)
    fun insertAuthors(vararg author: AuthorName) : List<Long>
```

Et ensuite on peut insérer par exemple comme :

```
insertAuthors( AuthorName( "Potocki", "Jan"), AuthorName("Lem","Stanislaw") )
```

La clé sera générée automatiquement.

Dao pour update

Un Dao pour updates. Je suppose que juste la table (entité) Authors est susceptible d'être modifié par update.

```
@Dao
interface MyDao {

    @Update
    fun updateAuthors(vararg authors: Author) : Int
```

L'annotation `@Update` indique qu'il s'agit de l'opération update. update utilise la clé primaire pour identifier les uplets à modifier. Si pas d'uplet avec la clé correspondante alors update ne fait pas de modification.
La valeur de retour (optionnelle) donne le nombre d'items (nombre de lignes) modifiés par update.

Dao pour delete

```
@Dao
interface MyDao {

@Delete
fun deleteAuthors( vararg authors: Author) : Int

@Delete
fun deleteAuthors( authors: MutableList<Author>) : Int

@Delete( entity = Author::class )
fun deleteSomeAuthors(authors: MutableList<Author>) : Int

@Delete
fun deleteBooks( books : List<Book> ) : Int
```

Dans ces Delete if faut construire les objets complets Author ou Book à supprimer.
Ce n'est pas très commode. On préfère de supprimer un objet juste on donnant sa clé.

Dao pour delete

Pour pouvoir supprimer des objets juste en spécifiant la clé il faut construire une nouvelle classe qui contient juste la clé.
Ici l'exemple pour la table Book.

On défini d'abord une nouvelle entité qui contient juste l'attribut qui sert de clé dans l'entité Book :

```
class BookInfo(
    val idBook: Long
)
```

et dans Dao :

la table dans laquelle on supprime

```
@Dao
interface MyDao {


@Delete( entity = Book::class )
fun deleteSomeBooks(  books : List<BookInfo> ) : Int

}
```

La valeur de retour (optionnelle) donne le nombre de lignes supprimées.

Dao pour select

- L'annotation `@Query` contient une requête select
- La fonction qui exécute select retourne un Array d'objets Entity englobé par un LiveData

```
@Dao
interface MyDao {
    @Query("SELECT * FROM Author")
    fun loadAllAuthors(): LiveData<Array<Author>>

    @Query("SELECT * FROM Book")
    fun loadAllBooks(): LiveData<Array<Book>>

//requêtes paramétrées.

    @Query("SELECT * FROM Author WHERE nom = :nom")
    fun loadAuthor(nom: String): LiveData<Array<Author>>

   //chercher par le préfix de nom
    @Query("SELECT * FROM Author WHERE nom like :nom ll '%' ")
    fun loadSomeAuthors(nom: String): LiveData<Array<Author>>
```

nom de colonne de Author

nom de paramètre de la fonction loadAuthors() précédé par :

Dao pour select (suite)

Android peut empaqueté le résultat dans l'objet LiveData. Une activité pourra s'enregistrer comme observer pour suivre les modifications. Par exemple si une de table Author ou Book change le contenu alors automatiquement l'objet LiveData retourné par la fonction loadAuthorsBook change la valeur contenu sans qu'on soit obligé de refaire la requête SELECT.

Dao pour select (suite)

```
@Query(
    "SELECT nom, prenom, titre FROM Author NATURAL JOIN  AuthorBook NATURAL JOIN Book"
)
fun loadFullAuthorsBooks(): LiveData<List<FullInfo>>
```

Le dernier select ne retourne pas un tableau d'objet Entity mais des objets qui contiennent (nom, prénom, tire)
qui viennent de deux tables différentes. Il faut définir une classe auxiliaire qui contient ces trois propriétés:

```
class FullInfo(
    val nom: String?,
    val prenom: String?,
    val titre: String
)
```

Dao pour select (suite)

```
@Query( "SELECT title, idBook FROM  Book NATURAL JOIN  AuthorBook
WHERE  " +
        " AuthorBook.idAuthor = :idAuthor ")

fun loadAllBooksOf( idAuthor : Long ) :
MutableLiveData<List<Book>>
```

Notez que le dernier SELECT retourne une liste et non pas un Array. De plus la liste est empaquetée dans un objet LiveData. Conclusion : nous avons le choix : obtenir les résultats sous forme de Array ou de List

