Android : quelques librairies incontournables

Kotlin

* Retrofit
* Présentation

Retrofit est une librairie qui permet d'accéder simplement à des web services et plus particulièrement dans le cadre d’API JSON. Elle permet de gérer l'envoi des requêtes et la réception des réponses jusqu’à leur conversion sous forme d'objets.

* Procédure

Voici la procédure pour mettre en place Retrofit dans votre application :

* Déclarer une permission pour accéder à Internet.
* Ajouter les dépendances nécessaires pour utiliser Retrofit dans votre projet.
* Créer des objets modèles stockant la réponse du web service.
* Créer une interface de communication qui permettra de créer automatiquement les fonctions d'accès au web service.
* Utiliser Retrofit.
* Démonstration

Pour présenter la mise en place de Retrofit, nous allons exploiter le web service offert par ce site : https://api.chucknorris.io/

Pour présenter les différentes étapes de mise en place de Retrofit, un nouveau projet Android est créé dont l'activité principale se nomme MainActivity et est de type EmptyActivity.

* Permission

Le but de Retrofit est d'accéder à des sites distants, il est donc nécessaire d'avoir une permission pour accéder à Internet.

AndroidManifeste.xml

<?xml version="1.0" encoding="utf-8"?>   
<manifest xmlns:android="http://schemas.android.com/apk/res/android"   
    package="fr.acos.retrofitwithkotlin">   
     <!-- Permission d'accèder à Internet-->   
    <uses-permission android:name="android.permission.INTERNET" />   
   
    <application   
        android:allowBackup="true"   
        android:icon="@mipmap/ic\_launcher"   
        android:label="@string/app\_name"   
        android:roundIcon="@mipmap/ic\_launcher\_round"   
        android:supportsRtl="true"   
        android:theme="@style/AppTheme">   
        <activity android:name=".MainActivity">   
            <intent-filter>   
                <action android:name="android.intent.action.MAIN" />   
   
                <category android:name="android.intent.category.LAUNCHER" />   
            </intent-filter>   
        </activity>   
    </application>   
   
</manifest>

* Dépendances

Voici les dépendances à ajouter pour pouvoir utiliser Retrofit :

dependencies {   
   
    ...   
   
   implementation 'com.squareup.retrofit2:retrofit:2.4.0'   
   
   implementation 'com.squareup.retrofit2:converter-gson:2.4.0'   
   
    ...   
}

* Création d'un objet modèle

Nous avons besoin de stocker la réponse du web service dans un objet. Nous devons donc créer une classe contenant autant de prorpiétés qu’il y a de propriétés dans la réponse JSON. Les propriétés de la classe doivent avoir les mêmes noms que les propriétés de la réponse JSON. Pour notre exemple, la classe Fact va donc être créée, elle représente la réponse du web service chuckNorrisFact. Ainsi, le résultat sera directement renvoyé dans un objet de type Fact. La désérialisation de l’objet est réalisée par GSON.

Exemple de réponse JSON (https://api.chucknorris.io/jokes/random)

{   
 "category":null,   
 "icon\_url":"https:\/\/assets.chucknorris.host\/img\/avatar\/  
chuck-norris.png",   
 "id":"0No5ycQlS-uhySfCxETcnw",   
 "url":"https:\/\/api.chucknorris.io\/jokes\/0No5ycQlS-uhySfCxETcnw",   
 "value":"Every morning, Chuck Norris has a bowl of nails for   
breakfast."}

Fact.kt

package fr.acos.retrofitwithkotlin.bo   
   
/\*\*   
 \* Classe objet permettant de réceptionner les valeurs   
 \* de la réponse JSON du web service   
 \* Les propriétés doivent avoir les mêmes noms que les propriétés   
 \* de la réponse JSON.   
 \*/   
data class Fact   
(   
    val category:List<String>,   
    val icon\_url:String,   
    val id:String,   
    val url:String,   
    val value:String   
)

* Création d'une interface de communication

Cette interface va permettre à Retrofit de créer les fonctions d'accès au web service.

Les annotations permettent de paramétrer la requête.

@GET : définie que la requête est de type GET.

@Path : personnalise le chemin.

@Query : personnalise les paramètres de la requête.

ChuckService.kt

package fr.acos.retrofitwithkotlin.services   
   
import fr.acos.retrofitwithkotlin.bo.Fact   
import retrofit2.Call   
import retrofit2.http.GET   
import retrofit2.http.Path   
import retrofit2.http.Query   
   
/\*\*   
 \* Interface permettant de créer les fonctions d'accès au web service   
 \*/   
interface ChuckService {   
   
    //adresse du web service   
    companion object {   
        val url ="https://api.chucknorris.io/jokes/"   
    }   
   
    /\*\*   
     \* Permet d'envoyer une requête qui retournera un objet de type Fact   
     \* Requête GET de la forme   
     \* https://api.chucknorris.io/jokes/random   
     \*/   
    @GET("random")   
    fun randomFact(): Call<Fact>   
   
