Communiquer avec une API REST

1. Objectif

•Communiquer avec une API REST

•Exploiter les données d’une API REST

1. Présentation

Les API REST constituent un moyen souple et léger pour connecter des applications à des sources de données distantes.

Une API, ou interface de programme d'application, est un ensemble de règles qui définissent comment les applications peuvent communiquer entre eux. Une API REST est une API conforme aux principes de conception du style architectural REST ou Representational State Transfer. C'est pour cette raison que les API REST sont parfois appelées API RESTful .

Défini pour la première fois en 2000 par l'informaticien Roy Fielding dans sa thèse de doctorat, REST offre un niveau relativement élevé de flexibilité et de liberté pour les développeuses et développeurs. Cette flexibilité n'est qu'une des raisons pour lesquelles les API REST sont devenues une méthode courante pour connecter des applications.

Toutes les demandes d'API pour une même ressource doivent avoir la même apparence, quelle que soit l'origine de la demande. L'API REST doit garantir que le même élément de données, tel que le nom ou l'adresse électronique d'un utilisateur, n'appartient qu'à un seul identificateur URI (Uniform Resource Identifier). Les ressources ne doivent pas être trop volumineuses, mais elles doivent contenir toutes les informations dont le client peut avoir besoin.

Dans la conception des API REST, les applications client et serveur doivent être totalement indépendantes les unes des autres. La seule information que l'application cliente doit connaître est l'URI de la ressource demandée ; elle ne peut interagir avec l'application serveur en aucune autre manière. De même, une application serveur ne doit pas modifier l'application client autrement qu'en lui transmettant les données demandées via HTTP.

Les API REST sont sans état, ce qui signifie que chaque demande doit inclure toutes les informations nécessaires à son traitement. En d'autres termes, les API REST ne nécessitent aucune session côté serveur. Les applications serveur ne sont pas autorisées à stocker des données liées à une demande du client.

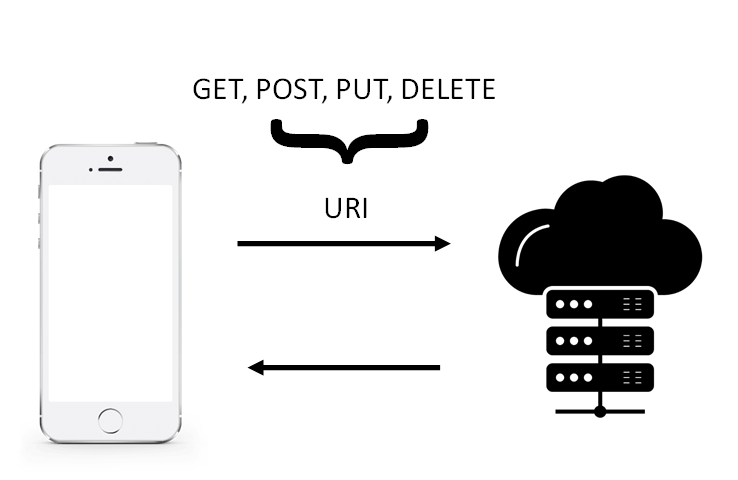
Dans la mesure du possible, les ressources doivent pouvoir être mises en cache du côté client ou côté serveur. Les réponses du serveur doivent également contenir des informations indiquant si la mise en cache est autorisée pour la ressource distribuée. L'objectif est d'améliorer les performances côté client, tout en augmentant l'évolutivité du serveur.

Dans les API REST, les appels et les réponses passent par différentes couches. En règle générale, ne partez pas du principe que les applications client et serveur se connectent directement l'une à l'autre. Il peut exister un certain nombre d'intermédiaires différents dans la boucle de communication. Les API REST doivent être conçues de manière à ce que ni le client ni le serveur ne puissent savoir s'ils communiquent avec l'application finale ou avec un intermédiaire.

Les API REST envoient généralement des ressources statiques, mais dans certains cas, les réponses peuvent également contenir du code exécutable (par exemple, des applets Java). Dans ce cas, le code ne doit être exécuté qu'à la demande.

