



# API REST y acceso a recursos remotos (Parte I)

---

## Cliente - Servidor

---

### Competencias:

- Identificar que es un cliente.
- Identificar que es un servidor.
- Comprender la integración entre un cliente y un servidor.

### Introducción

Internet ha traído un cambio revolucionario en el ámbito de las tecnologías, interconectando a el mundo entero. En este contexto, las tecnologías de comunicación en internet siguen una arquitectura y una estructura específica para conectarse entre sí. La más popular es la arquitectura cliente-servidor, la cual profundizaremos en este capítulo.

Es fundamental entender estos conceptos básicos de programación moderna, dado que hoy por hoy la gran mayoría de los sistemas y aplicaciones que se construyen siguen un standard de arquitectura basada en el concepto cliente-servidor.

## Arquitectura cliente-servidor

La arquitectura cliente-servidor se denomina estructura de computación de red porque cada solicitud y sus servicios asociados se distribuyen a través de una red privada o pública en internet.

En la arquitectura cliente-servidor, cuando la computadora cliente envía una solicitud de datos al servidor a través de Internet, el servidor acepta la solicitud, la procesa y entrega los paquetes de datos solicitados al cliente. Una característica especial es que la computadora servidor tiene el potencial de administrar numerosos clientes al mismo tiempo. Además, un solo cliente puede conectarse a numerosos servidores, donde cada servidor proporciona un conjunto diferente de servicios a ese cliente específico.



Imagen 1. Arquitectura clientes-servidor.

Las computadoras cliente proporcionan una interfaz para permitir que un usuario de la computadora solicite servicios del servidor y muestre los resultados que el servidor devuelve. Los servidores esperan que lleguen las solicitudes de los clientes y luego responden a ellos. Idealmente, un servidor proporciona una interfaz transparente estandarizada a los clientes para que los clientes no necesiten conocer los detalles del sistema (es decir, el hardware y el software) que proporciona el servicio.

Los clientes a menudo se encuentran en estaciones de trabajo o en computadoras personales, mientras que los servidores se encuentran en otras partes de la red, generalmente son las máquinas más potentes. Este modelo de computación es especialmente efectivo cuando los clientes y el servidor tienen tareas distintas que realizan rutinariamente.

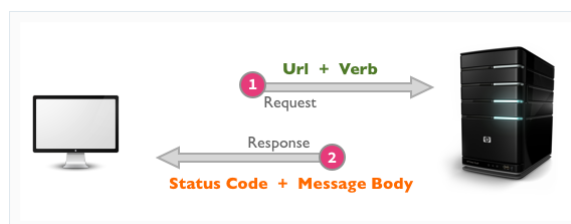


Imagen 2. Representación de modelo de comunicación Request - Response.

En el procesamiento de datos de un hospital, por ejemplo, una computadora cliente puede ejecutar un programa de aplicación para ingresar información del paciente mientras la computadora del servidor ejecuta otro programa que administra la base de datos en la que la información se almacena permanentemente. Muchos clientes pueden acceder a la información del servidor simultáneamente y, al mismo tiempo, una computadora cliente puede realizar otras tareas, como enviar un correo electrónico.

## Ciente

En una arquitectura cliente-servidor de una red computacional el cliente se puede definir como un procesador remoto que recibe y envía peticiones de un servidor. Cada teléfono móvil android desde su ubicación en el planeta es un cliente para cada una de las empresas o servicios que prestan diferentes aplicaciones desde sus servidores principales.



Imagen 3. Dispositivos cliente frecuentes.

## Principales Funciones de un cliente

- Iniciar el request o solicitud.
- Esperar y recibir respuestas.
- Poder conectarse con un servidor o múltiples servidores al mismo tiempo.
- Interactuar con los usuarios finales usando una Graphical User Interface (GUI)

## Servidor

Un servidor es un computador de red, programa o dispositivo que procesa solicitudes de un cliente. En la internet hoy por hoy un servidor web es una computadora o una red de computadoras que utiliza el protocolo HTTP para enviar datos y archivos a la computadora de un cliente cuando el este último lo solicita.

En el caso de una red de área local, por ejemplo, un servidor de impresoras administra una o más impresoras e imprime los archivos que le envían los equipos cliente. Los servidores de red (que administran el tráfico de red) y los servidores de archivos (que almacenan y recuperan archivos para clientes) son dos ejemplos mas de servidores.



Imagen 4. Google datacenter (servidores).

## Principales Funciones de un Servidor

- Esperar por los requerimientos de un cliente.
- Recibido un requerimiento, procesar, preparar y enviar las respuestas.
- Aceptar múltiples conexiones de una largo número de clientes.
- Normalmente, no interactuar directamente con los usuarios finales.

## Cliente-Servidor en la actualidad

Entendiendo los conceptos básicos anteriores, entremos en el escenario de la tecnología actual con otro ejemplo, el requerimiento de los datos meteorológicos de hoy para nuestra ciudad.

Antes de la existencia de las computadoras, las personas recibían estas noticias meteorológicas a través del diario(períodico) o lo hacían escuchando la radio o televisión transmitiendo noticias del clima.

En el mundo moderno con computadoras y servidores, los usuarios, que buscan el informe meteorológico a través de sus dispositivos móviles enviando peticiones a través de direcciones web (URL) que el servidor provee para la ejecución de un servicio o tarea específico, proporcionando información actualizada del clima.

Por lo tanto, los clientes y servidores son dos computadoras diferentes en diferentes partes del mundo que están conectadas a través de Internet, por lo tanto, como usuarios, podemos bajar una aplicación de consulta del clima y navegar por el informe meteorológico, gracias a los servicios expuestos desde los servidores de la aplicación.

Algunos factores adicionales entran en acción con tales tecnologías de recopilación de datos en línea. Un periódico o radio usa su idioma local para darle el informe meteorológico y otra información; sin embargo, para la arquitectura cliente-servidor en la web, se deben considerar factores específicos:

- Un conjunto específico de idiomas junto con un estándar de comunicación, exclusivamente un protocolo para la interacción de dos sistemas. Los más populares son HTTP y HTTPS (Protocolo seguro de transferencia de hipertexto).
- Mecanismo y protocolo para solicitar los aspectos requeridos del servidor. Eso podría estar en cualquier estructura de datos formateados. Los formatos más implementados y populares se realizan en XML y JSON.
- Respuesta del servidor enviando en una estructura de datos formateados (generalmente XML o JSON).

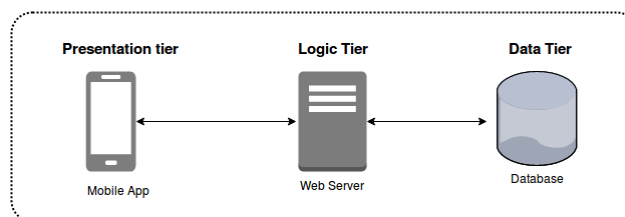


Imagen 5. Representación arquitectura moderna tradicional.

