Retrofit

Retrofit es una librería para Android y java compatible con Kotlin para hacer peticiones a un API de forma sencilla.

* 

## **Javier Ocampo**

* Read [more posts](https://engineering.creativesociety.mx/author/javier/) by this author.
* 

#### [JAVIER OCAMPO](https://engineering.creativesociety.mx/author/javier/)

31 AUG 2020 • 4 MIN READ



Retrofit es una librería para Android y java compatible con Kotlin para hacer llamadas de red, obtener el resultado y “parsearlo” de forma automática a su objeto, esto facilita mucho realizar peticiones a un API y procesar la respuesta.

Link de Retrofit: <https://square.github.io/retrofit/>

## **Antes de empezar**

Para utilizar Retrofit en Android es necesario agregar la librería y algún serializador (converter) en *app/build.gradle*, para este caso utilizaremos el serializador gson.

implementation 'com.squareup.retrofit2:retrofit:2.8.1'

implementation 'com.google.code.gson:gson:2.8.6'

implementation 'com.squareup.retrofit2:converter-gson:2.8.1'

Retrofit cuenta con soporte para varios serializadores que pueden revisar en el siguiente link: <https://square.github.io/retrofit/>

También necesitaremos las clases que representen los objetos de la respuesta esperada en formato *POJO* (Plain Old Java Object), estas clases las utilizará el serializador para “parsear” la respuesta a los objetos esperados y poderlos utilizar fácilmente.

## **Peticiones con retrofit**

Empecemos a agregar las peticiones del API al proyecto, para esto Retrofit necesita una interfaz donde agregaremos las llamadas al API.

interface CatalogosRequests{

@GET("api/catalogos/colores")

@Headers("Accept: application/json")

fun colores(): Call<List<Color>>

@FormUrlEncoded

@POST("api/catalogos/color")

@Headers("Accept: application/json")

fun agregarColor(@Field("color") color: String): Call<Color>

}

Para este caso “Color” es el *POJO* que corresponde a la respuesta de la petición del API.

La anotación (representada con @) *@GET* y *@POST* corresponde al método de la petición,hay otras anotaciones que se pueden utilizar como *@DELETE*, *@PUT*, *@PATCH*.

Headers (Cabeceras)

Con la anotación *@Headers* se pueden agregar las cabeceras necesarias a la petición, también se puede agregar *@Header(“”)* como parámetro de la función.

## **Encoding**

Para las peticiones de tipo *@POST* es necesario agregar la anotación *@FormUrlEncoded* y *@Multipart* que corresponden a la cabecera *application/x-www-form-urlencoded* Y *multipart/form-data*.

El parámetro *@Field(“”)* se utiliza para enviar datos en la petición utilizando la cabecera *@FormUrlEncoded* y el parámetro *@Part(“”)* se utiliza con la cabecera *@Multipart*.

## **Realizando peticiones con retrofit**

Retrofit tiene dos objetos especiales que son *Call<T>* y *Callback<T>* el primero es el objeto que realiza la petición y el segundo es el objeto que recibe el resultado de la petición, “T” representa el objeto que se espera de la respuesta y en ambos tiene que ser el mismo; en este caso para la primer petición tenemos *Call<List<Color>>*, entonces, el objeto de respuesta sería *Callback<List<Color>>*.

Para realizar la petición tenemos que utilizar el objeto Retrofit, que se utiliza de la siguiente manera:

Val retrofit = retrofit.Builder()

.baseUrl(BASEURL)

.addConverterFactory(GsonConverterFactory.create())

.build()

Donde “BASEURL” es la url base del API.

Al obtener el objeto Retrofit estamos listos para realizar peticiones al API utilizando la interfaz que creamos anteriormente.

Ahora ocuparemos el objeto retrofit para obtener el objeto con el que podremos realizar las peticiones para el API

val requestObject = retrofit.create(CatalogosRequests::class.java)

El método “create” de retrofit recibe como parámetro la interfaz que hemos creado anteriormente, al acceder al objeto podremos ver las funciones que declaramos anteriormente y realizar las peticiones al API.

Ahora utilizando el objeto requestObject realizaremos una petición al API de la siguiente manera

requestObject.colores().enqueue(object: Callback<List<Color>>{

override fun onResponse(call: Call<List<Color>>, response: Response<List<Color>>) {

}

override fun onFailure(call: Call<List<Color>>, t: Throwable) {

}

})

El método “enqueue” nos permite realizar la petición de manera asincrónica, también existe el método “execute” que nos permite obtener la respuesta de manera sincrónica.

Al realizar la petición de manera asincrónica no bloquearemos el hilo principal de la aplicación y a través del objeto Callback podremos obtener la respuesta de la petición.

La función “onFailure” del objeto Callback se ejecuta cuando ocurre algún error en la petición o en el procesamiento o “parseo” de los datos, y la función “onResponse” se ejecuta cuando la petición se ha realizado de forma correcta, esto no quiere decir que la petición haya sido exitosa ya que esta función se ejecuta con cualquier código de respuesta http como *400, 200 o 201* y a través del objeto “response” de esta función podemos obtener toda la información de la petición.

Podemos obtener el código de respuesta del servidor de la siguiente manera:

response.code()

Para obtener los datos de respuesta que en este caso es List<Color>:

response.body()!!

Para saber si la petición se realizó de forma exitosa se puede utilizar el siguiente atributo:

response.isSuccessful

Un ejemplo de cómo se podría procesar la respuesta de la petición sería de la siguiente manera:

override fun onResponse(call: Call< List<Color>>, response: Response<List<Color>>) {

if (response.isSuccessful && response.code() == 200) {

procesarInformacion(response.body()!!)

} else if (response.code() == 422) {

mostrarErrorDeValidacion()

} else {

mostrarError()

}

}

## **Conclusión**

En conclusión, Retrofit es una excelente librería para realizar peticiones a un API de una manera sencilla y rápida, gracias al “parseo” automático de la respuesta nos ahorra mucho trabajo, contiene muchas opciones muy interesantes como configurar el cliente http que se utilizara o agregar “loggers” para saber lo que sucede al realizar las peticiones, creo que es una librería que vale la pena tener en cuenta y explorar todas sus opciones.