1. Fonctionnement des API REST

Les API REST communiquent via des requêtes HTTP pour exécuter des fonctions de base de données standard telles que la création, la lecture, la mise à jour et la suppression d'enregistrements (également appelées CRUD) dans une ressource. Par exemple, une API REST utilise une requête GET pour récupérer un enregistrement, une requête POST pour en créer un, une requête PUT pour mettre à jour un enregistrement et une requête DELETE pour en supprimer un. Toutes les méthodes HTTP peuvent être utilisées dans les appels d'API. Une API REST bien conçue est similaire à un site Web exécuté dans un navigateur Web avec une fonctionnalité HTTP intégrée.



Ces informations peuvent être fournies à un client dans pratiquement n'importe quel format. JSON est le plus souvent utilisé, car il est lisible à la fois par l'utilisateur et les machines, et il est indépendant du langage de programmation.

Les en-têtes de demande et les paramètres sont également importants dans les appels d'API REST, car ils contiennent des informations d'identification importantes telles que les métadonnées, les autorisations, les identificateurs URI (Uniform Resource Identifier), la mise en cache, les cookies, etc. Les en-têtes de requête et les en-têtes de réponse, ainsi que les codes d'état HTTP conventionnels, sont utilisés dans les API REST bien conçues.

1. Retrofit
   1. Présentation

La plupart des applications mobiles consomment des API REST JSON pour récupérer toutes les données nécessaires à leur fonctionnement.

Retrofit est aujourd’hui une des façons les plus simples d’implémenter des appels à des webservices REST.

Il s’agit d’un client REST. Cette librairie permet d’implémenter plus facilement et rapidement des requêtes réseau sur Android (Java ou Kotlin).

Retrofit nous évite ainsi d’installer manuellement toutes les parties nécessaires à l’exécution d’une requête, comme par exemple la gestion des réponses JSON ou la création d’une [AsyncTask](https://blog.axopen.com/2016/03/les-asynctask/).

Cela permet un gain de temps conséquent et un code plus clair pour des performances équivalentes.

Depuis, Retrofit 2.6 supporte l’utilisation des coroutines. Cela permet de ne plus utiliser de callback ou la fonction enqueue.

* 1. Mise en place

1.Ajouter les dépendances

2.Ajouter les permissions

3.Créer le model

4.Créer le service client

5.Définir le style pour une ligne de liste

6.Définir un adapter​​​​​​​

7.Initialiser et lier les éléments

* 1. Ajouter les dépendances

Build.gradle du projet.

// Top-level build file where you can add configuration options common to all sub-projects/modules.  
buildscript {  
   
 ext {  
 version\_moshi = "1.9.3"  
 version\_retrofit = "2.9.0"  
 }  
  
 repositories {  
 google()  
 mavenCentral()  
 }  
 dependencies {  
 classpath "com.android.tools.build:gradle:7.0.4"  
 classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:1.6.0"  
 }  
}  
  
task clean(type: Delete) {  
 delete rootProject.buildDir  
}

Le code suivant montre les dépendances nécessaires pour utiliser Retrofit. Ce code est à ajouter dans le fichier build.gradle.

//Retrofit

dependencies {

implementation "com.squareup.moshi:moshi:$version\_moshi"

implementation "com.squareup.moshi:moshi-kotlin:$version\_moshi"

implementation "com.squareup.retrofit2:converter-moshi:$version\_retrofit"

}

Pour l'utilisation des coroutines, car il est obligatoire d'utiliser des coroutines pour accéder à une source de données distante afin d'éviter de ralentir l'IHM et donc de décevoir l'utilisateur de l'application.

//viewModelScope

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. Ajouter les permissions

Le code suivant montre les permissions nécessaires au bon fonctionnement de Retrofit. En effet, il est nécessaire que l'application ait la permission d'accéder à internet afin d'accéder au service REST.

<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="fr.eni.randomchuck">

<uses-permission android:name="android.permission.INTERNET" />

[…]

* 1. Créer le model

Le code suivant montre une entité permettant de stocker un enregistrement de données récupéré par Retrofit. Les propriétés de l'entité doivent avoir le même nom que le nom des propriétés renvoyées par le service REST sinon l'annotation @Json doit être utilisée. Par exemple, la propriété image permet de stocker la propriété icon\_url de l'enregistrement renvoyé par le service REST.

data class Joke(

val id: String,

@Json(name = "icon\_url") val image: String,

@Json(name = "value") val joke: String

)​

* 1. Créer le service client

Le code suivant permet de représenter le service REST définit à l'adresse définit dans la variable BASE\_URL ("https://api.chucknorris.io/jokes/"). Ce service offre une méthode qui permet d'interroger l'adresse suivante : https://api.chucknorris.io/jokes/random"

interface ChuckService {

companion object {

val BASE\_URL = "https://api.chucknorris.io/jokes/"

val moshi = Moshi.Builder().add(KotlinJsonAdapterFactory()).build()

val retrofit = Retrofit.Builder()

