L'ORM Room

1. Objectifs

* Comprendre ce qu’est Room
* Savoir utiliser Room

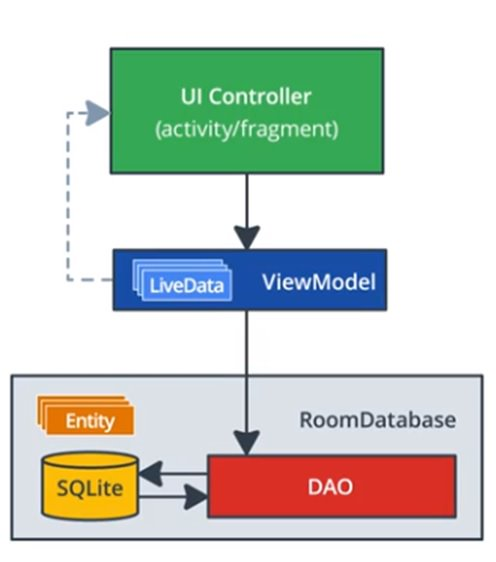
1. Présentation

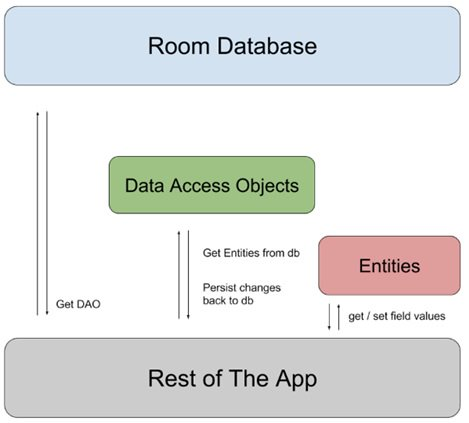
Les applications qui gèrent des quantités de données structurées peuvent grandement bénéficier de la persistance de ces données localement. Le cas d'utilisation le plus courant consiste à mettre en cache des éléments de données pertinents afin que, lorsque l'appareil ne peut pas accéder au réseau, l'utilisateur puisse toujours parcourir ce contenu lorsqu'il est hors ligne.

La bibliothèque de persistance Room est un ORM fournissant une couche d'abstraction sur SQLite pour permettre un accès fluide à la base de données tout en exploitant toute la puissance de SQLite. En particulier, Room offre les avantages suivants :

* Vérification au moment de la compilation des requêtes SQL.
* Des annotations pratiques qui minimisent le code passe-partout répétitif et sujet aux erreurs.

1. Architecture





1. Mise en place

* Ajouter les dépendances
* Créer les Entities
* Créer les DAO
* Créer la Database
* Tester

1. Ajouter les dépendances

Commencez par ajouter le plugin "kotlin-kapt" dans le dossier gradle :

"kotlin-kapt"

Puis ces dépendances :

dependencies {

def room\_version = "2.3.0"

implementation "androidx.room:room-runtime:$room\_version"

kapt "androidx.room:room-compiler:$room\_version"

implementation "androidx.room:room-ktx:$room\_version"

testImplementation "androidx.room:room-testing:$room\_version"

}

1. Entities

Le code suivant définit une entité Personne. Chaque instance de Personne représente un enregistrement de la table personnes dans la base de données de l'application.

import androidx.room.Entity

import androidx.room.PrimaryKey

​

@Entity

data class Personne (@PrimaryKey(autoGenerate = true) var id:Long = 0L, var nom:String, var prenom:String)

1. Réalisation des DAO

Le code suivant définit une DAO appelée PersonneDao. PersonneDao fournit les composants d'accès aux données pour la table Personnes.

import androidx.room.\*

​​

@Dao

interface PersonneDao {

@Insert

fun insert(personne: Personne)

​

@Query("SELECT \* FROM Personne WHERE id = :id")

fun get(id: Long): Personne

​

@Query("SELECT \* FROM Personne ORDER BY nom,prenom")

fun get(): List<Personne?>

​

@Update

fun update(personne: Personne)