# API Rest

---

## Competencias:

- Conocer qué es un api rest.
- Comprender cómo interactúa un api rest con las aplicaciones android.
- Diferenciar los métodos para consumir e interactuar con una API Rest.

## Introducción

En este capítulo estudiaremos los conceptos asociados a un api REST y aprenderemos a trabajar con este estilo de arquitectura distribuida muy popular hoy por hoy. REST es una tecnología que utiliza un conjunto de protocolos y estándares para intercambiar datos entre aplicaciones.

Al consumir servicios web REST desde nuestras aplicaciones somos capaces de desarrollar sistemas móviles multi-usuario, masivos y de largo alcance, entendiendo que desde el lugar en que se encuentre un cliente android, este se puede conectar con servicios centralizados expuestos con la tecnología rest.

En la actualidad casi no existen proyectos o aplicaciones que no disponga de una API REST para la creación de servicios profesionales. WhatsApp, YouTube, Twitter, Facebook y cientos de miles de empresas generan negocio gracias a REST y las APIs REST, sin esto, el crecimiento en horizontal sería prácticamente imposible y REST lo facilita gracias a que es el estándar más lógico, eficiente y habitual en la creación de APIs para servicios de Internet.

Conocer cómo funcionan estos servicios proporcionará al alumno herramientas para trabajar en cualquier empresa que ofrezca servicios rest, o necesite que las aplicaciones se conecten a un tercero.

## Api REST

Una api REST simplifica el desarrollo al admitir métodos HTTP estándar, manejo de errores y otras convenciones RESTful. Un api REST expone recursos a través de una o distintas direcciones web (URL) utilizando arquitectura basada en HTTP, exponiendo servicios web para el procesamiento de distintas tareas y operaciones.

REST es el acrónimo de Representational State Transfer, un estilo arquitectónico para sistemas hipertexto distribuidos que fue presentado por primera vez por Roy Fielding en el año 2000.

Al igual que cualquier otro estilo arquitectónico, REST también tiene sus propias seis(6) restricciones guía que deben cumplirse si una interfaz necesita ser referida como una interfaz RESTful.

REST le pide a los desarrolladores que usen métodos HTTP explícitamente y de manera consistente con la definición del protocolo. Este principio de diseño REST básico establece un mapeo uno a uno entre las operaciones de creación, lectura, actualización y eliminación (CRUD) y los métodos HTTP.

## Principios REST

1. **Cliente-servidor:** al separar las preocupaciones de la interfaz de usuario de las preocupaciones de almacenamiento de datos, mejoramos la portabilidad de la interfaz de usuario en múltiples plataformas y mejoramos la escalabilidad al simplificar los componentes del servidor.
2. **Sin estado (Stateless):** cada solicitud del cliente al servidor debe contener toda la información necesaria para comprender la solicitud y no puede aprovechar ningún contexto almacenado en el servidor. Por lo tanto, el estado de la sesión se mantiene completamente en el cliente.
3. **Caché:** las restricciones de caché requieren que los datos dentro de una respuesta a una solicitud se etiqueten implícita o explícitamente como almacenables o no almacenables. Si una respuesta es almacenable en caché, se le otorga a un caché de cliente el derecho de reutilizar esos datos de respuesta para solicitudes equivalentes posteriores.
4. **Interfaz uniforme:** al aplicar el principio de generalidad de ingeniería de software a la interfaz del componente, se simplifica la arquitectura general del sistema y se mejora la visibilidad de las interacciones. Para obtener una interfaz uniforme, se necesitan múltiples restricciones arquitectónicas para guiar el comportamiento de los componentes. REST está definido por cuatro restricciones de interfaz:
  - Identificación de recursos.
  - Manipulación de recursos a través de representaciones.
  - Mensajes autodescriptivos.
  - Hipermedia como motor del estado de la aplicación.
5. **Sistema en capas:** el estilo de sistema en capas permite que una arquitectura se base en capas jerárquicas al restringir el comportamiento de los componentes de modo que cada componente no pueda ver más allá de la capa inmediata con la que están interactuando.
6. **Código bajo demanda (opcional):** REST permite ampliar la funcionalidad del cliente descargando y ejecutando código en forma de scripts. Esto simplifica a los clientes al reducir la cantidad de funciones que se deben implementar previamente.

## REST - CRUD

REST significa REpresentational State Transfer y describe recursos (en nuestro caso, URL) en los que podemos realizar acciones.

CRUD significa Crear, Leer, Actualizar, Eliminar, son las acciones que realizamos. Aunque, REST y CRUD son los mejores amigos, los dos pueden funcionar bien por sí mismos. De hecho, cada vez que programamos un sistema que permite agregar, editar y eliminar elementos de la base de datos, y una interfaz que permite que viajen los datos de estos elementos; hemos estado trabajando con CRUD sin saberlo.

## HTTP Response Codes

Si intentamos acceder a una página que no existe, el servidor te devolverá la página 404 No encontrada o error 404, si arruinamos nuestro código rest del servidor, obtendremos un error 500 de sistemas. Estos son códigos de respuesta que el servidor envía para que su navegador sepa lo que está sucediendo.

### Tipos de Respuesta

- Entre 200-299 significa que la solicitud fue exitosa.
- 300-399 significa que la solicitud estuvo bien, pero debe hacer otra cosa.
- 400-499 es un error dónde no se encuentra algún objeto o recurso, o no se ha invocado correctamente.
- 500-599 es un error realmente malo.

**Código 200:** Obtendrá de una solicitud GET, PUT o DELETE. Significa que la solicitud se verificó y se tomaron las medidas apropiadas.

**Código 201 Creado:** Le notifica que el comando POST creó correctamente el objeto publicado.

**Código 404 No encontrado:** Significa que no se encontró el recurso de solicitud. Puede obtener esto de GET, PUT o DELETE.

**Código 406 no aceptable:** El verbo no está permitido en los recursos que solicitó (Más en esta próxima parte)

**Código 500 Error interno de servidor:** Algo sucedió terriblemente malo en el código del sistema

## Operaciones REST

Para comprender la forma en que funciona la arquitectura REST es fundamental conocer los métodos HTTP, por medio de los cuales le indicamos al servidor la forma en que debe de tratar una determinada petición, dicho esto, una misma URL puede ser tratada de forma diferente por el servidor.

HTTP define una gran cantidad de métodos que son utilizados en diferentes circunstancias. Los más relevantes para la construcción de servicios REST son los siguientes:

- **GET:** Es utilizado únicamente para consultar información al servidor, muy parecidos a realizar un SELECT a la base de datos. No soporta el envío del payload.
- **POST:** Es utilizado para solicitar la creación de un nuevo registro, es decir, algo que no existía previamente, es decir, es equivalente a realizar un INSERT en la base de datos. Soporta el envío del payload.
- **PUT:** Se utiliza para actualizar por completo un registro existente, es decir, es parecido a realizar un UPDATE a la base de datos. Soporta el envío del payload.
- **PATCH:** Este método es similar al método PUT, pues permite actualizar un registro existente, sin embargo, este se utiliza cuando actualizar solo un fragmento del registro y no en su totalidad, es equivalente a realizar un UPDATE a la base de datos. Soporta el envío del payload.
- **DELETE:** Este método se utiliza para eliminar un registro existente, es similar a DELETE a la base de datos. No soporta el envío del payload.
- **HEAD:** Este método se utilizar para obtener información sobre un determinado recurso sin retornar el registro. Este método se utiliza a menudo para probar la validez de los enlaces de hipertexto, la accesibilidad y las modificaciones recientes.



