Base de données embarquée : ORM Room

Room est un ORM (object relationnal mapping) dont le rôle est d'introduire une couche d'abstraction entre votre application et une base de données relationnelle.

L'objectif d'un tel composant est de masquer le fait qu'une base de données relationnelle est utilisée pour stocker les informations, et de manipuler à la place des objets métiers au travers de méthodes dédiées. Le développeur utilisant l'ORM ne devra pas exécuter de requête SQL, c'est l'ORM qui le fera pour lui de manière transparente.

Room prend en charge les types Observable et LiveData qui permettent d'effectuer des opérations de manière asynchrone. On évitera ainsi de bloquer l'interface lors des opérations en base.
Ces types disposent d'un mécanisme de notification qui permettent d'informer la vue de modification de données en base, et permettre ainsi de déclencher à l'aide d'évènements la mise à jour automatique des données affichés dans les composants graphiques.

Soit dans l'architecture MVVM :

- **Modèle (Room)** : Room fournit des Observables (Observable, LiveData) pour exécuter des requêtes. Les résultats des requêtes sont automatiquement émis lorsque la base de données change.
- **ViewModel** : Le ViewModel souscrit à ces Observables. Elle transforme les données au besoin (par exemple, en filtrant ou en formatant) avant de les exposer à la View.
- **View** : La View observe les données exposées par la ViewModel (par exemple, via LiveData.observe()) et met à jour automatiquement l'interface utilisateur.

1. Préparation du projet

Afin que votre projet puisse utiliser les fonctionnalités de l'ORM (annotations, génération de code) il faut lui ajouter certaines bibliothèques.

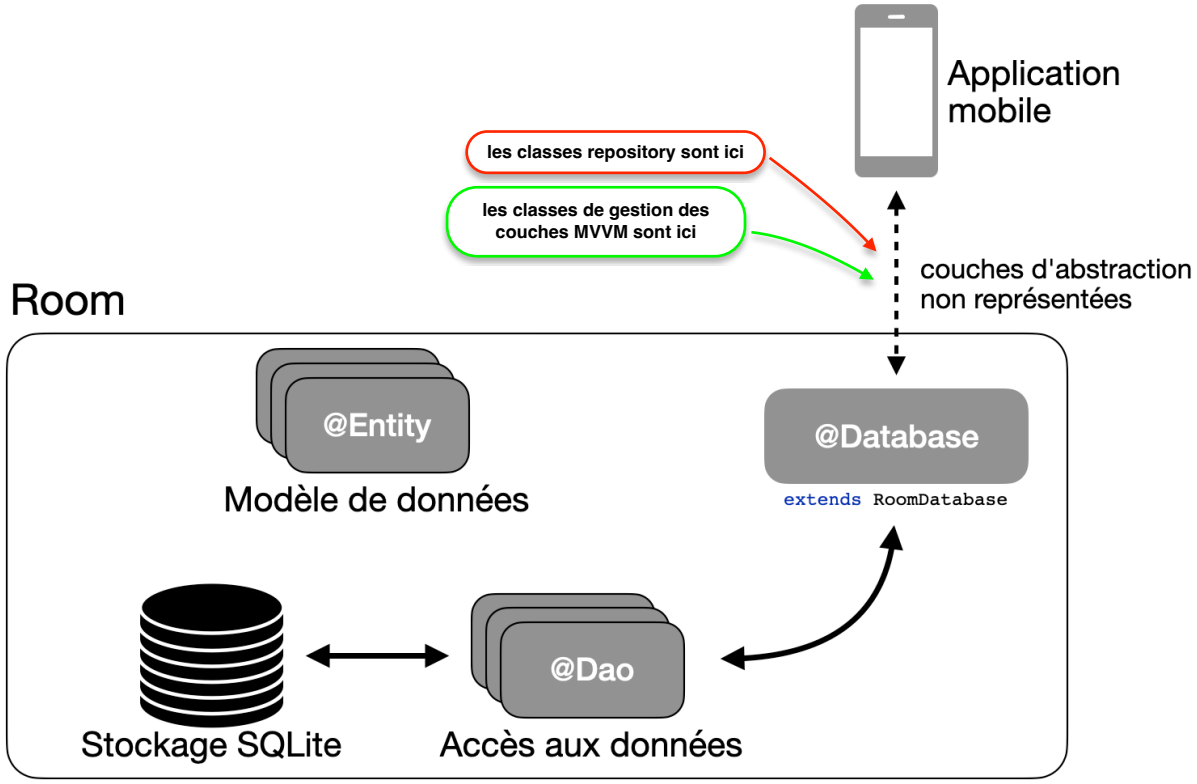
Ajouter ces dépendances au fichier build.gradle.kts (Module:app)

```
dependencies {
    [...]
val room_version = "2.6.1"

// bibliothèques permettant d'intéragir avec une BD SQLite
implementation("androidx.room:room-runtime:$room_version")


// permet de prendre en compte les annotations Room
annotationProcessor("androidx.room:room-compiler:$room_version")


// permettant d'utiliser des types réactifs comme Observable
// utiles lors des accès asynchrones
implementation("androidx.room:room-rxjava2:$room_version")
implementation("androidx.room:room-rxjava3:$room_version")

}
```