.addConverterFactory(MoshiConverterFactory.create(moshi))

.baseUrl(BASE\_URL)

.build()

}

@GET("random")

suspend fun randomFact(): Joke

}

​

object ChuckApi {

val retrofitService: ChuckService by lazy { retrofit.create(ChuckService::class.java) }

}

​

* 1. Utiliser le service client

Le code suivant permet de montrer comment utiliser le service REST dans un ViewModel. On notera l'utilisation d'une coroutine car il est recommandé de consulter des sources de données extérieur dans un thread parallèle.

class RandomjokeViewModel(application: Application): AndroidViewModel(application)

{

val joke = MutableLiveData<Joke>()

val status = MutableLiveData<String>()

​

fun randomPersonne()

{

viewModelScope.launch {

try {

var result = ChuckApi.retrofitService.randomFact()

status.value = "OK"

joke.value = result

}

catch (e: Exception)

{

status.value = "KO : ${e.message}"

}

}

}

}

* 1. Contrôleur et IHM

<?xml version="1.0" encoding="utf-8"?>  
<layout>  
  
 <data>  
 <variable  
 name="Model"  
 type="fr.acos.retrofitchucknorris.RandomjokeViewModel" />  
 </data>  
  
 <androidx.constraintlayout.widget.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=".MainActivity"  
 android:background="#E91E63"  
 >  
 <TextView  
 android:id="@+id/tv\_bonjour"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="!!!!!!! Bonjour à toutes et à tous !!!!!!!"  
 app:layout\_constraintBottom\_toBottomOf="parent"  
 app:layout\_constraintLeft\_toLeftOf="parent"  
 app:layout\_constraintRight\_toRightOf="parent"  
 app:layout\_constraintTop\_toTopOf="parent"  
 android:textColor="#FFEB3B"  
 />  
 <TextView  
 android:id="@+id/tv\_joke"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 app:layout\_constraintTop\_toBottomOf="@id/tv\_bonjour"  
 app:layout\_constraintLeft\_toLeftOf="parent"  
 app:layout\_constraintRight\_toRightOf="parent"  
 android:text="@{Model.joke.joke}"  
 android:textColor="#FFEB3B"  
 />  
 <Button  
 android:id="@+id/btn\_blague"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 app:layout\_constraintTop\_toBottomOf="@id/tv\_joke"  
 app:layout\_constraintLeft\_toLeftOf="parent"  
 android:backgroundTint="#FFEB3B"  
 android:textColor="@color/black"  
 android:text="BLAGUE"  
 android:onClick="@{() -> Model.randomBlague()}"  
 />  
 </androidx.constraintlayout.widget.ConstraintLayout>  
</layout>

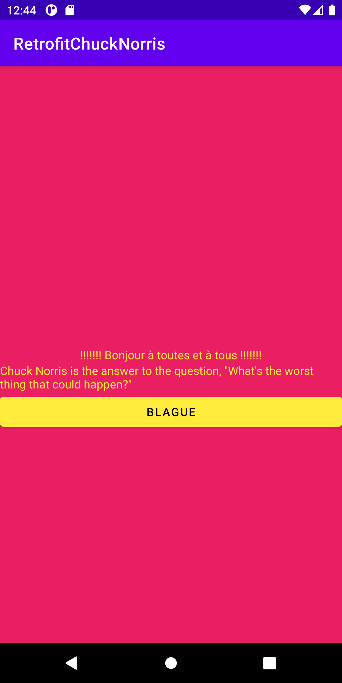
buildFeatures {

dataBinding true

}

class MainActivity : AppCompatActivity() {  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 val db = ActivityMainBinding.inflate(layoutInflater)  
 setContentView(db.root)  
  
 val vm = ViewModelProvider(this).get(RandomjokeViewModel::class.java)  
  
 vm.randomBlague()  
  
 vm.joke.observe(this, Observer {  
 db.model = vm  
 })  
 }  
}

* 1. Rendu final



1. Autre démonstration
2. Conclusion

Grâce à sa simplicité, Retrofit est largement utilisé pour interroger des services REST.