​

@Delete

fun delete(personne: Personne)

}

​

1. DataBase

Le code suivant définit une classe AppDatabase qui représente la base de données. AppDatabase définit la configuration de la base de données et sert de point d'accès principal aux données de l'application. La classe représentant la base de données doit satisfaire les conditions suivantes :

* La classe doit être annotée avec l'annotation @Database qui inclut un tableau d'entités qui répertorie toutes les entités de données associées à la base de données.
* La classe doit être une classe abstraite qui hérite de RoomDatabase.
* Pour chaque DAO associée à la base de données, la classe de base de données doit définir une méthode abstraite qui n'a aucun argument et renvoie une instance de la classe DAO.

@Database(entities = arrayOf(Personne::class), version = 1)

abstract class AppDatabase : RoomDatabase() {

abstract fun personneDao(): PersonneDao

companion object {

@Volatile

private var INSTANCE: AppDatabase? = null

fun getInstance(context: Context): AppDatabase {

synchronized(this) {

var instance = INSTANCE

if (instance == null) {

instance = Room.databaseBuilder(

context.applicationContext,

AppDatabase::class.java,

"la\_base\_de\_donnees"

)

.fallbackToDestructiveMigration()

.allowMainThreadQueries()

.addCallback(object : RoomDatabase.Callback() {

override fun onCreate(db: SupportSQLiteDatabase) {

super.onCreate(db)

Log.i("ACOS","Création")

}

})

.build()

INSTANCE = instance

}

return instance

}}}}

1. Test

Le code suivant définit des tests unitaires permettant de tester la DAO PersonneDAO.

@RunWith(AndroidJUnit4::class)

class AppDatabaseTest {

private lateinit var personneDao: PersonneDao

private lateinit var db: AppDatabase

@Before

@Throws(Exception::class)

fun creerBdd()

{

val context = InstrumentationRegistry.getInstrumentation().targetContext

db = Room.inMemoryDatabaseBuilder(context, AppDatabase::class.java).allowMainThreadQueries().build() personneDao = db.personneDao()

}

​

@After

fun deconnexionBdd()

{

db.close()

}

​

@Test

fun insertions()

{

personneDao.insert(Personne(0, "Terieur", "Alex"))

personneDao.insert(Personne(0, "Centrique", "Alex"))

val personne = personneDao.get(1)

assertTrue(personne.nom == "Terieur")

}

​

@Test

fun update()

{

personneDao.insert(Personne(0, "Terieur", "Alex"))

personneDao.insert(Personne(0, "Centrique", "Alex"))

var personne = personneDao.get(1)

personne.nom = "XXX"

personne.prenom = "YYY"

personneDao.update(personne)

​

personne = personneDao.get(1)

assertTrue(personne.nom == "XXX")

assertTrue(personne.prenom == "YYY")

}

}

​

1. ViewModelFactory et injection de dépendance
   1. Présentation

L'injection de dépendance consiste à créer dynamiquement les dépendances entre les différents objets en s'appuyant sur une description (fichier de configuration ou métadonnées) ou de manière programmatique. Ainsi les dépendances entre composants logiciels ne sont plus exprimées dans le code de manière statique mais déterminées dynamiquement à l'exécution.

L’injection de dépendance c’est la séparation de l’utilisation et de la configuration.