    /\*\*   
     \* Permet d'envoyer une requête qui retournera un objet de type Fact   
     \* Requête GET de la forme   
     \* https://api.chucknorris.io/jokes/random/  
     \* {complement}?category={category}   
     \* {complement} et {category} sont paramétrables.   
     \*/   
    @GET("{complement}")   
    fun randomFactByCategory(@Path("complement")complement:  
String, @Query("category")category:String): Call<Fact>   
}

* Utilisation

Activity\_main.xml

<?xml version="1.0" encoding="utf-8"?>   
<android.support.constraint.ConstraintLayout   
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"   
    android:layout\_width="match\_parent"   
    android:layout\_height="match\_parent"   
    tools:context="fr.acos.retrofitwithkotlin.MainActivity">   
   
    <!-- Affiche le fait -->   
    <TextView   
        android:id="@+id/tv\_info"   
        android:layout\_width="wrap\_content"   
        android:layout\_height="wrap\_content"   
        android:text="Hello World!"   
        app:layout\_constraintBottom\_toBottomOf="parent"   
        app:layout\_constraintLeft\_toLeftOf="parent"   
        app:layout\_constraintRight\_toRightOf="parent"   
        app:layout\_constraintTop\_toTopOf="parent" />   
   
</android.support.constraint.ConstraintLayout>

Cette interface graphique contient uniquement une TextView pour afficher le résultat. Les éléments intéressants sont en gras.

MainActivity.kt

package fr.acos.retrofitwithkotlin   
   
import android.support.v7.app.AppCompatActivity   
import android.os.Bundle   
import fr.acos.retrofitwithkotlin.services.ChuckService   
import kotlinx.android.synthetic.main.activity\_main.\*   
import retrofit2.Retrofit   
import retrofit2.converter.gson.GsonConverterFactory   
   
/\*\*   
 \* Activité pour tester l'accès au web service   
 \*/   
class MainActivity : AppCompatActivity() {   
   
    override fun onCreate(savedInstanceState: Bundle?)   
    {   
        super.onCreate(savedInstanceState)   
        setContentView(R.layout.activity\_main)   
   
        //Création d'une implémentation de l'interface ChuckService   
        val retrofit = Retrofit.Builder()   
                .baseUrl(ChuckService.url)   
.addConverterFactory(GsonConverterFactory.create())   
                .build()   
        //Récupération d'une instance de ChuckService   
        val serviceChuck =   
retrofit.create(ChuckService::class.java)   
   
        //Récupération d'un objet de type Call qui permettra de   
        //faire des requêtes sur le web service   
        //Les requêtes ont cette forme :   
        //https://api.chucknorris.io/jokes/random   
        val callServiceRandom = serviceChuck.randomFact()   
   
        //Récupération d'un objet de type Call qui permettra de   
        //faire des requêtes sur le web service   
        //Les requêtes ont cette forme :   
        //https://api.chucknorris.io/jokes/random?category=dev   
        val callServiceCategory =   
serviceChuck.randomFactByCategory("random","dev")   
   
   
        Thread{   
            //Envoi de la requête   
            val reponse =  callServiceCategory.execute()   
   
            //Récupération du résultat de type Fact   
            val fact = reponse.body()   
   
            //Affichage   
            runOnUiThread {   
                tv\_info.text = fact?.value   
            }   
        }.start()   
    }   
}

Les éléments intéressants sont en gras et commentés.

* Anko
* Présentation

Le but de la librairie Anko est de rendre les développements Android plus rapides et plus simples. Elle a été réalisée par l’équipe de Kotlin. Son nom vient de la contraction d’Android et de Kotlin. Il est évidemment possible de s’en passer, mais les différentes parties d’Anko permettent aux développeurs d’améliorer leur code en tous points pour augmenter leur productivité et la qualité de leurs applications.

Les quatre parties de la librairie Anko sont :

* Anko Commons : utilisation des Intents, boîtes de dialogues, toasts, logs et ressources,
* Anko Layouts : création dynamique d’interfaces graphiques,
* Anko SQLite : création et exploitation de bases de données,
* Anko Coroutines : utilisation des tâches asynchrones (encore expérimental et non traité dans le livre).
* Installation

Pour utiliser Anko, une dépendance est nécessaire dans le fichier build.gradle du module de l’application afin de télécharger les librairies requises.

Dans un premier temps, il faut ajouter une variable nommée anko\_version dans le bloc buildscript du fichier build.gradle du projet, cette variable indique la version d’Anko :

buildscript {   
   
    ext.anko\_version = '0.10.2'

Puis, pour télécharger la totalité de la librairie Anko, il faut ajouter la dépendance suivante dans le build.gradle du module :

dependencies {   
   
    implementation "org.jetbrains.anko:anko:$anko\_version"

Pour utiliser uniquement des parties spécifiques d’Anko, il est possible d’utiliser des dépendances permettant de ne télécharger que ce qui est nécessaire.