Documentation :
https://developer.android.com/training/data-storage/room?hl=fr#kts

Remarques :
Ce code est utilisable dans un fichier build.gradle.kts dont la syntaxe reprend celle de Kotlin, pour un fichier build.gradle classique il faudra l'adapter.

2. Composants de l'ORM Room



Architecture de stockage de données avec Room

Le stockage d'informations dans une base de données à l'aide de Room se fait avec un ensemble de composants :

- Les Entity : classes métiers permettant de définir le schéma de la base
- Les DAO : classes contenant les méthodes de gestion de la base de données
- Une implémentation de RoomDatabase : point d'accès unique à la base de données

2.1. Les Entity

Creer 1 classe métier par table de la base de données (Classe Entity). On parle de classes POJO pour Plain Old Java Object.
On retrouve aussi le terme DTO. Les classes DTO (Data Transfert Object) sont définies pour permettre les échanges entre les couches d'une application. Les DTO sont parfois une version spécifique des classes métiers (pour masquer certaines informations, le mot de passe d'un utilisateur par exemple).

Les spécificités (contraintes de clé, attribut de table, etc) seront ajoutées par des annotations dans le code java de chaque classe métier
La structure de ces classes correspond à la structure des tables dans la base de données, il leur faudra, un champ par attribut, un constructeur sans paramètre, une méthode toString() pour l'affichage et le debuggage.

On retrouve les annotations suivantes : @Entity, @PrimaryKey, @ForeignKey qui permettent de déclarer les caractéristiques propres au monde relationnel.

➔ Voir l'exemple de code en annexe 1.

2.2. Les classes DAO

Les classes DAO (Data Access Object) contiennent les spécifications des méthodes d'accès aux tables de la base de données.
Il faudra créer une classe DAO par table (une classe DAO par classe métier).

D'une manière générale, les classes DAO contiennent les méthodes CRUD (Create, Retreive, Update, Delete) qui sont les opérations de base sur une table. On pourra aussi y ajouter des méthodes spécialisées (avec des restrictions, des requêtes multi-tables, des jointures, etc.)

Room permet à l'aide d'annotations de générer du code de manière transparente, on aura juste à déclarer les requêtes à exécuter et/ou le type de requête et room écrira le corps des méthodes à votre place.

C'est dans ces méthodes qu'on utilise le type LiveData qui permet de se tenir informé de changements dans la base de données et ainsi mettre à jour l'affichage automatiquement.
On parle de mécanisme de Callback : c'est la modification de la base de données qui fait exécuter un code d'actualisation de l'affichage.