```
@Query("SELECT * FROM user WHERE uid IN (:userIds)")
    fun loadAllByIds(userIds: IntArray): List<User>
```

Dans le dernier SELECT on cherche des "user"s en utilisant leurs identifiants.

Chaque table possède un attribut uid qui contient l'identifiant d'une ligne de table.

ROOM

Construire la base de données

Une classe pour représenter la base de données

```
@Database(entities = [Author::class, Book::class, AuthorBook::class], version = 6)
abstract class BooksDatabase : RoomDatabase() {
    abstract fun myDao(): MyDao

    companion object {
        @Volatile
        private var instance: BooksDatabase? = null

        fun getDatabase( context : Context ): BooksDatabase{
            if( instance != null )
                return instance!!
            val db = Room.databaseBuilder( context.applicationContext,
                                    BooksDatabase::class.java , "books")
                    .fallbackToDestructiveMigration()
                    .build()
            instance = db
            return instance!!
        }
```

méthode abstrait qui retourne mon DAO

version de la BD

le nom de la bd

- l'annotation `@Database` pour indiquer que la class définit la BD
- la propriété `entities` de l'annotation : un Array des classes d'entités (de tables)
- `BooksDatabase` : le nom de la BD, à vous de choisir, la classe abstraite dérivée de `RoomDatabase`

- **`compagnon object`** est l'objet unique attaché à une classe (un singleton). Dans la variable privée **`instance`** on stocke une référence vers la BD. Une fois instance initialisée on retourne la référence mémorisée dans instance.

Garder une seule connexion à la BD

Ouvrir une BD prend beaucoup de ressources et de temps. Il faut faire en sorte qu'on préserve la connexion durant l'exécution de l'application.
Pour cela il suffit d'ajouter la référence vers la BD dans Application :

```
class BookApplication : Application() {

    val database by lazy{
        BooksDatabase.getDatabase(this)
    }
}
```

Il faut indiquer à Android que la classe qu'on vient de définir c'est notre application. Pour cela on modifie le fichier AndoidManifest.xml en ajoutant l'attribut **android:name** avec le nom de notre classe dans la balise **application:**

```
<application
    android:name=".BookApplication"
```

Dans le ViewModel nous pouvons maintenant récupérer la référence vers le DAO :

```
val dao = (application as BookApplication).database.myDao()
```

ROOM

Exécuter les requêtes

ROOM

les requêtes SELECT

Exécuter les requêtes SELECT

Les requêtes SELECT peuvent prendre beaucoup de temps. Pour éviter qu'un SELECT bloque l'interface graphique Android interdit d'exécuter les requêtes SELECT dans le thread principal d'activité. Le remède : les fonctions qui implémentent les SELECT dans DAO doivent retourner les objets de type LiveData au lieu de retourner directement le résultat.

```
@Dao
interface MyDao{
   @Query("SELECT * FROM Author")
    fun loadAllAuthors(): LiveData<List<Author>>

}
```

dans DAO retourner tjrs un LiveData

```
class AddAuthorViewModel(application: Application) : AndroidViewModel(application) {
        val dao = (application as BookApplication).database.myDao()
    fun loadAllAuthors() = dao.loadAllAuthors()
}
```

dans le modèle

```
class AddAuthorsActivity : AppCompatActivity() {
val model by lazy {
    ViewModelProvider(this).get(AddAuthorViewModel::class.java) }

et dans onCreate créer un observer qui sera activé quand le contenu de table
change
    model.loadAllAuthors().observe(this) {
    Log.d(TAG, "nouvelle liste auteurs")
    adapter.setAuthors(it, model.selectedAuthors)
}
```

dans Activity observer LiveData

les requêtes INSERT

Exécuter les requêtes INSERT(DAO)