Dépendances spécifiques

Anko Commons :

dependencies {   
   
    implementation   
"org.jetbrains.anko:anko-commons:$anko\_version"

Anko Layout :

dependencies {   
   
    implementation "org.jetbrains.anko:anko-sdk25:$anko\_version"  
   
    implementation   
"org.jetbrains.anko:anko-appcompat-v7:$anko\_version"

Anko SQLite :

dependencies {   
   
    implementation "org.jetbrains.anko:anko-sqlite:$anko\_version"

Anko Coroutines :

dependencies {   
   
    implementation   
"org.jetbrains.anko:anko-sdk25-coroutines:$anko\_version"   
   
    implementation   
"org.jetbrains.anko:anko-appcompat-v7-coroutines:$anko\_version"

* Anko Commons
* Présentation

Anko Commons permet de simplifier l’utilisation d’intents, de boîtes de dialogues, de logs et de toasts. L’écriture du code permettant l’utilisation de ces outils est verbeuse. En effet, de nombreux paramètres ont des valeurs constantes, les structures utilisées sont souvent identiques et les variations sont rares. De ce fait, la librairie Anko accélère l’utilisation de ces outils. Elle met la partie variable et intéressante en exergue tout en occultant les parties verbeuses et rébarbatives.

* Intent

Démarrer une nouvelle activité

Sans Anko, l’utilisation des intents requiert a minima deux lignes, ainsi que de multiples répétitions de paramètres. Lorsqu’un intent contient des extras, le nombre de lignes augmente en conséquence. Anko améliore ce mécanisme en se concentrant sur les paramètres essentiels et en simplifiant l’utilisation des extras. En une ligne il est possible de définir :

* un intent
* plusieurs extras

La fonction startActivity permet de passer sur une autre activité.

Syntaxe

fun <reified T: Activity> startActivity(vararg params: Pair<String, Any>)

* Le paramètre de type T attend le nom de l’activité vers laquelle on souhaite se diriger.
* Le paramètre params attend les potentiels extras.

Exemple

startActivity<AddActivity>("info1" to 15, "info2" to "AA")

Dans cet exemple, la fonction startActivity permet de se diriger vers l'activité AddActivity tout en fournissant deux extras. La clé du premier extra est "info1" et sa valeur est 15. La clé du second extra est "info2" et sa valeur est "AA".

Démarrer des services spécifiques

Anko améliore l’utilisation des intents pour l’accès à certains services du téléphone.

Passer un appel téléphonique

La fonction makeCall permet de passer un appel depuis une application.

Syntaxe

makeCall(number: String): Boolean

Cette fonction attend en paramètre le numéro de téléphone à appeler et retourne true si le passage sur l’interface d’appel a été réalisé avec succès ; dans le cas contraire la valeur retournée est false.

Attention, une application doit disposer du droit suivant pour avoir la possibilité de passer un appel :

            android.permission.CALL\_PHONE

Exemple

makeCall("0651865318")

Envoyer un SMS

La fonction sendSMS permet d'accéder à l’interface d’envoi de SMS en préremplissant le numéro du destinataire ainsi que le message à envoyer.

Syntaxe

sendSMS(number: String, text: String = ""): Boolean

Cette fonction attend en premier paramètre le numéro de téléphone du destinataire et en second paramètre le texte du SMS. Le second paramètre est optionnel. Cette fonction retourne true si l’accès au service a été réalisé avec succès, dans le cas contraire la valeur retournée est false.

Exemple

sendSMS("0651865317","Bonjour")

La taille maximale d'un SMS est de 160 caractères.

Accéder à un site internet

La fonction browse permet d'accéder à un site internet via un navigateur.

Syntaxe

browse(url: String): Boolean

Cette fonction attend en paramètre l’adresse du site internet. Une fois exécutée, elle ouvre votre navigateur web avec le site internet choisi. Cette fonction retourne true si l’ouverture a été réalisée avec succès, dans le cas contraire la valeur retournée est false.

Exemple

browse("https://www.kotlinlang.org")

Envoyer un email

La fonction email permet d'accéder à l’interface d’envoi d'email et préremplit l'adresse email du destinataire, ainsi que le sujet et le message à envoyer.

Syntaxe

email(email: String, subject: String = "", text: String = ""):   
Boolean

Cette fonction attend en premier paramètre l’adresse email du destinataire, en second paramètre le sujet de l’email et en troisième paramètre le texte de l’email. Le second et le troisième paramètres sont optionnels. Cette fonction retourne true si l’accès au service a été réalisé avec succès, dans le cas contraire la valeur retournée est false.

Exemple

email("antho.cosson@gmail.com","Kotlin","Bonjour, je vous inf…")

* Les boîtes de dialogues

Anko propose de simplifier l’utilisation de plusieurs boîtes de dialogue telles que :

* les alertes
* les sélecteurs
* les barres de progression

Les alertes

La fonction alert permet de créer des alertes. Une alerte se comporte comme une fenêtre pop-up sur un navigateur. Elle peut contenir un titre, un texte, un formulaire et des boutons. Lors de son affichage, l’alerte prend le focus. Une fois l’alerte affichée, l’utilisateur peut cliquer en dehors de l’alerte afin qu’elle disparaisse, ou bien interagir avec les boutons de l’alerte si la possibilité est donnée.

