# Android et l’accès aux données

## Objectifs

Utiliser l’ORM Room, surcouche de SQL Lite largement utilisée pour développer des applications avec Android Studio.

Utiliser Java pour développer une interface graphique.

## Fonctionnalités de l’application

L’utilisateur doit pouvoir sélectionner des éléments dans une liste déroulante, ajoutez des éléments et les faire persister dans la base de données.

Pour ajouter des éléments, une nouvelle activité s’ouvre permettant à l’utilisateur de saisir la nouvelle compétence. Lorsqu’il valide il revient à l’écran principal avec la liste de compétences actualisée.

## Contenus

Environnement Android Studio –outil de build Gradle (packaging et dépendances) – DAO – Repository – ORM – SQL Lite - Room – interface - LiveData

## Source

Inspiré des CodeLabs google, meilleure source pour le développement Android.

**Partie 0 : création d’un nouveau projet Android Studio, activité vide**

Etape 1 : ne pas cocher les supports Kotlin et C++

Garder la dernière version de SDK.

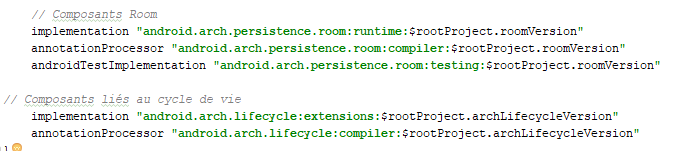
Choisir l’ Activity **Basic**.

Etape 2 : configurer les propriétés gradle

Il y a 2 fichiers gradle dans lequel se fait le paramétrage : <https://developer.android.com/studio/build/>

**Gradle** est l’outil de build et de gestion de dépendances d’Android Studio (également dispo sous Eclipse en alternative à Maven). Un langage dédié est lié à Gradle : groovy.

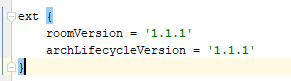
Dans le fichier **build.gradle**, partie Module : app

****

**A copier coller :**

*// Composants Room* implementation **"android.arch.persistence.room:runtime:**$rootProject.roomVersion**"** annotationProcessor **"android.arch.persistence.room:compiler:**$rootProject.roomVersion**"** androidTestImplementation **"android.arch.persistence.room:testing:**$rootProject.roomVersion**"***// Composants liés au cycle de vie* implementation **"android.arch.lifecycle:extensions:**$rootProject.archLifecycleVersion**"** annotationProcessor **"android.arch.lifecycle:compiler:**$rootProject.archLifecycleVersion**"**

Dans le fichier **build.gradle**, partie Module : RoomExemple

****

**A copier coller :**

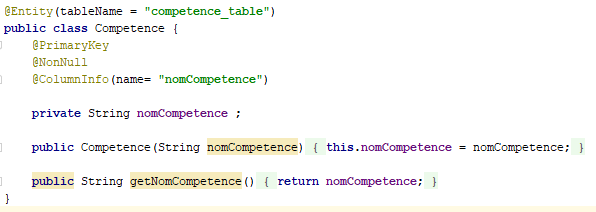
ext {  
 roomVersion = **'1.1.1'** archLifecycleVersion = **'1.1.1'**}

**Partie 1 :** création de l’entité et du DAO

**Etape 3 : création d’une classe métier**

Nous allons créer une classe métier qui va représenter une compétence dans notre application.

Créer une classe Competence et ajoutez le code suivant :



Les @ en Java dont des Annotations.

On peut en trouver pour :

* Générer de la Javadoc
* Donner des instructions de configuration, par exemple dans un framework comme Spring

Ici elles sont utilisées par la librairie ROOM, surcouche ORM de SqlLite, pour donner des informations utiles au moment de la création de la base de données et pour faire le mapping entre les attributs des objets et les champs des tables persistées dans la base.

Avec un ORM, le développeur manipule des objets en langage OO, et la librairie se charge de faire le mapping avec la base de données relationnelle.

**Etape 4 : création d’une interface DAO**

Le pattern DAO est un pattern relativement récent, ne faisant pas partie des Design Patterns originaux définis par Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides (plus connus sous le terme de Gang of Four).