Les méthodes de récupération de données retournent des LiveData d'objets métiers dont le contenu est synchronisé avec la base de données.

➔ Exemple extrait de l'annexe 2 :

```
@Query("select * from produit where id = :unId")
LiveData<Produit> getProduitById(long unId);
```

On retrouve dans les classes DAO les annotations suivantes : @Dao, @Query, @Insert, @Delete, @Transaction, etc.

Documentation :
https://developer.android.com/training/data-storage/room/accessing-data?hl=fr#java)

2.3. Point d'accès à la base de données

Une classe héritant de RoomDatabase (AppDatabase dans l'exemple) donne accès à la base de données à l'aide d'un singleton.
Cette classe permet de spécifier les classes métiers à prendre en compte, la version et le nom de la base de données, le lieu de stockage et les classes DAO

Dans l'exemple on laisse le système choisir où enregistrer la BD ce qui génère un warning lors de la compilation.

➔ Voir l'exemple de code en annexe 3.

2.4. Liens entre tables

Les liaisons entre tables sont à spécifier à l'aide d'annotations dans les Entity pour la structure de la base de données, et dans des classes Entity spécialisées pour la récupération de données liées.

Exemple avec un lien one-to-many :

- Dans la classe métier : spécifier l'attribut clé étrangère comme un attribut classique
- Dans l'annotation de l'Entity : décrire la clé étrangère

```
@Entity(foreignKeys = @ForeignKey(entity = Categorie.class,
        parentColumns = "idC",
        childColumns = "idCProduit"))
public class Produit {
```

- En option : créer une nouvelle classe faisant le lien (exemple en annexe 4) : cette classe vous permettra de récupérer des instances contenant la liaison entre 2 tables

Les contraintes de clé étrangères peuvent être spécifiées dans les caractéristiques de la classe :

```
@Entity(foreignKeys = @ForeignKey(entity = Categorie.class,
        parentColumns = "idC",
        childColumns = "idCProduit"))
public class Produit {
...
}
```

Documentation :
lien one-to-many

➔ https://developer.android.com/training/data-storage/room/relationships/one-to-many

Liaison many-to-many

➔ https://developer.android.com/training/data-storage/room/relationships/many-to-many

Exemple de liaison

➔ https://medium.com/android-news/android-architecture-components-room-relationships-bf473510c14a

2.5. Génération de la base et des classes d'accès

Les classes métiers et DAO ainsi créées permettent à ROOM de générer le code SQL et Java de création de la base de données, et d'accès aux tables (méthodes CRUD).
Si on modifie le schéma de la base, il faut regénérer ce code SQL et Java : menu build->Clean project, puis rebuild project.
De même sur une application en test il faudra supprimer le contenu de la base (supprimer l'application ou supprimer les données d'une application installée) pour que l'application crée à nouveau la base de données.

3. Test et jeu d'essai

Les codes présentés ici sont disponibles dans le code source fourni en ressources.

3.1. Création du jeu d'essai

La création de l'instance d'AppDatabase a pour effet la création des tables de la base de données. Dans la méthode onCreate() de MainActivity :

```
AppDatabase db = AppDatabase.getDatabase(this);
```

Utiliser l'ExecutorService de la classe AppDatabase pour exécuter des requêtes d'insertion :

```
db.databaseWriteExecutor.execute(() -> {
    db.categorieDao().insert(new Categorie(1,"periph usb"));
    db.categorieDao().insert(new Categorie(2,"ecran"));

    db.produitDao().insert(new Produit(1,"clé USB","AF6UO",10.5,1));
    db.produitDao().insert(new Produit(2,"clavier USB","BF7UO",34.99,1));
    db.produitDao().insert(new Produit(3,"ecran LCD 24
pouces","DISP3F",135.99,2));
        }
);
```

Ces requêtes sont exécutées de manière asynchrone dans un thread parallèle à celui du thread d'affichage de l'interface. Elles sont donc non bloquantes et n'entrainent pas de paralysie de l'interface graphique.