Syntaxe

alert(   
        message: String,   
        title: String? = null)

Cette fonction attend en premier paramètre le message à afficher et en second paramètre le titre de la fenêtre. Le second paramètre est optionnel.

Exemple

alert("Titre","Ceci est le message").show()

Ne pas oublier d'utiliser la fonction show pour afficher l'alerte.

Les sélecteurs

La fonction selector permet de créer des sélecteurs. Un sélecteur se comporte comme une fenêtre pop-up sur un navigateur. Il contient un titre et une liste de choix. Lors de son affichage, le sélecteur prend le focus. Une fois affiché, l’utilisateur peut cliquer en dehors du sélecteur afin qu’il disparaisse, ou bien sélectionner un choix.

Syntaxe

selector(   
        title: CharSequence? = null,   
        items: List<CharSequence>,   
        onClick: (DialogInterface, Int) -> Unit   
)

Cette fonction attend en premier paramètre le titre de la fenêtre à afficher, en second paramètre la liste des items à afficher et en troisième paramètre la fonction à exécuter lorsque l'utilisateur fait un choix.

Exemple

   val couleurs = listOf("Bleu", "Blanc", "Rouge", "Vert")   
   
    selector("Quelle est votre couleur préférée ?", couleurs,    
   
    {       
   
        dialogInterface, i ->   
            toast("Votre couleur : ${couleurs[i]}")   
   
    })

Les barres de progression

La fonction progressDialog permet de créer une barre de progression avec un état visible. Les barres de progression permettent de faire patienter les utilisateurs. Anko propose deux barres de progression. La première permet de connaître l’état de la progression et la seconde affiche un état indéterminé.

Syntaxe

progressDialog(   
        message: String? = null,   
        title: String? = null    
) : ProgressDialog

Cette fonction attend en premier paramètre le message de la barre de progression et en second paramètre le titre de la barre de progression. La fonction retourne un objet de type ProgressDialog qui représente notre barre de progression.

Exemple

val dialog = progressDialog(message = "Veuillez patienter",   
title = "Chargement en cours")

Pour créer une barre de progression avec un état indéterminé, Anko propose la fonction indeterminateProgressDialog.

Syntaxe

indeterminateProgressDialog(   
        message: String? = null,   
        title: String? = null   
) : ProgressDialog

Exemple

val dialog = indeterminateProgressDialog(message = "Veuillez   
patienter", title = "Chargement en cours")

* Les logs

Sans Anko, l’utilisation des fonctions de log impose toujours au moins deux paramètres, le tag et le message. Cela est gênant, car il faut systématiquement définir une variable pour le tag ou bien saisir le tag en dur dans la fonction, or, cela n’est pas une bonne pratique. Anko améliore ce mécanisme.

Pour utiliser les logs avec Anko, une classe doit implémenter l’interface AnkoLogger. Cette implémentation offre ensuite la possibilité d’utiliser les fonctions verbose, debug, info, warn, error et wtf. Le tag prend automatiquement comme valeur le nom de la classe dans laquelle se situe la fonction de log, ainsi le développeur n’a plus qu’à se soucier du message qu’il souhaite émettre.

La personnalisation du tag est toujours possible en surchargeant la propriété loggerTag ou en créant un objet de type AnkoLogger.

Le tag ne doit jamais faire plus de 23 caractères.

Syntaxe

info(message: String)

La fonction info attend en paramètres les informations à loguer.

Exemple d’utilisation des logs avec Anko

class AddActivity : AppCompatActivity(),AnkoLogger {   
   
    private fun testerLesFonctionsDeLog(info : String)   
    {   
        wtf("WTF : " + info)   
        verbose("VERBOSE : " + info)   
        debug("DEBUG : " + info)   
        info("INFO : " + info)   
        warn("INFO : " + info)   
        error ("ERROR : " + info)   
    }

Exemple de personnalisation du tag en surchargeant la propriété loggerTag

class AddActivity : AppCompatActivity(),AnkoLogger {   
   
    override val loggerTag: String   
        get() = "PERSO"

Tous les logs provenant de cette classe auront comme tag la valeur "PERSO".

Exemple de personnalisation du tag   
en créant un objet de type AnkoLogger

class AddActivity : AppCompatActivity(),AnkoLogger {   
   
    private val logAvecTagPerso = AnkoLogger("PERSO")   
   
    private fun testerLesObjetsLogger(info : String)   
    {   
        logAvecTagPerso.info("INFO" + info)   
    }

Tous les logs diffusés via l'objet logAvecTagPerso auront comme tag la valeur "PERSO".

* Les toasts

La fonction toast permet d'envoyer un toast avec une durée courte :

Syntaxe

toast(message: CharSequence)

La fonction longToast permet d'envoyer un toast avec une durée longue.

Syntaxe

longToast(message: CharSequence)

Ces deux fonctions attendent en paramètre le message à afficher.

Exemple

toast("Bonjour !")    
   
longToast("Hello !")