***Le pattern DAO*** répond à la problématique suivante :

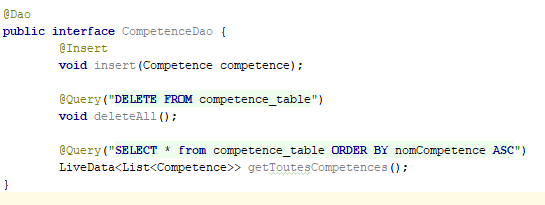
**Problématique** : Comment accéder aux données, dans une application (Java, PHP, C# ..) :

* sans connaître la logique de stockage ? (nature de la base de données, tables, vues …)
* ou de représentation ? (liens entre les tables ?, modèle relationnel ? objet ? NoSQL ?

**Solution apportée** : On définit une classe et implémente des objets qui encapsulent la logique de récupération / mise à jour de données.



Créer une interface nommée ***CompetenceDao***, avec le code suivant :



**Documentation** :

« Marks the class as a Data Access Object.

Data Access Objects are the main classes where you define your database interactions. They can include a variety of query methods.

The class marked with @Dao should either be an interface or an abstract class. At compile time, Room will generate an implementation of this class when it is referenced by a [Database](https://developer.android.com/reference/android/arch/persistence/room/Database.html).

An abstract @Dao class can optionally have a constructor that takes a [Database](https://developer.android.com/reference/android/arch/persistence/room/Database.html) as its only parameter.

It is recommended to have multiple Dao classes in your codebase depending on the tables they touch.

»

*Il n’y aura donc pas de classe qui* vont implémenter cette interface, cela va être fait **automatiquement** par Room.

En savoir plus sur Room avec Dao : <https://developer.android.com/training/data-storage/room/accessing-data>

**Remarque** : nous utilisons la librairie LiveData, qui permet d’observer les données, et éventuellement les rafraichir si besoin : <https://developer.android.com/topic/libraries/architecture/livedata>

Le rafraichissement se fera au moyen d’actions déclenchées par des ***observateurs*** (Observer) des données LiveData

### Documentation : « **Use LiveData with Room**

The [Room](https://developer.android.com/training/data-storage/room/index.html) persistence library supports observable queries, which return [LiveData](https://developer.android.com/reference/android/arch/lifecycle/LiveData.html) objects. Observable queries are written as part of a Database Access Object (DAO).

Room generates all the necessary code to update the LiveData object when a database is updated. The generated code runs the query asynchronously on a background thread when needed. This pattern is useful for keeping the data displayed in a UI in sync with the data stored in a database. You can read more about Room and DAOs in the [Room persistent library guide](https://developer.android.com/topic/libraries/architecture/room.html).”

**Utilisation de LiveData :**

1. Création d’une instance de la classe LiveData pour contenir un certain type de données

* Ce sera effectué par la suite au moyen de notre classe CompetenceViewModel (cf partie 3)

1. Création d’une instance de la classe Observer qui va implémenter la méthode onChanged(), qui définira l’action à entreprendre lorsque les données contenues dans LiveData changent

* Ce sera fait dans l’UI, dans la partie 6, lorsque nous allons définir une nouvelle activité (pour ajouter une compétence) puis la connection aux données

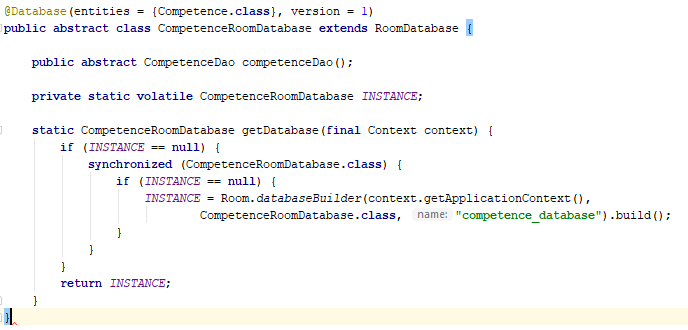
**Partie 2 :** création de la base de données Room et du Repository

**Etape 5 : création d’une base de données Room**

Room est une surcouche à SQLite. Les requêtes sont faites de manière asynchrone dans un thread séparé du thread principal.

Créez une classe abstraite publique **CompetenceRoomDatabase** qui étend la classe RoomDatabase.