# HTTP VERBS (Métodos de solicitud)

Cualquier desarrollador web que haya tenido que lidiar con datos de formularios conocerá los métodos GET y POST. El primero enviará datos al servidor a través de una cadena de consulta que se parece a esto "direccion?Key2=value2&key2=value2" y el segundo envía los datos a través de encabezados HTTP. Lo que quizás no nos hemos dado cuenta es que cada vez que carga una página (que no es una forma de tipo POST) está realiza una solicitud de tipo GET. Entonces, cuando hacemos clic en el enlace de nuestra preferencia, estaremos haciendo una solicitud GET.

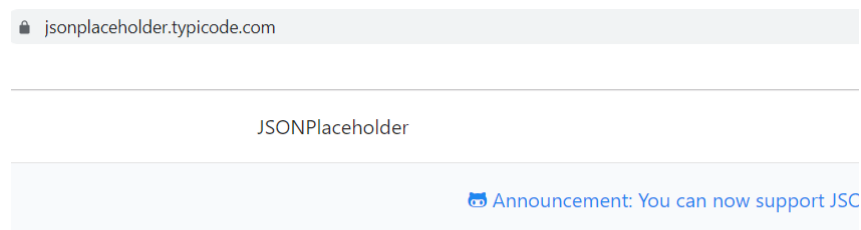
Hay otros dos verbos de los que quizás no hayas oído hablar porque los navegadores no los admiten: PUT y DELETE. La acción de eliminar es obvia, si le indica al servidor que elimine algo. La instrucción PUT es un poco más compleja, es lo mismo que la acción POST desde el punto de vista que enviar datos a través de encabezados, pero está diseñada para modificar o actualizar datos o archivos.

## Ejercicio 1:

Descargar e instalar postman y utilizarlo para el consumo de métodos REST del api público fake JsonPlaceholder.

Postman es una plataforma de colaboración para el desarrollo ágil de APIs. Las características de Postman simplifican cada paso de la creación de una API y agilizan la colaboración para que podamos crear más y mejores APIs.

1. Descargar e instalar postman desde la siguiente url: <https://www.getpostman.com/downloads/>
2. Ir a la web de JsonPlaceholder <https://jsonplaceholder.typicode.com>, para utilizar los métodos REST de su api con data de prueba. Utilizaremos los datos de posts para este ejemplo como lo muestra la siguiente imagen:



## Resources

JSONPlaceholder comes with a set of 6 common resources:

<a href="#">/posts</a>	100 posts
<a href="#">/comments</a>	500 comments
<a href="#">/albums</a>	100 albums
<a href="#">/photos</a>	5000 photos
<a href="#">/todos</a>	200 todos
<a href="#">/users</a>	10 users

Imagen 6. JSONPlaceholder.

3. Abrir el programa y presionar nuevo, seguido de request como lo muestra la siguiente imagen:

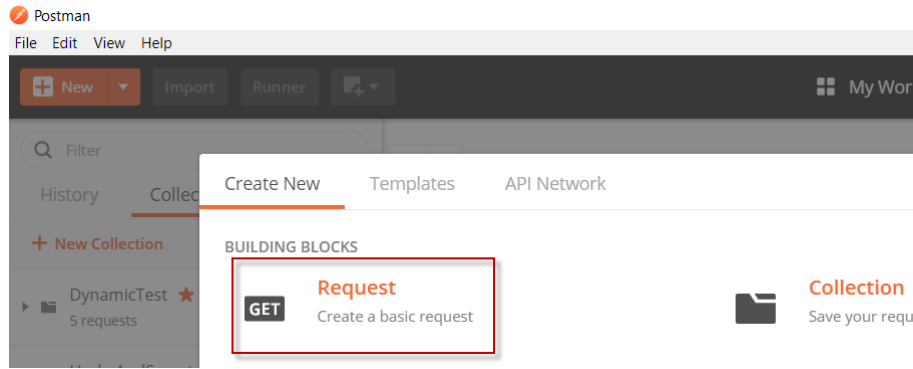


Imagen 7. Request.

4. Acto seguido, indicar un nombre a la operación de consultar posts y crear la colección Desafío Latam como se muestra en la siguiente imagen:

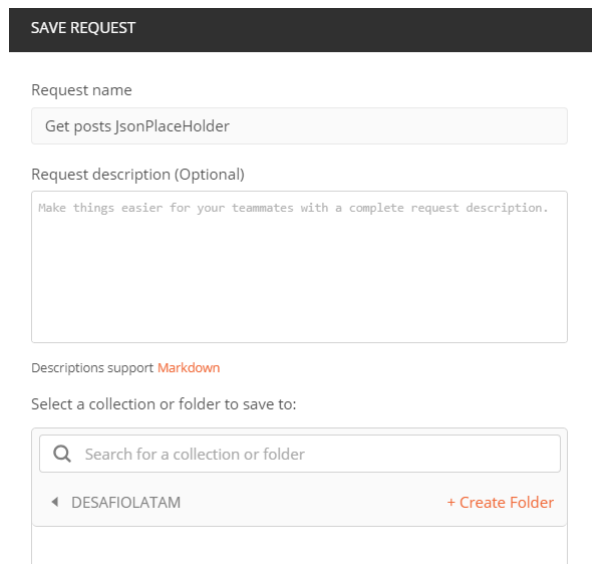


Imagen 8. Operación a consultar.

5. Copiando la url de la consulta de posts de la página tendremos una vista de detalle y rápido acceso a la operación en Postman que se visualizará de la siguiente manera:

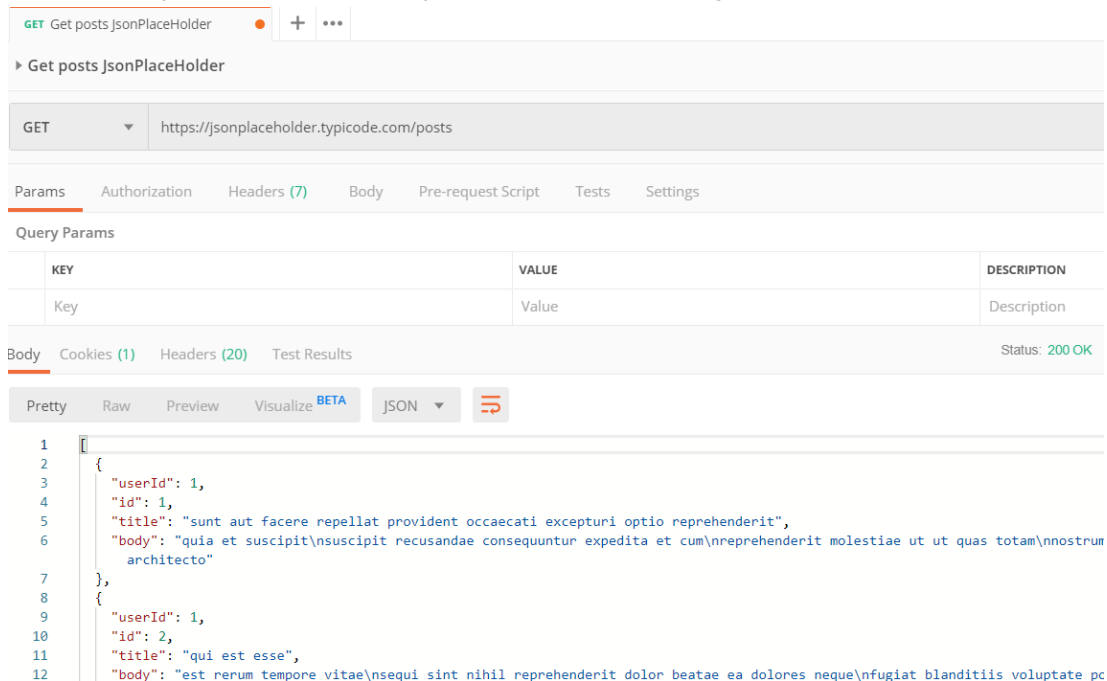


Imagen 9. Detalle de la consulta.

# REST en Android

Mientras desarrollamos aplicaciones android, regularmente nos encontramos trabajando con un back-end a través de las api REST. Estas api pueden ser desarrolladas por un equipo interno o por terceros.

A continuación comenzaremos los ejercicios interactuando desde nuestro entorno de desarrollo de aplicaciones android studio (IDE) con el api público de JsonPlaceholder ya que dispone de diferentes métodos que nos serán útiles a lo largo de esta unidad.

## Ejercicio 2:

Consultar el método get posts del api JsonPlaceholder desde android studio en un proyecto llamado "RestApi", con la utilización de los métodos nativos de HttpURLConnection, para visualizar por consola del IDE los resultados una vez inicie la aplicación android.

1. Crear un proyecto android llamado "RestApi" con las opciones minSdk 19, blank activity y lenguaje kotlin.

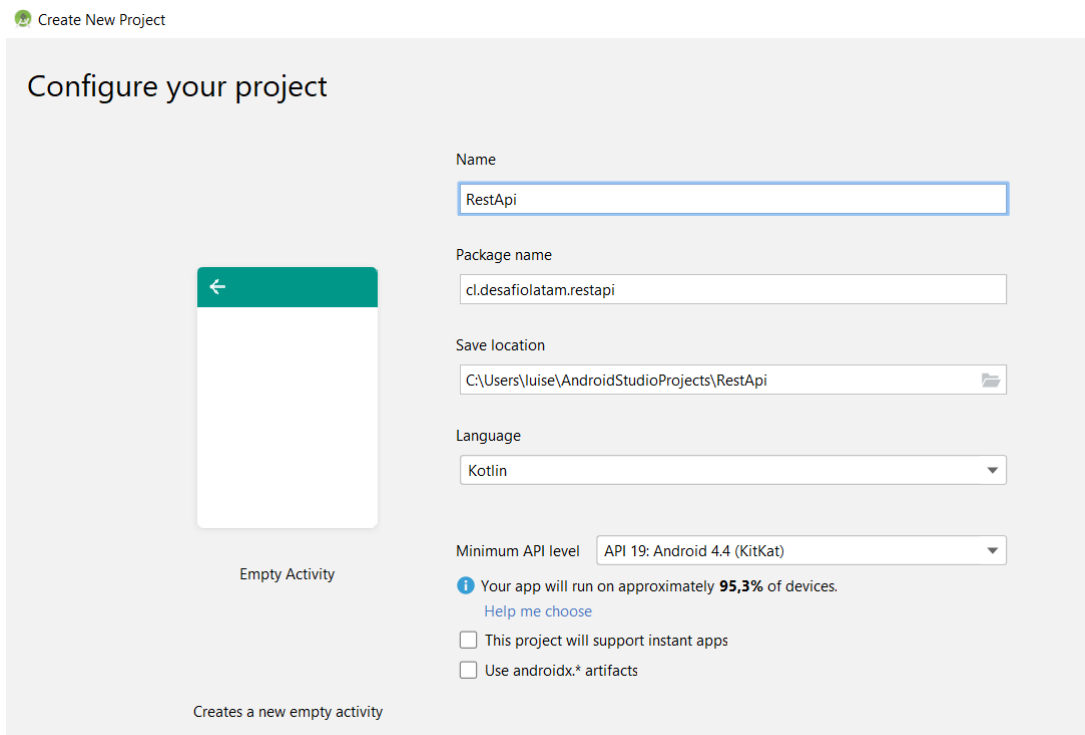


Imagen 10. Crear proyecto.

2. A continuación en el package `cl.desafiolatam.restapi`, creamos una nueva clase llamada `TEST.kt`, en la cual desarrollaremos un método llamado `connectApi()` con una variable `string` con la url del `get Posts` de la api `JsonPlaceholder` y una segunda variable de inicialización de `URLConnection`, de la siguiente forma:

```
fun connectApi() {
    try {
        val url =
            URL("https://jsonplaceholder.typicode.com/posts")
        val conn = url.openConnection() as HttpURLConnection
    } catch (e: MalformedURLException) {
        e.printStackTrace()
    } catch (e: IOException) {
        e.printStackTrace()
    }
}
```

3. A continuación desarrollamos el llamado indicando el tipo de método, validando el código de respuesta y leyendo la respuesta con la clase `BufferedReader` de la siguiente forma:

```
conn.requestMethod = "GET"

if (conn.responseCode != 200) {
    throw RuntimeException("Failed : HTTP error code : " + conn.responseCode)
}

val br = BufferedReader(
    InputStreamReader(
        conn.inputStream
    )
)

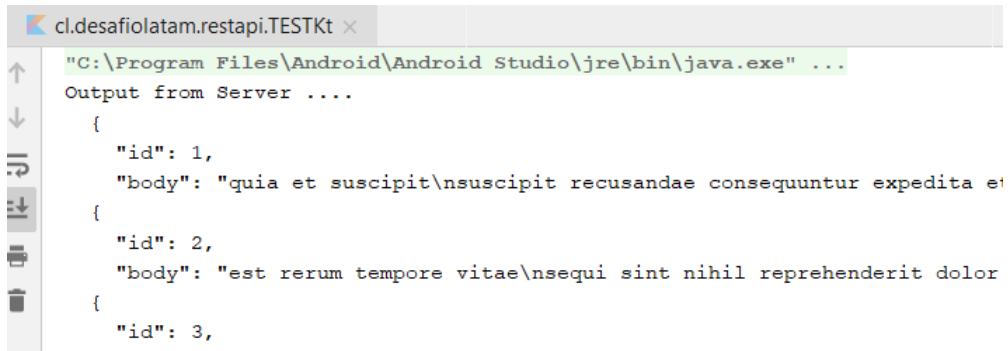
println("Output from Server ....")
while (br.readLine() != null) {
    println(br.readLine())
}

conn.disconnect()
```