```
@Dao
interface MyDao {

    @Insert(entity = Author::class, onConflict = OnConflictStrategy.IGNORE)
    fun insertAuthors(vararg authors: AuthorName) : List<Long>
}
```

Insert retourne une liste d'identifiants d'éléments insérés. Si un élément n'a pas été inséré (problème d'insertion) à la place de son identifiant figure la valeur -1L. Par exemple si au retour de

insertAuthors(a1,a2,a3,a4)

on obtient la liste

(-1,5,-1,8)

alors sur 4 éléments à insérer on a réussi à insérer deux.

Exécuter les requêtes INSERT (ViewModel)

```
class AddAuthorViewModel(application: Application) :
AndroidViewModel(application) {
    val dao = (application as BookApplication).database.myDao()

    val insertInfo = MutableLiveData<Int>(0)
    fun insertAuthors(vararg authors: AuthorName) {
      Thread {
          val l = dao.insertAuthors( *authors )
          insertInfo.postValue( l.fold(0)
           { acc: Int, n: Long -> if (n >= 0) acc + 1 else
acc } )
          }.start()
    }
```

L'expression :

```
  l.fold(0){ acc: Int, n: Long -> if (n >= 0) acc + 1 else acc }
```

donne le nombre d'éléments différents de -1L sur la liste.

```
insertInfo.postValue( nouvelle_valeur )
```

exécutée dans un thread différent de thread principal conduit à la modification de contenu de insertInfo dans le thread principal.

Exécuter les requêtes INSERT (ViewModel)

```
class AddAuthorViewModel(application: Application) :
AndroidViewModel(application) {
    val dao = (application as BookApplication).database.myDao()

    val insertInfo = MutableLiveData<Int>(0)
    fun insertAuthors(vararg authors: AuthorName) {
      Thread {
          val l = dao.insertAuthors( *authors )
          insertInfo.value = nouvelle_valeur    //NO
          .start()
      }
```

Le code ci-dessus est incorrect, la valeur contenu dans MutableLiveData doit être modifiée dans le thread principale. Il faut utiliser la méthode postValue :

```
insertInfo.postvalue(  nouvelle_valeur ) //OK
```

Exécuter les requêtes INSERT (Activity)

Dans l'activité l'insertion est déléguée au ViewModel :

```
model.insertAuthors( AuthorName(n,p) )
```

Si nous voulons de savoir si l'insertion a réussi il faut installer un observer (sans doute dans onCreate():

```
model.insertInfo.observe(this){
    Toast.makeText(this, "inserer $it élément(s)",
                   Toast.LENGTH_SHORT)
         .show()
}
```

les requêtes DELETE

Exécuter les requêtes DELETE (Dao)

```
@Dao
interface MyDao{

  @Delete
  fun deleteAuthor( author : Author ) : Int

}
```

la fonction retourne le nombre d'éléments supprimés (ici soit 0 soit 1).

Pour DELETE avec une condition WHERE il faut utiliser @Query

Exécuter les requêtes DELETE (dans le ViewModel)

```
val deleteResult : MutableLiveData<Int> =
MutableLiveData()

fun delete(a: Author) {

  thread{
     val i = dao.delete( a )
     deleteResult.postValue( i )
     ...
  }

}
```

On met le résultat de DELETE dans un objet LiveData pour l'affichage dans l'activité.

Exécuter les requêtes DELETE (dans l'Activité)

```
override fun onCreate( savedInstanceState : Bundle? ){

    viewModel.deleteResult.observe(this) {
    val txt = if (it == 0) "aucun élément supprimé"
              else "suppression réussie"
    Toast.makeText(this, txt, Toast.LENGTH_LONG).show()
    }

}
```

petit problème à régler : Toast s'affiche au lancement de l'activité et quand on tourne l'appareil même si aucun DELETE n'a été exécuté.

(Pourquoi ?)

les requêtes UPDATE

Exécuter les requêtes DELETE (Dao)

```
@Dao
interface MyDao{

   @Update
   fun updateAuthor( author : Author ) : Int

}
```

la fonction retourne le nombre d'éléments modifiés (ici soit 0 soit 1).

Impossible de modifier la clé primaire avec cette méthode.

Exécuter les requêtes UPDATEZ (dans le ViewModel)