* Anko SQLite
* Présentation

L’utilisation de SQLite pour les applications Android est incontournable. L’écriture des classes de manipulation de la base de données est fastidieuse et les structures de ces classes sont souvent identiques les unes des autres. Anko propose de nombreuses fonctions permettant de simplifier ces mécanismes, tout d’abord parce que ces fonctions sont beaucoup plus explicites et plus simples d’utilisation. De plus, les fonctions proposées par Anko incitent les développeurs à créer un programme respectant les bonnes pratiques.

Les fonctions proposées par Anko simplifient les points suivants :

* accès à la base de données
* définition de la structure de la base de données
* insertion de données
* extraction de données
* mise à jour de données
* suppression de données
* Accès et définition de la base de données

Sans Anko, la classe de création de la base de données hérite de SQLiteOpenHelper. Anko optimise et simplifie son utilisation : optimisée au niveau de l’utilisation de la base de données par plusieurs threads et simplifiée grâce à son système d’auto-fermeture des connexions ainsi qu'à ses paramètres de type Pair. Pour profiter de ces améliorations, on doit implémenter la classe Anko nommée ManagedSQLiteOpenHelper.

Exemple d’utilisation

class BddHelper(ctx: Context) : ManagedSQLiteOpenHelper(ctx,   
"wineBdd",null,2)   
{   
    companion object   
    {   
        private var instance:BddHelper?=null   
   
        @Synchronized   
        fun getInstance(ctx:Context):BddHelper   
        {   
            if(instance==null)   
            {   
                instance = BddHelper(ctx)   
            }   
            return instance as BddHelper   
        }   
    }   
   
    override fun onCreate(db: SQLiteDatabase?) {   
        db?.createTable(FriendTableContract.NAME,   
                false,   
                FriendTableContract.COL\_ID to INTEGER + PRIMARY\_KEY,   
                FriendTableContract.COL\_NAME to TEXT,   
                FriendTableContract.COL\_WINE to TEXT)   
    }   
   
    override fun onUpgrade(db: SQLiteDatabase?, p1: Int, p2: Int) {   
        db?.dropTable(FriendTableContract.NAME,true)   
        onCreate(db)   
    }   
}   
   
val Context.database : SQLiteDatabase   
    get() = BddHelper.getInstance(applicationContext)

Explication de l’exemple

L’utilisation du design pattern singleton avec l’annotation @Synchronised permet de garantir l’unicité de l’instance même en cas de multithreading.

La redéfinition de la fonction onCreate va permettre de créer la base de données. Dans la fonction onCreate, la fonction createTable est définie pour créer une table. Elle attend trois paramètres :

* Le nom de la table à créer.
* Un booléen permettant de contrôler si la table existe déjà ou non.
* Un paramètre variadique. Cela signifie que la fonction attend autant de paramètres qu’il y a de colonnes dans la table. Pour définir une colonne, il faut indiquer une paire d’informations, le nom de la colonne et les caractéristiques de la colonne. Une colonne peut avoir plusieurs caractéristiques.

Voici la liste des caractéristiques possibles :

* NULL
* INTEGER
* REAL
* TEXT
* BLOB
* FOREIGN\_KEY
* PRIMARY\_KEY
* NOT\_NULL,
* AUTOINCREMENT
* UNIQUE
* DEFAULT

La redéfinition de la seconde fonction onUpgrade permet de définir le comportement d’une potentielle mise à jour de la base de données. Elle est exécutée lorsque le numéro de version de la base de données change. Dans l’exemple, il a été choisi de supprimer l’unique table de la base de données, puis de la recréer.

Pour finir, une propriété d’extension nommée database est créée sur la classe Context, cela permet d’accéder facilement à l’instance représentant la base de données.

* Insertion de données

Anko simplifie l’insertion en base de données. Il est dorénavant inutile d’utiliser un objet de type contentValue. Dans la fonction insert, il suffit de passer en premier paramètre le nom de la table dans laquelle il y a une insertion à faire, puis de fournir autant de paires clé/valeur qu’il y a de colonnes dans l’enregistrement, la clé correspond au nom d’une colonne et la valeur à la valeur à insérer. Pour finir, l’instruction « use » permet de fermer automatiquement la connexion à la base de données.

Syntaxe

insert(tableName: String, vararg values: Pair<String, Any?>): Long

La fonction insert retourne un Long qui est égal à -1 en cas d’échec, sinon à l’identifiant du nouvel enregistrement.

Exemple

object FriendDao   
{   
    private fun insert(context:Context, item: Friend):Long  {   
   
        return context.database.use {   
            insert(FriendTableContract.NAME,   
                    FriendTableContract.COL\_ID to item.id,   
                    FriendTableContract.COL\_NAME to item.name,   
                    FriendTableContract.COL\_WINE to item.wine)   
        }   
    }

* Mise à jour de données

Anko simplifie la mise à jour en base de données. Il est dorénavant inutile d’utiliser un objet de type contentValue. Dans la fonction update, il suffit de passer en premier paramètre le nom de la table dans laquelle il y a une mise à jour à faire, puis de fournir autant de paires clé/valeur qu’il y a de colonnes à mettre à jour, la clé correspond au nom d’une colonne et la valeur à la nouvelle valeur de la colonne. On utilise ensuite la fonction whereSimple pour définir la condition de mise à jour. Pour finir, l’instruction « use » permet de fermer automatiquement la connexion à la base de données.