4. Por último, creamos el método `main` de prueba fuera de la clase, haciendo referencia al método `connectApi()` que creamos anteriormente, de la siguiente forma:

```
fun main() {
    val test = TEST()
    test.connectApi()
}
```

5. Al ejecutar la clase TEST.kt, el resultado se visualiza como lo muestra la siguiente imagen:



```
cl.desafiolatam.restapi.TESTKt x
"C:\Program Files\Android\Android Studio\jre\bin\java.exe" ...
Output from Server ....
{
  "id": 1,
  "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita e
{
  "id": 2,
  "body": "est rerum tempore vitae\nsequi sint nihil reprehenderit dolor
{
  "id": 3,
```

Imagen 11. Ejecutar .TEST.kt .

De este ejercicio, es de notar que no hemos realizado el llamado al api desde los métodos del ciclo de vida de una actividad por una razón fundamental, es necesario seguir las políticas de android para el consumo de métodos HTTP fuera del hilo principal (main thread) y de manera asíncrona. Por lo tanto, comenzamos a visualizar con este ejemplo porque se utilizan librerías como Retrofit para realizar estas tareas, y la razón no es otra que ahorrarnos mucho código de creación de métodos asíncronos e hilos en segundo plano, además de código httpclient, y posibles errores al consumir los datos externos. En los próximos capítulos profundizaremos estos conceptos.

## Ventajas de usar REST

- **Separación entre el cliente y el servidor:** el protocolo REST separa totalmente la interfaz de usuario del servidor y el almacenamiento de datos. Eso tiene algunas ventajas cuando se hacen desarrollos. Por ejemplo, mejora la portabilidad de la interfaz a otro tipo de plataformas, aumenta la escalabilidad de los proyectos y permite que los distintos componentes de los desarrollos se puedan evolucionar de forma independiente.
- **Visibilidad, fiabilidad y escalabilidad.** La separación entre cliente y servidor tiene una ventaja evidente y es que cualquier equipo de desarrollo puede escalar el producto sin excesivos problemas. Se puede migrar a otros servidores o realizar todo tipo de cambios en la base de datos, siempre y cuando los datos de cada una de las peticiones se envíen de forma correcta. Esta separación facilita tener en servidores distintos el front y el back y eso convierte a las aplicaciones en productos más flexibles a la hora de trabajar.
- **La API REST siempre es independiente del tipo de plataformas o lenguajes:** la API REST siempre se adapta al tipo de sintaxis o plataformas con las que se estén trabajando, lo que ofrece una gran libertad a la hora de cambiar o probar nuevos entornos dentro del desarrollo. Con una API REST se pueden tener servidores PHP, Java, Python o Node.js. Lo único que es indispensable es que las respuestas a las peticiones se hagan siempre en el lenguaje de intercambio de información usado, normalmente XML o JSON.

# HTTP Client con Retrofit

---

## Competencias:

- Conocer los conceptos del cliente http retrofit.
- Implementar Retrofit en una aplicación Android.
- Conocer alternativas a Retrofit.

## Introducción

En este capítulo utilizaremos retrofit, el cual es una librería que encapsula los métodos de httpCLient. Es necesario aprender a implementar retrofit para la comunicación con apis externas a nuestras aplicaciones móviles, por su amplio uso y características que lo han posicionado como casi un estándar en la industria.

Retrofit es ampliamente utilizada en la comunidad de desarrollo android, convirtiéndose prácticamente en un estándar de la industria, razón que nos compromete estudiar y conocer sus métodos e implementación en nuestros proyectos.

Realizaremos ejercicios con distintas operaciones de creación, actualización, consulta y eliminación de datos.

## Retrofit

Es una librería Http client para específicamente realizar peticiones http tipo rest para android y java, permitiendo el envío y recepción de respuestas de datos entre nuestras aplicaciones y distintas plataformas de software a través de internet. Retrofit simplifica la codificación para lograr esta comunicación, gestionando y transformando nuestros datos para ser traspasados a nuestras clases pojo (plain old java object).

Para realizar una petición con Retrofit se necesitará lo siguiente:

- **Una clase Retrofit:** donde se cree la instancia a la librería de retrofit para utilizar sus métodos base, definiendo aquí nuestra url base que nuestra aplicación usará para todas las peticiones http.
- **Una interfaz de operaciones:** en la cual definamos todas nuestras peticiones con sus respectivos endpoints y datos de entrada o salida.
- **Clases de modelo de datos o pojo:** para mapear los datos que llegan desde el servidor traspasando estos a objetos e instancias automáticamente para ser usados en nuestra aplicación.

## GSON Converter

Habitualmente, las peticiones y respuestas son intercambiadas en formato JSON, razón por la cual se debe utilizar un convertidor junto con retrofit para lidiar con la serialización de datos en este formato. Retrofit puede deserializar por defecto respuestas de tipo OkHttp, pero admite el uso de convertidores como gson.

Gson es una biblioteca de java que nos permite convertir objetos java a json y viceversa de manera automática, sin mayores configuraciones.

### Ejercicio 3:

Agregar dependencias retrofit y gson al proyecto “RestApi”.

1. Abrir el archivo build.gradle a nivel de nuestra carpeta app e incluir las dependencias de retrofit y gson de la siguiente forma:

```
implementation 'com.squareup.retrofit2:retrofit:2.6.0'  
implementation 'com.squareup.retrofit2:converter-gson:2.6.0'
```

2. Sincronizar nuestro archivo gradle.



## Características de Retrofit como cliente HTTP

- **Declaración de API:** las anotaciones sobre los métodos de interfaz y sus parámetros indican cómo se maneja una solicitud.
- **Método request (solicitud):** cada método debe tener una anotación HTTP que proporcione el método de solicitud y la URL relativa. Hay cinco anotaciones incorporadas: GET, POST, PUT, DELETE, y HEAD. La URL relativa del recurso se especifica en la anotación. Ejemplo:

```
@GET("posts/list")
```

También puedes especificar parámetros en la url de la consulta, de la siguiente manera:

```
@GET("posts/list?sort=desc")
```

- **Manipulación de url:** una url de solicitud se puede actualizar dinámicamente utilizando bloques y parámetros de reemplazo en el método. Un bloque de reemplazo es una cadena alfanumérica rodeada por { and }. Un parámetro correspondiente debe ser agregado con la anotación @Path usando la misma cadena. Ejemplo:

```
@GET("group/{id}/posts")  
Call<List<Post>> groupList(@Path("id") int groupId);
```