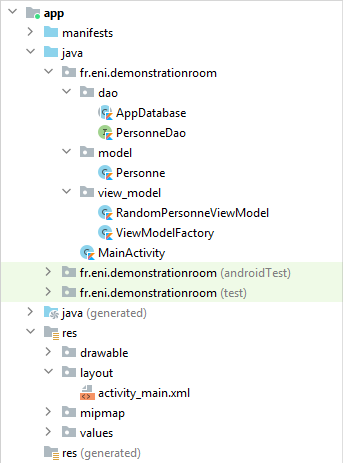
* 1. Mise en place

Créer un ViewModel ayant besoin de ressources

Créer une factory de ViewModel et brancher le ViewModel créé précédement

Brancher la factory au contrôleur

* 1. Démonstration



Le code suivant définit une entité Personne.

import androidx.room.Entity

import androidx.room.PrimaryKey

​

@Entity

data class Personne (@PrimaryKey(autoGenerate = true) var id:Long = 0L, var nom:String, var prenom:String)

Le code suivant définit une DAO pour intéragir avec la table Personnes.

import androidx.room.\*

​​

@Dao

interface PersonneDao {

@Insert

fun insert(personne: Personne)

​

@Query("SELECT \* FROM Personne WHERE id = :id")

fun get(id: Long): Personne

​

@Query("SELECT \* FROM Personne ORDER BY nom,prenom")

fun get(): List<Personne?>

​

@Update

fun update(personne: Personne)

​

@Delete

fun delete(personne: Personne)

}

Le code suivant définit la base de données et le point d'entré pour accéder aux DAO de l'application.

@Database(entities = arrayOf(Personne::class), version = 1)

abstract class AppDatabase : RoomDatabase() {

abstract fun personneDao(): PersonneDao

companion object {

@Volatile

private var INSTANCE: AppDatabase? = null

fun getInstance(context: Context): AppDatabase {

synchronized(this) {

var instance = INSTANCE

if (instance == null) {

instance = Room.databaseBuilder(

context.applicationContext,

AppDatabase::class.java,

"la\_base\_de\_donnees"

)

.fallbackToDestructiveMigration()

.allowMainThreadQueries()

.addCallback(object : RoomDatabase.Callback() {

override fun onCreate(db: SupportSQLiteDatabase) {

super.onCreate(db)

Log.i("ACOS","Création")

}

})

.build()

INSTANCE = instance

}

return instance

}}}}

Le code suivant définit un ViewModel fournissant les données nécessaire à la vue et la logique métier nécessaire à la vue.

import androidx.lifecycle.MutableLiveData

import androidx.lifecycle.ViewModel

​

class RandomPersonneViewModel(val personneDao: PersonneDao) : ViewModel()

{

val personne = MutableLiveData<Personne>()

​

init {

if(personneDao.get().size==0) {

personneDao?.insert(Personne(0, "Cosson", "Anthony"))

personneDao?.insert(Personne(0, "Cosson", "Nicolas"))

personneDao?.insert(Personne(0, "Cosson", "Michel"))

personneDao?.insert(Personne(0, "Cosson", "Barbara"))

personneDao?.insert(Personne(0, "Cosson", "Emma"))

personneDao?.insert(Personne(0, "Cosson", "Ricola"))

}

personne.value = personneDao.get(1)

}

​

fun randomPersonne()

{

val count = personneDao.get().size

val id = (0..count-1).random().toLong()

var hasard = personneDao.get(id)

personne.value = hasard

}

}

Le code suivant définit la factory permettant de fournir une DAO au ViewModel RandomPersonneViewModel.

import android.content.Context

import androidx.lifecycle.ViewModel

import androidx.lifecycle.ViewModelProvider

import java.lang.IllegalArgumentException

​

class ViewModelFactory(val ctx:Context):ViewModelProvider.Factory

{

override fun <T : ViewModel?> create(modelClass: Class<T>): T

{

if(modelClass.isAssignableFrom(RandomPersonneViewModel::class.java))

{

val dao = AppDatabase.getInstance(ctx).personneDao()

var vm = RandomPersonneViewModel(dao)

return vm as T

}

throw IllegalArgumentException("Ce ViewModel n'existe pas")

}

​

}

La code suivant fournit la vue de l'application.