Syntaxe

update(tableName: String, vararg values: Pair<String, Any?>):   
UpdateQueryBuilder

La fonction update retourne un objet de type UpdateQueryBuilder, cet objet permet de générer la requête.

Exemple

object FriendDao   
{   
    private fun update(context:Context, item: Friend) {   
   
       context.myDatabase.use {   
            update(FriendTableContract.NAME,   
                    FriendTableContract.COL\_NAME to item.name,   
                    FriendTableContract.COL\_WINE to item.wine   
            ).whereSimple("${FriendTableContract.COL\_ID} = ?",   
item.id.toString())   
        }   
    }

* Suppression de données

Pour supprimer un enregistrement avec Anko, il suffit d’utiliser la fonction delete et de passer en premier paramètre le nom de la table dans laquelle il y a suppression, puis de fournir une condition particulière en second paramètre. Et en troisième paramètre variadique, une ou des paires d’informations pour compléter la condition de suppression. Pour finir, l’instruction « use » permet de fermer automatiquement la connexion à la base de données.

Syntaxe

delete(tableName: String, whereClause: String = "", vararg args:   
Pair<String, Any>): Int

La fonction delete retourne le nombre de lignes supprimées.

Exemple

object FriendDao   
{   
    private fun delete(context:Context, item: Friend) {   
   
        context.myDatabase.use {   
            delete(FriendTableContract.NAME,"${FriendTableContract.COL\_ID} =   
{id}",FriendTableContract.COL\_ID to item.id.toString())   
        }   
    }

* Extraction de données

Anko simplifie l’extraction en base de données. Il suffit d’utiliser la fonction select en indiquant le nom de la table sur laquelle l’extraction est souhaitée.

Syntaxe

select(tableName: String): SelectQueryBuilder

La fonction select retourne un objet de type SelectQueryBuilder, cet objet permet de générer la requête.

Exemple permettant de récupérer la totalité   
des enregistrements d’une table

object FriendDao   
{   
    fun get(context:Context) : List<Friend>   
    {   
        return   
context.database.select(FriendTableContract.NAME).parseList(parser)   
    } 

Exemple permettant de récupérer un enregistrement spécifique

object FriendDao   
{   
    fun get(context:Context,id:Int) : Friend   
    {   
        return   
context.database.select(FriendTableContract.NAME)  
.whereArgs("${FriendTableContract.COL\_ID} = {id}",  
FriendTableContract.COL\_ID to id).parseSingle(parser)   
    } 

Il existe de nombreuses méthodes pour ajouter des instructions à la requête :

* column(String) : permet de définir la projection de la requête.
* distinct(Boolean) : permet d’utiliser l’instruction distinct dans la requête.
* whereArgs(String) : permet d’utiliser l’instruction SQL WHERE avec une condition statique.
* whereArgs(String, args) : permet d’utiliser l’instruction SQL WHERE avec des arguments de paires clé/valeur.
* orderBy(String, [ASC/DESC]) : permet d’utiliser l’instruction SQL ORDER BY.
* groupBy(String) : permet d’utiliser l’instruction SQL GROUP BY.
* limit(Int) : permet de limiter le nombre de résultats.
* having(String,args) : permet d’utiliser l’instruction SQL HAVING avec des arguments de paires clé/valeur.
* Anko Layouts
* Présentation

Anko Layouts permet de définir simplement et dynamiquement une IHM par programmation, cela permet de ne plus passer par le XML.

Voici un exemple simple :

MainActivity.kt

package fr.acos.ankolayoutswithkotlin   
   
import android.graphics.Color   
import android.support.v7.app.AppCompatActivity   
import android.os.Bundle   
import org.jetbrains.anko.editText   
import org.jetbrains.anko.\*   
import org.jetbrains.anko.sdk25.coroutines.onClick   
   
class MainActivity : AppCompatActivity() {   
   
    override fun onCreate(savedInstanceState: Bundle?) {   
        super.onCreate(savedInstanceState)   
   
         UI(true) {   
            verticalLayout {   
                val etNom = editText()   
                {   
                    hint = "Nom"   
                }   
                val etPrenom = editText()   
                {   
                    hint = "Prénom"   
                }   
   
                val btnApply = button("Valider") {   
                    onClick {   
                        toast("Saisie : ${etNom.text} ${etPrenom.text}")   
                    }   
                    textColor = Color.WHITE   
                    backgroundResource = R.color.colorPrimary   
                }   
            }   
        }   
    }   
}

Dans cet exemple, les instructions créent deux champs de saisie et un bouton à l'intérieur d'un LinearLayout et attachent un OnClickListener à ce bouton permettant ainsi d’afficher un toast lors du clic. On définit ensuite les couleurs du bouton. *Le fichier XML définissant l'IHM n'est plus lié à l'activité*.