Parámetros de consulta también se pueden agregar.

```
@GET("group/{id}/posts")  
Call<List<Post>> groupList(@Path("id") int groupId, @Query("sort") String sort);
```

Para combinaciones complejas de parámetros de consulta se puede utilizar un "Map".

```
@GET("group/{id}/posts")  
Call<List<Post>> groupList(@Path("id") int groupId, @QueryMap Map<String, String> options);
```

- **Request Body:** se puede especificar un objeto para usarlo como un cuerpo de solicitud HTTP con la anotación @Body.

```
@POST("posts/new")  
Call<Post> createPost(@Body Post post);
```

- **Form Encode, multipart:** los métodos también se pueden declarar para enviar datos codificados por formularios y multiparte. Los datos codificados en forma se envían cuando `@FormUrlEncoded` están presentes en el método. Cada par clave-valor se anota con `@Field` del nombre y el objeto que proporciona el valor.

```
@FormUrlEncoded
@POST("post/edit")
Call<Post> updatePost(@Field("first_name") String first, @Field("last_name") String last);
```

- **Manipulación de cabecera (header):** podemos establecer encabezados estáticos para un método utilizando la anotación `@Headers` anotación.

```
@Headers("Cache-Control: max-age=640000")
@GET("widget/list")
Call<List<Widget>> widgetList();

@Headers({
    "Accept: application/vnd.github.v3.full+json",
    "User-Agent: Retrofit-Sample-App"
})
@GET("users/{username}")
Call<User> getUser(@Path("username") String username);
```

## Ejercicio 4:

Crear una aplicación que se conecte a un api pública que disponga de varios métodos HTTP para utilizar servicios REST. En este caso utilizaremos la api pública de JsonPlaceholder.com la cuál nos brinda distintos métodos HTTP Rest, con data de prueba que nos ayuda a emular este tipo de desarrollos con comodidad.

**Nota:** A lo largo de la unidad estaremos trabajando sobre este ejercicio, ampliando sus funciones y operaciones.

1. Revisar los métodos expuestos en la página de la api pública siguiente: <https://jsonplaceholder.typicode.com/>

De los recursos del api anterior, nos vamos a concentrar en utilizar el referido a los posts. <https://jsonplaceholder.typicode.com/posts>



Imagen 12. Respuesta del api para retornar posts.

2. Abrir el proyecto del capítulo 2 “RestApi” y continuar con el desarrollo del mismo, agregando el archivo pojo “Post.kt”, con los mismos datos que se muestran en cada objeto de respuesta de post del api en un nuevo package “cl.desafiolatam.restapi.pojo”, esto para ser utilizado como clase de datos para ser instancia de la respuesta desde retrofit, por ejemplo, de la siguiente forma:

```
package cl.desafiolatam.restapi.pojo

import com.google.gson.annotations.Expose
import com.google.gson.annotations.SerializedName

class Post {

    @SerializedName("userId")
    @Expose
    var userId: Int? = null
    @SerializedName("id")
    @Expose
    var id: Int? = null
    @SerializedName("title")
    @Expose
```

```

var title: String? = null
@SerializedName("body")
@Expose
var body: String? = null

}

```

Existen alternativas prácticas para crear estos archivos pojo a partir de un json, como por ejemplo un convertidor online llamado “jsonschema2pojo.org”.

jsonschema2pojo

Generate Plain Old Java Objects from JSON or JSON-Schema.

Package: cl.desafiolatam.restapi

Class name: Posts

Target language:

☒ Java ☐ Scala

Source type:

☐ JSON Schema ☒ JSON

☐ YAML Schema ☐ YAML

Annotation style:

☐ Jackson 2.x ☐ Jackson 1.x

☒ Gson ☐ Moshi ☐ None

Imagen 13. Convertidor online Json - Pojo.

3. Creamos una nueva interface llamada “Api.kt” en el package cl.desafiolatam.restapi.utils, donde declaramos el método GET para obtener los datos de los posts existentes en el api, señalando la url relativa del método GET, en este caso “/posts” , de la siguiente forma:

```

package cl.desafiolatam.restapi.utils

import cl.desafiolatam.restapi.pojo.Post
import retrofit2.Call
import retrofit2.http.*

import java.util.ArrayList

interface Api {
    @GET("/posts")
    fun getAllPosts(): Call<ArrayList<Post>>
}

```

4. Creamos un nuevo archivo kotlin llamado RetrofitClient.kt dentro del package cl.desafiolatam.restapi.utils, que servirá de cliente base para retornar una instancia retrofit a la sección del código de nuestra app que lo requiera, con el código siguiente:

```
package cl.desafiolatam.restapi.utils

import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory

class RetrofitClient {

    companion object {
        private const val BASE_URL = "https://jsonplaceholder.typicode.com"

        fun retrofitInstance(): Api {

            val retrofit = Retrofit.Builder()
                .baseUrl(BASE_URL)
                .addConverterFactory(GsonConverterFactory.create())
                .build()

            return retrofit.create(Api::class.java)
        }
    }
}
```

5. Para mostrar la lista de posts a partir de los datos del api en forma de cards, como lo indica el estándar de las aplicaciones, es necesario que implementemos las librerías RecyclerView y CardView en nuestro archivo gradle de la app:

```
implementation 'com.android.support:cardview-v7:28.0.0'
implementation 'com.android.support:recyclerview-v7:28.0.0'
```

6. A continuación editamos el archivo principal del layout de la actividad "activity\_main.xml" para agregar la vista del RecyclerView que nos ayudará a crear la lista de elementos en la pantalla (UI) a partir de los datos retornados por el método GET del api. El archivo de estilo quedaría de la siguiente forma añadiendo un coordinator layout y un recyclerview:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    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:id="@+id/coordinatorLayout"
    android:layout_height="match_parent">

    <android.support.v7.widget.RecyclerView
        android:id="@+id/recyclerView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
```

```
        android:orientation="vertical"
        app:layoutManager="android.support.v7.widget.LinearLayoutManager" />

</android.support.design.widget.CoordinatorLayout>
```

7. A continuación crearemos un nuevo archivo layout en la carpeta res para dar un diseño de carta (card) a los datos de la api. Creamos el archivo "posts\_list.xml" con el único propósito de mostrar el título de cada ítem de la respuesta usando la clase `CardView`, de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.CardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    android:id="@+id/cardView"
    android:layout_width="match_parent"
    android:layout_height="64dp"
    android:layout_margin="8dp"
    card_view:cardCornerRadius="0dp"
    card_view:cardElevation="2dp">

    <RelativeLayout
        android:id="@+id/relativeLayout"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:paddingBottom="8dp"
        android:paddingLeft="8dp"
        android:paddingRight="8dp">

        <TextView
            android:id="@+id/txtTitle"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_centerVertical="true"
            android:text="Item 1"
            android:textAppearance="@style/TextAppearance.Compat.Notification.Title" />
    </RelativeLayout>

</android.support.v7.widget.CardView>
```