Ajoutez le code suivant :



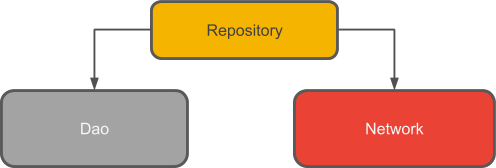
Notes :

* l’annotation @Database indique les entités crées dans notre base de données, en l’occurrence Competence
* la methode abstraite competenceDao() est un getter qui renvoie un Dao pour notre entité.
* INSTANCE est l’instance unique de la base de données Room (évite de saturer la mémoire) : on appelle ça un singleton.
* La méthode getDatabase() renvoie
  + notre INSTANCE si elle n’est pas nulle
  + sinon il crée une instance au moyen de la méthode databaseBuilder() de la classe Room

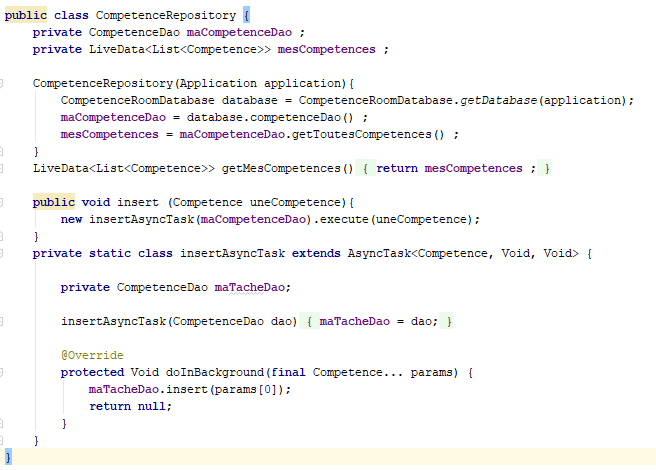
**Etape 6 : création d’un Repository**

Un Repository est une classe permettant l’accès à des sources de données multiples et potentiellement différentes. C’est une meilleure pratique pour séparer le code de l’architecture de l’application.

C’est une API permettant d’accéder aux données pour les autres éléments de l’application (modèle et View – UI).



Créez une classe publique nommée **CompetenceRepository**.

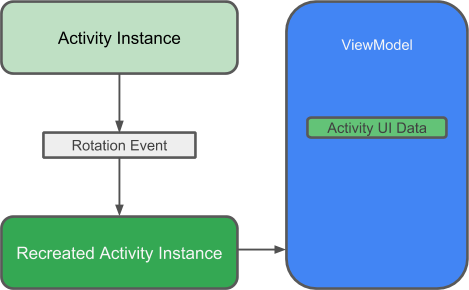


Notes :

* le constructeur
  + initialise la base de données (récupère l’instance ou en crée une),
  + l’interface à cette base de données (la Dao) et
  + une liste de compétences
* LiveData va permettre d’informer les observateurs lorsque les données ont changé
* getMesCompetences permet d’accéder aux compétences (intialisées dans le constructeur via le Dao)
* La méthode insert permet d’ajouter de manière asynchrone une compétence à la BDD au moyen d’une classe permettant de créer une tache asynchrone séparée du thread principal ainsi que de l’interface utilisateur et s’exécutant en arrière-plan

**Partie 3 :** création d’un ViewModel

ViewModel

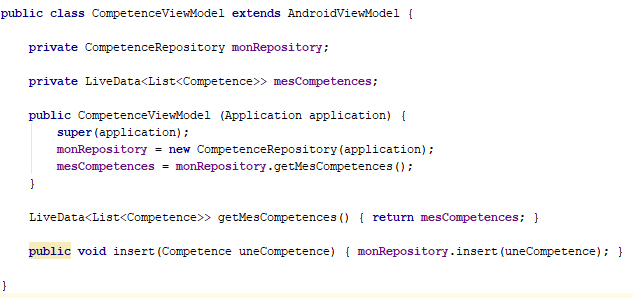


**Documentation** :

« A ViewModel holds your app's UI data in a lifecycle-conscious way that survives configuration changes. Separating your app's UI data from your Activity and Fragment classes lets you better follow the single responsibility principle: Your activities and fragments are responsible for drawing data to the screen, while your ViewModel can take care of holding and processing all the data needed for the UI. »

Le ViewModel se charge donc de fournir les données pour l’interface utilisateur (UI), à savoir les activités et fragment.

Créez une classe qui s’appelle **CompetenceViewModel** qui étend AndroidViewModel, et ajoutez le code suivant :

****

Notes :

* 2 attributs sont déclarés : un repository et une liste de compétences
* Le constructeur récupère le repository lié à l’application et la liste de compétences (getter défini dans le repository)
* getMesCompetences est un accesseur à l’attribut mesCompetences (liste de Competence) de notre classe CompetenceViewModel
* la méthode insert(Competence ..) est une enveloppe autour de la méthode insert() définie dans le Repository