⚠ Ces requêtes ne pourront être exécutées qu'une seule fois sans générer d'erreur. Une 2eme exécution provoquera une erreur de contrainte d'unicité de clé primaire.

3.2. Test d'accès aux données

Test de récupération de données à l'aide des classes DAO :

```
db.produitDao().getAll().observe(MainActivity.this,lesProduits -> {
    Log.d("testLog",lesProduits.toString());
});
```

Dans cet exemple, on appelle la méthode getAll() de la classe DAO de Produit. La requête est exécutée de manière asynchrone et son résultat est accessible dans le LiveData retourné.
Le LiveData possède la méthode observe prenant en paramètre :

- L'activity ayant fait l'appel
- Une fonction anonyme prenant en paramètre la variable qui contiendra le type embarqué dans le LiveData indiqué à la déclaration de la méthode - ici : un type List<Produit>

Le paramètre et code de la fonction anonyme sont spécifié à l'aide de la syntaxe suivante :

```
(paramètre) -> { code }
```

Ici un affichage en console est effectué dans le logcat :

```
db.produitDao().getAll().observe(MainActivity.this,lesProduits -> {
    Log.d("testLog",lesProduits.toString());
});
```

Explications de la syntaxe :
Les 2 syntaxes suivantes sont équivalentes.

```
db.produitDao().getAll().observe(MainActivity.this,lesProduits -> {
    Log.d("testLog",lesProduits.toString());
});
```

```
db.produitDao().getAll().observe(MainActivity.this, new
Observer<List<Produit>>(){
    @Override
    public void onChanged(List<Produit> lesProduits) {
        Log.d("testLog",lesProduits.toString());
    }
});
```

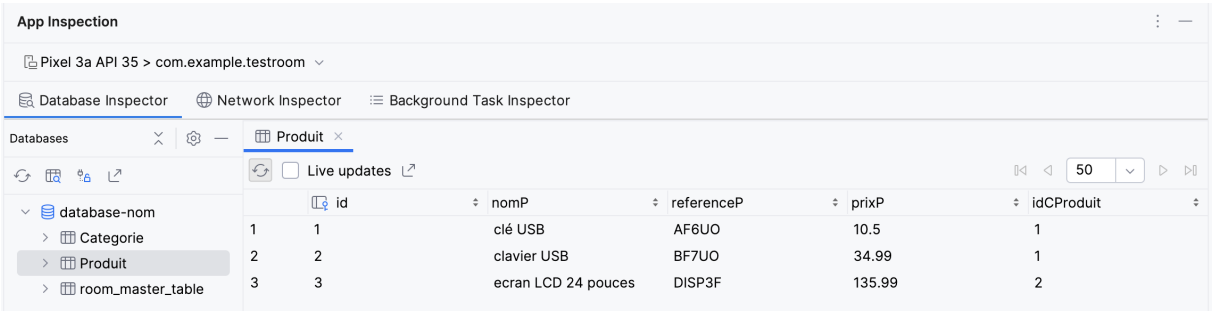
Le code en gras montre l'utilisation de la syntaxe condensée avec le symbole -> {}
Le code de la syntaxe condensée est ainsi plus lisible. On retrouve cette syntaxe à chaque fois que la surcharge (override) d'une méthode d'une interface est effectuée. (Notamment lors de la prise en charge d'évènements) :

Exemple :

```
bouton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        // placer ici le code à exécuter lors du clic
    }
});
```

```
bouton.setOnClickListener((view)->{
    // placer ici le code à exécuter lors du clic
});
```

3.3. Exploration de la base de données embarquée depuis Android Studio



View -> tool windows -> app inspection

Par défaut la base de données n'est accessible que lorsque l'application est en cours d'exécution.