Rendu



* Définition rapide des propriétés essentielles

Il existe une fonction pour chaque élément graphique : editText(), button(), textView()... Ces fonctions attendent en paramètres les propriétés les plus souvent utilisées (intitulé du bouton, texte de la textView…).

Exemple

verticalLayout {   
   
    textView("Hello world!!!")   
   
    textView(R.string.hello\_world)   
   
}

Il est possible de définir directement l'identifiant d'une ressource.

* Définition rapide du thème des éléments graphiques

Les fonctions préfixées du mot-clé themed permettent de définir un élément graphique et son thème associé.

Exemple

verticalLayout {   
    themedEditText(R.style.MonTheme)   
    themedButton("Ok", theme = R.style.MonTheme)   
}

Il est possible d'associer un thème aux éléments graphiques.

* Modification des propriétés des éléments graphiques

Il est possible de modifier les propriétés d'un élément de façon plus spécifique.

Exemple

val etNom = editText("")   
{   
    backgroundColor=R.color.colorPrimary   
    textColor = Color.WHITE   
}

Toutes les propriétés d'un élément peuvent être modifiées entre les accolades.

* Modification des propriétés layout des éléments graphiques

La fonction lparams() peut être utilisée pour définir les paramètres liés au layout.

Exemple

verticalLayout {   
    val etPrenom = editText("Prénom :").lparams(width =   
wrapContent,height = wrapContent) {   
        topMargin = dip(30)   
    }   
   
}

La largeur et la hauteur ainsi que des marges ont été définies sur l'editText.

Le code ci-dessus équivaut à :

<EditText    
   
    android:layout\_width="wrap\_content"   
   
    android:layout\_height="wrap\_content"   
   
    android:layout\_marginTop="30dp"   
   
/>

* UI Wrapper

Afin de prioriser l'affichage de l'interface graphique, il faut englober le code de définition d'IHM comme ceci :

 UI(true) {   
   
    //Définition de l'IHM   
   
}

* Conclusion

Ce chapitre a permis de découvrir la librairie Anko. Cette librairie a été construite avec la même philosophie que celle qui a bâti le langage Kotlin. Il est évidemment possible de s’en passer, mais les quatre parties d’Anko permettent aux développeurs d’améliorer leur code en tous points pour augmenter leur productivité et la qualité de leurs applications.

* Room
* Présentation

Room est l'ORM officiel pour mettre en place une persistance de données dans une application Android. Cette librairie brille par son efficacité, sa simplicité et son système de mise en cache.

Voici les trois composantes principales de Room :

* L'objet de type Database qui représente la base de données de l'application.
* Les objets de type Entity qui représentent les enregistrements des tables de la base de données de l'application. En général, il existe une classe de type Entity par table.
* Les objets de type DAO qui contiennent les fonctions permettant de manipuler les tables. En général, il existe une classe de type DAO par table.
* Procédure

Voici la procédure pour mettre en place Room dans votre application :

* Ajouter les dépendances nécessaires pour utiliser Room dans votre projet.
* Définir les classes de type Entity. Concrètement, ce sont les classes du modèle avec des annotations.
* Définir les DAO. Concrètement, ce sont des interfaces contenant des signatures de fonctions qui manipuleront la base de données de l'application. Le corps est généré par Room. Dans les DAO se trouvent des signatures de fonctions de sélection, insertion, mise à jour et suppression de données stockées en base de données.
* Définir une classe de type Database. Cette classe permet de fournir des objets représentant la base de données de l'application.

Un objet de type Database permet d'accéder aux DAO. Les DAO permettent de fournir les objets de type Entity qui représentent les données de l'application.

* Création de la base de données

C'est Room qui est responsable de la création de la base de données. Room s'appuie sur les annotations définies dans les classes de type Entity pour créer la base de données et les tables.

* Dépendances

Voici les dépendances à ajouter pour pouvoir utiliser Room :

dependencies {   
   
...   
    implementation "android.arch.persistence.room:runtime:1.1.0"   
    annotationProcessor "android.arch.persistence.room:compiler:1.1.0"   
...   
}

* Création d'une classe de type Entity

Une classe de type Entity représente une table. Un objet d'une classe Entity représente un enregistrement.

Exemple d'une classe de type Entity

package fr.acos.entities   
   
import android.arch.persistence.room.ColumnInfo   
import android.arch.persistence.room.Entity   
import android.arch.persistence.room.PrimaryKey   
   
@Entity(tableName = "personnes")   
data class Personne(   
        @ColumnInfo(name="id") @PrimaryKey(autoGenerate = true) var   
id: Long = 0,   
        @ColumnInfo(name="nom") var nom: String,   
        @ColumnInfo(name="prenom") var prenom: String   
)

Cette classe va permettre à Room de créer une table personnes. Cette table contiendra trois colonnes :

* Une colonne nommée id de type INTEGER, cette colonne représente la clé primaire auto-incrémentée.
* Une colonne nommée nom de type TEXT.
* Une colonne nommée prenom de type TEXT.

Les enregistrements de cette table seront représentés par des objets de type Personne.