<?xml version="1.0" encoding="utf-8"?>

<layout

xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:app="http://schemas.android.com/apk/res-auto"

xmlns:tools="http://schemas.android.com/tools">

<data>

<variable

name="Model"

type="fr.eni.demonstrationroom.model.Personne" />

</data>

​

<androidx.constraintlayout.widget.ConstraintLayout

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

tools:context=".MainActivity">

​

<TextView

android:id="@+id/tv\_personne"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="@{Model.prenom}"

app:layout\_constraintBottom\_toBottomOf="parent"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

app:layout\_constraintTop\_toTopOf="parent" />

​

<Button

android:id="@+id/btn\_random"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

app:layout\_constraintTop\_toBottomOf="@id/tv\_personne"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

android:text="Random"/>

​

</androidx.constraintlayout.widget.ConstraintLayout>

​

</layout>

Le code suivant fournit le contrôleur de la vue.

import androidx.appcompat.app.AppCompatActivity

import android.os.Bundle

import androidx.databinding.DataBindingUtil

import androidx.lifecycle.Observer

import androidx.lifecycle.ViewModelProvider

​

class MainActivity : AppCompatActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.activity\_main)

​

var vm = ViewModelProvider(this,ViewModelFactory(this)).get(RandomPersonneViewModel::class.java)

var db = DataBindingUtil.setContentView<ActivityMainBinding>(this,R.layout.activity\_main)

​

vm.personne.observe(this, Observer {

db.model = it

})

​

db.btnRandom.setOnClickListener {

vm.randomPersonne()

}

}

}

Le code suivant fournit la liste des dépendances nécessaires au projet.

// Top-level build file where you can add configuration options common to all sub-projects/modules.

buildscript {

repositories {

google()

mavenCentral()

}

dependencies {

classpath "com.android.tools.build:gradle:7.0.3"

classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:1.5.20"

​

// NOTE: Do not place your application dependencies here; they belong

// in the individual module build.gradle files

}

}

​

task clean(type: Delete) {

delete rootProject.buildDir

}

Le code suivant fournit la liste des dépendances et des plugins nécessaires au projet.

plugins {

id 'com.android.application'

id 'kotlin-android'

id "kotlin-kapt"

}

​

android {

compileSdk 31

​

defaultConfig {

applicationId "fr.eni.demonstrationroom"

minSdk 30

targetSdk 31

versionCode 1

versionName "1.0"

​

testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"

}

​

buildFeatures{

dataBinding true

}

​

buildTypes {

release {

minifyEnabled false

proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'

}

}

compileOptions {

sourceCompatibility JavaVersion.VERSION\_1\_8

targetCompatibility JavaVersion.VERSION\_1\_8

}

kotlinOptions {

jvmTarget = '1.8'

}

}

​

dependencies {

implementation 'androidx.core:core-ktx:1.7.0'

implementation 'androidx.appcompat:appcompat:1.3.1'

implementation 'com.google.android.material:material:1.4.0'

implementation 'androidx.constraintlayout:constraintlayout:2.1.1'

testImplementation 'junit:junit:4.+'

androidTestImplementation 'androidx.test.ext:junit:1.1.3'

androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'

}

​

dependencies {

def room\_version = "2.3.0"

implementation "androidx.room:room-runtime:$room\_version"

kapt "androidx.room:room-compiler:$room\_version"

implementation "androidx.room:room-ktx:$room\_version"

testImplementation "androidx.room:room-testing:$room\_version"

}

1. Database Inspector
2. Présentation

Client SQL permettant de visualiser la base de données d'une application.

1. Utilisation

* Exécutez l'application.
* Cliquez sur View -> Tools Window -> App Inspection.
* Dans l'outils App Inspection sélectionnez la BDD à afficher.
* Double cliquez sur la table à afficher.

1. Multi Threading
2. Présentation

Pour empêcher les requêtes de bloquer l'interface utilisateur, Room n'autorise pas l'accès à la base de données sur le thread principal. Cette restriction signifie que vous devez rendre vos requêtes DAO asynchrones. La bibliothèque Room comprend des intégrations avec plusieurs frameworks différents pour fournir une exécution de requête asynchrone.