4. Repository

Le rôle des classes repository est d'orienter vers diverses sources de données (données en base, en webservice, sous forme de fichiers, de ressources, etc). La classe masque le choix de la source de données au développeur qui l'utilise.

Plusieurs approches sont possibles :

- Créer classe repository par table : bonne segmentation, réutilisation possible, complexité lors de l'utilisation simultanée de plusieurs tables
- Créer un repository regroupant plusieurs tables pour simplifier l'architecture de l'application, au détriment cependant de son évolutivité.

Contrairement à la phase de test où le singleton était créé dans MainActivity, en production, il sera créé dans la classe repository. C'est à partir de cette classe qu'on utilise l'accès DAO.
Extrait du constructeur du repository :

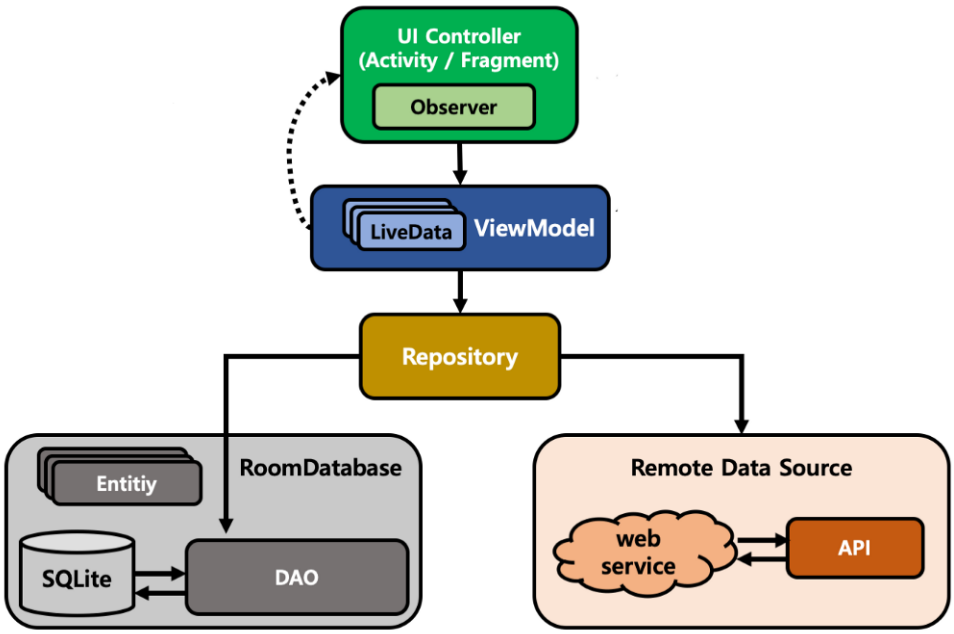
```
public ProduitRepository(Application application) {
    AppDatabase db = AppDatabase.getDatabase(application);
    mProduitDao = db.produitDao();

}
```

Dans cette classe on retrouve les mêmes méthodes (CRUD) que dans la classe DAO. Cette classe sert d'intermédiaire entre la classe UI (MainActivity dans l'exemple) et la classe DAO.

➔ Voir l'exemple de code en annexe 5.

Tutoriel :
https://developer.android.com/codelabs/android-room-with-a-view#8

5. Exemple dans l'architecture MVVM



https://medium.com/androidmood/comprendre-larchitecture-mvvm-sur-android-aa285e4fe9dd

➔ Gestion des données affichées dans l'application : ViewModel

Le ViewModel contient tous les traitements qui s'effectuent dans le but de servir la vue, il est important de noter que chaque Activité/Fragment possède un seul ViewModel.

MVVM repose sur la programmation réactive d'où la notion de LiveData qui sont des objets *observables,* c'est à dire qu'ils permettent d'écouter les changements de comportements du ViewModel afin de traiter et communiquer les données à la vue.
L'observable créé dans la vue se chargera de mettre à jour l'affichage lorsque qu'il recevra un message indiquant que les données ont été mises à jour dans la base.