Liste d'annotations utilisées pour les entités :

|  |  |
| --- | --- |
| Annotation | Description |
| @ColumnInfo | Permet de personnaliser la colonne associée à la propriété. |
| @Embedded | Annotation utilisée sur un objet Entity ; cela permet de définir autant de colonne que l’objet Entity a de propriétés. |
| @Entity | Marque une classe en tant qu’entité. |
| @ForeignKey | Déclare une clé étrangère sur une autre entité. |
| @Ignore | Ignore la propriété. |
| @Index | Déclare un index. |
| @PrimaryKey | Marque une propriété comme clé primaire. |

* Création d'une interface de type DAO

Pour créer une DAO (Data Access Object) offrant des fonctions manipulant la bdd, il suffit de définir une interface avec des annotations au-dessus des signatures de fonctions, comme ci-dessous :

package fr.acos.dao   
   
import android.arch.persistence.room.\*   
import fr.acos.entities.Personne   
   
@Dao   
interface PersonneDao {   
   
    @Query("SELECT \* FROM personnes")   
    fun get(): List<Personne>   
   
    @Query("SELECT \* FROM personnes WHERE id = :id")   
    fun get(id: Long): Personne   
   
    @Insert   
    fun insertAll(vararg listCategories: Personne)   
   
    @Update   
    fun updatePersonne(task: Personne)   
   
    @Delete   
    fun deletePersonne(task: Personne)   
}

Liste d'annotations utilisées pour les DAO :

|  |  |
| --- | --- |
| Annotation | Description |
| @DAO | Marque une classe en tant que DAO. |
| @Delete | Marque une fonction dans une DAO comme fonction de suppression. |
| @Insert | Marque une fonction dans une DAO comme fonction d’insertion. |
| @Ignore | Ignore une fonction. |
| @Query | Marque une fonction dans une DAO comme fonction de sélection. |
| @Update | Marque une fonction dans une DAO comme fonction de mise à jour. |

* Création d'une classe de type Database

Pour créer une classe de type Database fournissant l'accès aux Dao, il faut suivre l'exemple ci-dessous :

package fr.acos.dao   
   
import android.arch.persistence.room.Database   
import android.arch.persistence.room.RoomDatabase   
import fr.acos.entities.Personne   
   
@Database(entities = [(Personne::class)], version = 1)   
abstract class MaBaseDeDonnees : RoomDatabase()   
{   
    abstract fun personneDao(): PersonneDao   
}

Liste d'annotations utilisées pour les Database :

|  |  |
| --- | --- |
| Annotation | Description |
| @Database | Marque une classe comme représentante de la base de données. |

* Récupération d'une instance de la base de données

Le code ci-dessous présente comment récupérer un objet représentant la base de données :

AppDatabase db = Room.databaseBuilder(getApplicationContext(),   
        AppDatabase.class, "bddPersonnes").build();

* Utilisation
* Insertion

AsyncTask.execute({   
    val db = Room.databaseBuilder(applicationContext,   
            MaBaseDeDonnees::class.java, "database-name").build()   
   
    val dao = db.personneDao()   
   
    val p1 = Personne(0,"Hassoun","Kiril")   
    val p2 = Personne(0,"GuenGuen","Maxime")   
   
    dao.insertAll(p1,p2)   
})

Dans cet exemple, deux enregistrements sont insérés dans la table personnes.

* Lecture

AsyncTask.execute({   
    val db = Room.databaseBuilder(applicationContext,   
            MaBaseDeDonnees::class.java, "database-name").build()   
   
    val dao = db.personneDao()   
    val liste = dao.get()   
   
    liste.forEach { item -> Log.i("XXX",item.toString())}   
})

Dans cet exemple, tous les enregistrements de la table personnes sont affichés.

AsyncTask.execute({   
    val db = Room.databaseBuilder(applicationContext,   
            MaBaseDeDonnees::class.java, "database-name").build()   
   
    val dao = db.personneDao()   
   
    val pX = Personne(1,"Kikiriri","Kirille")   
   
    val recuperation = dao.get(2L)   
   
    Log.i("XXX",recuperation.toString())   
})

Dans cet exemple, un enregistrement ayant un id égal à 2, de la table personnes, est affiché.

* Modification

AsyncTask.execute({   
    val db = Room.databaseBuilder(applicationContext,   
            MaBaseDeDonnees::class.java, "database-name").build()   
   
    val dao = db.personneDao()   
   
    val pX = Personne(1,"Kikiriri","Kirille")   
   
    dao.updatePersonne(pX)   
})

Dans cet exemple, un enregistrement est mis à jour dans la table personnes.

* Suppression

AsyncTask.execute({   
    val db = Room.databaseBuilder(applicationContext,   
            MaBaseDeDonnees::class.java, "database-name").build()   
   
    val dao = db.personneDao()   
   
    val pX = Personne(1,"Kikiriri","Kirille")   
   
    dao.deletePersonne(pX)   
})

Dans cet exemple, un enregistrement est supprimé de la table personnes.

* Conclusion

Room est simple à mettre en place et à maintenir et c'est pourquoi il est vivement conseillé d'utiliser cette solution pour la persistance en base de données.