```
val updateResult : MutableLiveData<Int> =
MutableLiveData()

fun update(a: Author) {

  thread{
     val i = dao.update( a )
     updateResult.postValue( i )
     ...
  }

}
```

On met le résultat de UPDATE dans un objet LiveData pour l'affichage dans l'activité.

Exécuter les requêtes DELETE (dans l'Activité)

```
override fun onCreate( savedInstanceState : Bundle? ){

    viewModel.updateResult.observe(this) {
    val txt = if (it == 0) "aucun élément modifié"
              else "modification réussie"
    Toast.makeText(this, txt, Toast.LENGTH_LONG).show()
    }

}
```

petit problème à régler : Toast s'affiche au lancement de l'activité et quand on tourne l'appareil même si aucun UPDATE n'a été exécuté.

Remède ?

ROOM

les requêtes SELECT paramétrées

solution 1

SELECT paramétré (Dao)

solution 1

```
@Dao
interface MyDao{

 @Query("SELECT * FROM Auteur WHERE nom LIKE :nom||'%'")
 fun loadPartialName(nom: String): List<Pays>

}
```

Avec un changement de paramètre nom il y a chaque fois une nouvelle requête. Dans cette solution on retourne une liste qui n'est pas englobée dans LiveData.

SELECT paramétré (ViewModel) solution 1

```
// ViewModel
var certainsAuteurs = MutableLiveData<List<Auteur>>()

 fun loadPartialName(nom: String) {

     thread { certainsAuteurs.postValue(dao.loadPartialName(nom)) }
 }
```

Le fait que loadPartialNamer() des DAO retourne une liste sans LiveData nous oblige d'exécuter la requête dans un nouveau thread.

---

certainsAuteurs est une référence vers un objet MutableLiveData. Cette référence ne change jamais mais le contenu de MutableLiveData change à chaque fois quand on lance SELECT avec un nouveau paramètre

SELECT paramétré (Activity)

solution 1

Dans l'activité il suffit d'installer un observer :

```
viewModel.certainsAuteurs.observe(this){
    adapter.setListAuteurs(it)
}
```

Problème : si dans la même activité on exécute DELETE, UPDATE, INSERT alors après chacune de ces opération il faut **refaire la requête SELECT avec le dernier paramètre.**

En effet, la requête SELECT qui retourne une Liste ne sera pas réexécutée automatiquement quand le contenu de la table change.
Il faut relancer la requête SELECT nous même.

ROOM

les requêtes SELECT paramétrées

solution 2

SELECT paramétré (Dao)

solution 2

```
@Dao
interface MyDao{

 @Query("SELECT * FROM Auteur WHERE nom LIKE :nom||'%'")
 fun loadPartialName(nom: String): Live<List<Pays>>

}
```

Avec un changement de paramètre nom il y a chaque fois une nouvelle requête, donc un nouveau objet LiveData

SELECT paramétré (ViewModel) solution 2

```
 var resultatSelect : LiveData<Author>? = null
fun loadPartialName(nom: String) {
    resultatSelect = dar.loadPartialName( nom )
    return resultatSelect
}
```

Plus besoin de faire la requête dans un nouveau thread, l'objet LiveData le fera pour nous.

SELECT paramétré (Activity) solution 2

Chaque fois quand le paramètre de recherche change il faut installer un nouveau observer.

Mais d'abord il faut installer le premier observer :

```
override fun onCreate(savedInstanceState: Bundle?) {
.........
/* savedInstanceState est null au lancement de l'activité
 * et il faut faire le SELECT initial sinon on commencera avec la liste vide */

if ( saveInstanceState == null )
     loadPartialName( viewModel.prefixe )
viewModel.resultatSelect.observe(this){ adapter.setList( it ) }

}
```

Et chaque fois quand on refait SELECT avec un nouveau paramètre (sans doute dans un listener):

```
viewModel.resultatSelect.removeObservers( this@RechercheActivity )
viewModel.loadPartialName( nouveau_prefixe )
viewModel.resultatSelect.observe( this@RechercheActivity )
                                { adapter.setList( it ) }
```

Résumé : chaque fois quand on change préfixe il y a un nouveau LiveData donc il faut réinstaller observer et faire oublié le précédent.