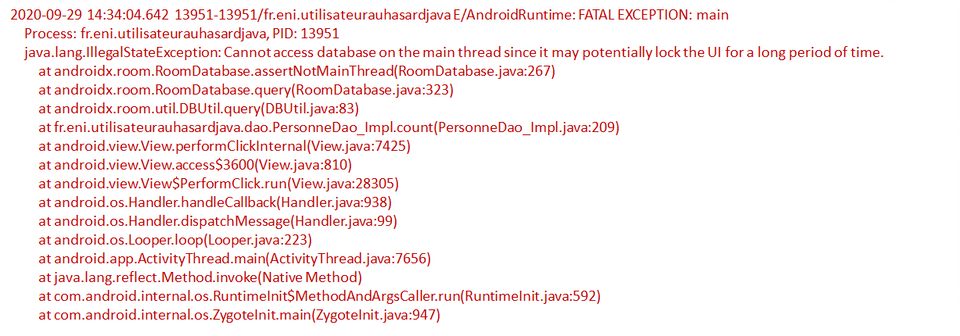
Les requêtes DAO se répartissent en trois catégories :

Requêtes suspendues d'écriture uniques qui insèrent, mettent à jour ou suppriment des données dans la base de données.

Requêtes suspendues de lecture unique qui lisent les données de votre base de données une seule fois et renvoient un résultat avec l'instantané de la base de données à ce moment-là.

Des requêtes de lecture observables qui lisent les données de votre base de données à chaque fois que les tables de base de données sous-jacentes changent et émettent de nouvelles valeurs pour refléter ces changements.

1. Problématique



1. Mise en place

* Ajouter les dépendances
* Définir les fonctions de type "suspend"
* Utiliser les coroutines pour mettre en place le multi threading

1. Ajouter les dépendances

Le code suivant fournit les dépendances nécessaires pour utiliser les coroutines.

dependencies {

implementation 'androidx.lifecycle:lifecycle-extensions:2.2.0'

implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.4.0"

implementation 'androidx.lifecycle:lifecycle-livedata-ktx:2.4.0'

}

​

1. Définir les fonctions "suspend"

Une fonction de suspension (suspend) est une fonction qui peut être démarrée, interrompue et reprise. L'un des points les plus importants à retenir concernant les fonctions de suspension est qu'elles ne peuvent être appelées qu'à partir d'une coroutine ou d'une autre fonction de suspension. On les indique comme suspendues pour s'assurer qu'elles ne sont pas exécutées sur le thread principal.

import androidx.room.\*

​​

@Dao

interface PersonneDao {

@Insert

suspend fun insert(personne: Personne)

​

@Query("SELECT \* FROM Personne WHERE id = :wololo")

suspend fun get(wololo:Long):Personne

​

@Query("SELECT \* FROM Personne Order by nom,prenom")

fun get():LiveData<List<Personne>>

​

@Update

suspend fun update(personne : Personne)

​

@Delete

suspend fun delete(personne : Personne)

​

@Query("SELECT count(\*) FROM Personne")

suspend fun count():Long

}

1. Utiliser les coroutines

Le code suivant montre comment utiliser du code dans une coroutine grâce à viewModelScope.launch {}

class RandomPersonneViewModel(

val personneDao:PersonneDao) : ViewModel()

{

val personne = MutableLiveData<Personne>();

​

fun randomPersonne()

{

viewModelScope.launch {

randomFromDatabase()

}

}

​

private suspend fun randomFromDatabase()

{

val count = personneDao.count()

val numero = (0..count-1).random()

personne.value = personneDao.get(numero.toLong())

}

}

1. Conclusion

Ce chapitre à présenté comment utiliser l'ORM Room dans une base de données. L'utilisation de l'ORM est accompagnée de l'utilisation des coroutines, de LiveData, de ViewModel et du dataBinding.