Exemple : dans la méthode onCreate() de MainActivity (Annexe 7)

```
mMainActivityProduitViewModel = new
      ViewModelProvider(this).get(MainActivityProduitViewModel.class);

mMainActivityProduitViewModel.getUnProduit(2).observe(this, produit -> {
      if (produit != null) {
          unTextView.setText(produit.nomP);
      }
});
```

L'objet `mMainActivityProduitViewModel` contient les données affichées sous forme de LiveData, et les méthodes d'accès par le repository.
La création d'un observable permet d'automatiser la mise à jour de l'affichage lors d'un changement du résultat de la requête exécutée par getUnProduit() :

```
mMainActivityProduitViewModel.getUnProduit(2).observe(...)
```

à chaque changement de résultat, le code associé est exécuté :

```
[...]observe(this, produit -> {
      unTextView.setText(produit.nomP);
});
```

Annexe 1 : classes métiers

```
@Entity
public class Categorie {
    @PrimaryKey public long idC;
    public String nomC;


    public Categorie(long idC, String nomC) {
        this.idC = idC;
        this.nomC = nomC;
    }

    @Override
    public String toString() {
        return "Categorie{" +
                "idC=" + idC +
                ", nomC='" + nomC + '\'' +
                '}';
    }
}
```

```
@Entity(foreignKeys = @ForeignKey(entity = Categorie.class,
        parentColumns = "idC",
        childColumns = "idCProduit"))
public class Produit {
    @PrimaryKey public long id;
    public String nomP;
    public String referenceP;
    public double prixP;
    public long idCProduit;

    public Produit(long id, String nomP, String referenceP, double prixP,
long idCProduit) {
        this.id = id;
        this.nomP = nomP;
        this.referenceP = referenceP;
        this.prixP = prixP;
        this.idCProduit = idCProduit;
    }

    @Override
    public String toString() {
        return "Produit{" +
                "id=" + id +
                ", nomP='" + nomP + '\'' +
                ", referenceP='" + referenceP + '\'' +
                ", prixP=" + prixP +
                ", IdCProduit=" + idCProduit +
                '}';
    }
}
```

Annexe 2 : classes DAO

```
@Dao
public interface CategorieDao {
    @Query("SELECT * FROM categorie")
    LiveData<List<Categorie>> getAll();

    @Query("select * from categorie where idC = :unId")
    LiveData<Categorie> getCategorieById(int unId);

    @Insert
    void insert(Categorie uneCategorie);

    @Delete
    void delete(Categorie uneCategorie);


    @Transaction // on utilise une transaction car cette méthode exécute 2
                 // requêtes
    @Query("SELECT * FROM Categorie")
    public LiveData<List<CategorieWithProduits>>
getCategoriesWithProduits();
}
```

```
@Dao
public interface ProduitDao {
    @Query("SELECT * FROM produit")
    LiveData<List<Produit>> getAll();

    @Query("select * from produit where id = :unId")
    LiveData<Produit> getProduitById(long unId);

    @Insert
    void insert(Produit unProduit);

    @Delete
    void delete(Produit unProduit);
}
```

Annexe 3 : Point d'accès à la base de données

```
@Database(entities = {Produit.class, Categorie.class}, version = 2)
public abstract class AppDatabase extends RoomDatabase {
    public abstract ProduitDao produitDao();
    public abstract CategorieDao categorieDao();

    private static volatile AppDatabase INSTANCE;

    private static final int NUMBER_OF_THREADS = 4;

    static final ExecutorService databaseWriteExecutor =
            Executors.newFixedThreadPool(NUMBER_OF_THREADS);

    static AppDatabase getDatabase(final Context context) {
        // on ne crée pas le singleton car il existe deja
        if (INSTANCE == null) {

            synchronized (AppDatabase.class) {

                if (INSTANCE == null) {
                    INSTANCE = Room.databaseBuilder(
                                    context.getApplicationContext(),
                                    AppDatabase.class, "base-nom"
                                    ).build();
                }
            }
        }
        return INSTANCE;
    }
}
```