8. Para trabajar con el recyclerview necesitamos crear un adaptador que le indique a la vista cuales son los elementos a mostrar, su posición, cuántos son en total, permitiendo una inyección sobre el elemento de vista “recyclerview” declarado en el archivo activity\_main.xml. El código de la clase “CustomAdapter” es el siguiente:

```
package cl.desafiolatam.restapi.adapters

import android.support.v7.widget.RecyclerView
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.TextView
import cl.desafiolatam.restapi.R
import cl.desafiolatam.restapi.pojo.Post

import java.util.ArrayList

class CustomAdapter(val data: ArrayList<Post>) :
    RecyclerView.Adapter<CustomAdapter.CustomViewHolder>() {

    inner class CustomViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        val mTitle: TextView = itemView.findViewById(R.id.txtTitle)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): CustomViewHolder {
        val itemView = LayoutInflater.from(parent.context).inflate(R.layout.posts_list, parent, false)
        return CustomViewHolder(itemView)
    }

    override fun onBindViewHolder(holder: CustomViewHolder, position: Int) {

        holder.mTitle.text = data[position].title
    }

    override fun getItemCount(): Int {
        return data.size
    }
}
```

9. Por último, editamos nuestra actividad principal la “MainActivity.kt”, incluyendo un nuevo método llamado “loadDataApi()”, el cual cumplirá la función de hacer el llamado al método rest GET del api , utilizando retrofit. Adicionalmente, crearemos las instancias necesarias para activar el recyclerView a partir de la respuesta no nula del api de datos, como lo muestra el código siguiente:

```
package cl.desafiolatam.restapi
import android.os.Bundle
import android.support.design.widget.CoordinatorLayout
import android.support.v7.app.AppCompatActivity
import android.support.v7.widget.RecyclerView
import android.util.Log
import android.widget.Toast
import cl.desafiolatam.restapi.adapters.CustomAdapter
import cl.desafiolatam.restapi.pojo.Post
import cl.desafiolatam.restapi.utils.RetrofitClient
import retrofit2.Call
import retrofit2.Callback
import retrofit2.Response
import java.lang.Exception
import java.util.ArrayList

class MainActivity : AppCompatActivity(){
    internal lateinit var recyclerView: RecyclerView
    internal lateinit var mAdapter: CustomAdapter
    internal lateinit var coordinatorLayout: CoordinatorLayout

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        recyclerView = findViewById(R.id.recyclerView)
        coordinatorLayout = findViewById(R.id.coordinatorLayout)
        loadApiData()
    }

    private fun loadApiData(){
        val service = RetrofitClient.retrofitInstance()
        val call = service.getAllPosts()
        //Async
        call.enqueue(object : Callback<ArrayList<Post>> {
            override fun onResponse(call: Call<ArrayList<Post>>, response: Response<ArrayList<Post>>) {

                val postsFromApi = response.body()
                if (postsFromApi != null) {
                    mAdapter = CustomAdapter(postsFromApi)
                    recyclerView.adapter = mAdapter
                }
            }
        })

        override fun onFailure(call: Call<ArrayList<Post>>, t: Throwable) {
            Log.d("MAIN", "Error: "+t)
            Toast.makeText(
                applicationContext,
                "Error: no pudimos recuperar los posts desde el api",
                Toast.LENGTH_SHORT
            ).show()
        }
    }
}
```



```
}  
}
```

10. Iniciamos la aplicación para ver el resultado del código desarrollado anteriormente, la cual se debe ver como lo demuestran las siguientes imagen:



Imagen 14. Lista de datos consultados al api JsonPlaceholder a través del método Http GET, modo Rest con Retrofit, y visualizados con la ayuda de las librerías recyclerview y cardview de android.

## Ejercicio 5:

Continuando con el ejercicio anterior, profundizaremos la funcionalidad de nuestra app “Rest Api”, agregando en esta oportunidad la opción de eliminar un post de nuestra vista y emulando su eliminación en el api invocando el método http DELETE como lo indica la página del api, además realizaremos esta tarea aprendiendo a trabajar con una técnica denominada “SwipeToDelete”, de la misma forma que gmail envía a archivo los correos de tu bandeja.

### Routes

All HTTP methods are supported.

GET	/posts
GET	/posts/1
GET	/posts/1/comments
GET	/comments?postId=1
GET	/posts?userId=1
POST	/posts
PUT	/posts/1
PATCH	/posts/1
DELETE	/posts/1

Imagen 15. Invocar método DELETE.

**Nota:** Recuerden no se altera la base de datos de un api público, solo emulamos el método DELETE a través del HTTP expuesto).

1. Abrimos nuestro archivo CustomAdapter.kt, para agregar un método que maneje la eliminación de un objeto de la lista de posts, al final de la clase de la siguiente manera:

```
fun removeItem(position: Int) {  
    data.removeAt(position)  
    notifyItemRemoved(position)  
}
```

2. A continuación, creamos un método adicional en la misma clase “CustomAdapter.kt”, para agregar la funcionalidad de deshacer la eliminación del post, con el código siguiente al final de la clase:

```
fun restoreItem(item: Post, position: Int) {  
    data.add(position, item)  
    notifyItemInserted(position)  
}
```

3. Verificamos si tenemos la librería “com.android.support.design:28.0.0” instalada en nuestro archivo gradle de la app, de no ser así agregamos el siguiente código y sincronizamos:

```
implementation 'com.android.support.design:28.0.0'
```