Lister toutes les classes métiers entre accolades

Singleton qui est l'unique point d'entrée à la base de données volatile = pas de mise en cache dans chaque thread

Permet l'exécution de requêtes de manière asynchrone Le nombre de threads défini au-dessus indique combien d'accès sont autorisés au maximum à un instant

Permet de limiter l'exécution de ce bloc à un seul thread à un instant donné.
On évite ainsi les accès concurrents et l'éventuelle tentative de création de plusieurs instances (ce qui est contraire au pattern singleton (1 seul point d'entrée à la BD)

il est possible qu'un autre appel à la fonction getDatabase() soit en attente de synchronisation, car le 1er test peut être exécuté simultanément par plusieurs thread (plus rapide).
Dans le cas où un autre thread est en attente mais qu'un autre a déjà créé le singleton, il ne faut pas que celui en attente ne le crée.
on aurait pu placer synchronized directement à la 1ere ligne de la méthode, et ainsi ne faire qu'un seul test mais dans ce cas un seul accès concurrent de récupération de l'instance n'est possible, c'est moins performant

Annexe 4 : Classe de liaison one-to-many

```
public class CategorieWithProduits {
        @Embedded
        public Categorie categorie;
        @Relation(
                parentColumn = "idC", // clé primaire de la table source de la DF
                entityColumn = "idCProduit" // clé étrangère
        )
        public List<Produit> produits;

        public CategorieWithProduits(Categorie categorie, List<Produit> produits) {
                this.categorie = categorie;
                this.produits = produits;
        }

        @Override
        public String toString() {
                return "CategorieWithProduits{" +
                        "categorie=" + categorie +
                        ", produits=" + produits +
                        '}';
        }
}
```

Annexe 5 : Classe repository de la table produit

```
public class ProduitRepository {
    private ProduitDao mProduitDao;
    private LiveData<List<Produit>> mAllProduits;

    public ProduitRepository(Application application) {
        AppDatabase db = AppDatabase.getDatabase(application);
        mProduitDao = db.produitDao();
        mAllProduits = mProduitDao.getAll();
    }

    public LiveData<List<Produit>> getProduits() {
        return mAllProduits;
    }

    public LiveData<Produit> getUnProduit(long id){
        return mProduitDao.getProduitById(id);
    }

    public void insert(Produit produit) {
        AppDatabase.databaseWriteExecutor.execute(() -> {
            mProduitDao.insert(produit);
        });
    }
}
```

Annexe 6 : Exemple de classe viewModel

```
public class MainActivityProduitViewModel extends AndroidViewModel {
        private ProduitRepository mRepository;
        private LiveData<List<Produit>> mAllProduits;
        private LiveData<Produit> mUnProduit;

        public MainActivityProduitViewModel (Application application) {
            super(application);
            mRepository = new ProduitRepository(application);
            mAllProduits = mRepository.getProduits();
        }

        public LiveData<Produit> getUnProduit(long id){
            mUnProduit = mRepository.getUnProduit(id);
            return mUnProduit;
        }

        public LiveData<List<Produit>> getProduits() { return mAllProduits; }
        public void insert(Produit produit) { mRepository.insert(produit); }
}
```

Annexe 7 : Extrait de code de MainActivity

```
public class MainActivity extends AppCompatActivity {

    private MainActivityProduitViewModel mMainActivityProduitViewModel;
    private TextView unTextView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        unTextView = findViewById(R.id.textView);


        mMainActivityProduitViewModel = new
                ViewModelProvider(this).get(MainActivityProduitViewModel.class);


        mMainActivityProduitViewModel.getUnProduit(2).observe(this, produit -> {
            unTextView.setText(produit.nomP);
        });
```