4. Creamos una clase “UIHelper.kt” en el package `cl.desafiolatam.restapi.utils`, de ayuda para el trato del “swipe”, colores, icono de basura, en fin, gestos aplicados a la interfaz de usuario (UI), utilizando el paquete “`android.graphics`”. El código es el siguiente:

```
package cl.desafiolatam.restapi.utils

import android.content.Context
import android.graphics.Canvas
import android.graphics.Color
import android.graphics.Paint
import android.graphics.PorterDuff
import android.graphics.PorterDuffXfermode
import android.graphics.drawable.ColorDrawable
import android.graphics.drawable.Drawable
import android.support.v4.content.ContextCompat
import android.support.v7.widget.RecyclerView
import android.support.v7.widget.helper.ItemTouchHelper

abstract class UIHelper internal constructor(mContext: Context,
                                             private val mBackground: ColorDrawable = ColorDrawable(),
                                             private val backgroundColor: Int = Color.parseColor("#b80f0a"),
                                             private val mClearPaint: Paint = Paint()
) : ItemTouchHelper.Callback() {
    private val deleteDrawable: Drawable?
    private val intrinsicWidth: Int
    private val intrinsicHeight: Int

    init {
        mClearPaint.xfermode = PorterDuffXfermode(PorterDuff.Mode.CLEAR)
        deleteDrawable = ContextCompat.getDrawable(mContext, android.R.drawable.ic_menu_delete)
        intrinsicWidth = deleteDrawable!!.intrinsicWidth
        intrinsicHeight = deleteDrawable.intrinsicHeight
    }

    override fun getMovementFlags(recyclerView: RecyclerView, viewHolder: RecyclerView.ViewHolder):
    Int {
        return ItemTouchHelper.Callback.makeMovementFlags(0, ItemTouchHelper.LEFT)
    }

    override fun onMove(
        recyclerView: RecyclerView,
        viewHolder: RecyclerView.ViewHolder,
        viewHolder1: RecyclerView.ViewHolder
    ): Boolean {
        return false
    }

    override fun onChildDraw(
        c: Canvas,
        recyclerView: RecyclerView,
        viewHolder: RecyclerView.ViewHolder,
        dX: Float,
        dY: Float,
        actionState: Int,
        isCurrentlyActive: Boolean
    ) {
```

```

    ){
        super.onChildDraw(c, recyclerView, viewHolder, dX, dY, actionState, isCurrentlyActive)

        val itemView = viewHolder.itemView
        val itemHeight = itemView.height

        val isCancelled = dX == 0f && !isCurrentlyActive

        if (isCancelled) {
            clearCanvas(
                c,
                itemView.right + dX,
                itemView.top.toFloat(),
                itemView.right.toFloat(),
                itemView.bottom.toFloat()
            )
            super.onChildDraw(c, recyclerView, viewHolder, dX, dY, actionState, isCurrentlyActive)
            return
        }

        mBackground.color = backgroundColor
        mBackground.setBounds(itemView.right + dX.toInt(), itemView.top, itemView.right,
            itemView.bottom)
        mBackground.draw(c)

        val deleteIconTop = itemView.top + (itemHeight - intrinsicHeight) / 2
        val deleteIconMargin = (itemHeight - intrinsicHeight) / 2
        val deleteIconLeft = itemView.right - deleteIconMargin - intrinsicWidth
        val deleteIconRight = itemView.right - deleteIconMargin
        val deleteIconBottom = deleteIconTop + intrinsicHeight
        deleteDrawable!!.setBounds(deleteIconLeft, deleteIconTop, deleteIconRight, deleteIconBottom)
        deleteDrawable.draw(c)

        super.onChildDraw(c, recyclerView, viewHolder, dX, dY, actionState, isCurrentlyActive)
    }

    private fun clearCanvas(c: Canvas, left: Float?, top: Float?, right: Float?, bottom: Float?) {
        c.drawRect(left!!, top!!, right!!, bottom!!, mClearPaint)
    }

    override fun getSwipeThreshold(viewHolder: RecyclerView.ViewHolder): Float {
        return 0.7f
    }
}

```

5. Agregamos un nuevo método Rest, para invocar el método de la api de JsonPlaceHolder que elimina los posts, en nuestra interfaz "Api.kt", de la siguiente manera:

```
@DELETE("/posts/{postId}")
fun deletePost(@Path("postId") postId: Int?): Call<Void>
```

{postId} es un parámetro que recibe la anotación y se envía a través del método deletePost, donde @Path hace la conexión de la variable hacia la anotación, que en este caso es de tipo número (Int).

6. Con todos estos elementos creados, es hora de editar nuestra actividad principal "MainActivity.kt" para crear un método que llame la acción a través de retrofit para invocar el api rest con el método DELETE; para esto es necesario crear el siguiente método al final de la clase de actividad:

```
private fun deletePostFromApi(postId: Int?){
    val service = RetrofitClient.retrofitInstance()
    val call = service.deletePost(postId)
    call.enqueue(object : Callback<Void> {
        override fun onResponse(call: Call<Void>, response: Response<Void>) {

            if(response.code() == 200) {
                Toast.makeText(
                    applicationContext,
                    "Se ha eliminado correctamente el post en el api. Código respuesta de servidor: "+
                    response.code(), Toast.LENGTH_LONG).show()

                Toast.makeText(applicationContext, "Recuerda, nuestra api sólo facilita métodos http públicos
                para REST. Esta operación no elimina datos reales.",
                    Toast.LENGTH_LONG
                ).show()
            }
        }
    })

    override fun onFailure(call: Call<Void>, t: Throwable) {
        // handle failure
        Toast.makeText(applicationContext, "Ocurrió un error borrando el post en el api. "+t,
            Toast.LENGTH_LONG).show()
    }
}
```

**Nota:** Recordemos que esta api nos devuelve un código desde su servicio rest para pruebas, no existe una eliminación de los datos reales en la base de datos de esta página por el solo hecho de ser pública.

7. El método anterior será invocado desde un nuevo método que ejecutará las acciones de modificación de la UI con la clase creada anteriormente llamada "UIHelper.kt", este método llamado "enableSwipeToDeleteAndUndo()" cumplirá tres funciones:

- Ejecutar la eliminación de un post deslizando el dedo desde la derecha hacia la izquierda de la pantalla.
- Enviar la solicitud de eliminación tipo rest DELETE al servidor del api de JsonPlaceholder y procesar un código de respuesta, el cual se mostrará a través de un mensaje Toast en la pantalla.
- Mostrar un widget tipo Snackbar con la opción de deshacer la acción de borrar para restaurar el post que se ha eliminado. Esta barra deberá aparecer al final de la pantalla y durará entre 12 y 15 segundos, lo suficiente para permanecer luego de los mensajes Toast.

Al final de nuestra MainActivity.kt, creamos el nuevo método y su código es el siguiente:

```
private fun enableSwipeToDeleteAndUndo() {  
    val swipeToDeleteCallback = object : UIHelper(this) {  
        override fun onSwiped(viewHolder: RecyclerView.ViewHolder, i: Int) {  
  
            val position = viewHolder.adapterPosition  
            val item = mAdapter.data.get(position)  
  
            //ELIMINAMOS DEL SERVIDOR EL POST (FAKE)  
            deletePostFromApi(item.id)  
            mAdapter.removeItem(position)  
  
            val snackbar = Snackbar  
                .make(coordinatorLayout, "El post ha sido eliminado de la lista", Snackbar.LENGTH_LONG)  
            snackbar.setAction("DESHACER") {  
                mAdapter.restoreItem(item, position)  
                recyclerView.scrollToPosition(position)  
            }  
  
            snackbar.setActionTextColor(Color.YELLOW)  
            snackbar.setDuration(12000)  
            snackbar.show()  
  
        }  
    }  
  
    val itemTouchhelper = ItemTouchHelper(swipeToDeleteCallback)  
    itemTouchhelper.attachToRecyclerView(recyclerView)  
}
```

8. Por último, agregamos al método ya existente “loadApiData()” de la misma clase de actividad la instancia al método enableSwipeToDeleteAndUndo(), dentro de su sub método onResponse(), justo después de la asignación del adaptador a la variable recyclerView, como se muestra a continuación:

```
if (postsFromApi != null) {  
    mAdapter = CustomAdapter(postsFromApi)  
    recyclerView.adapter = mAdapter  
  
    enableSwipeToDeleteAndUndo()  
}
```

Esto permitirá la funcionalidad de eliminar los posts en la pantalla para cada uno de los elementos de la lista cargados desde el api JsonPlaceHolder con el método GET posts.

9. Ejecutamos nuestra aplicación y la funcionalidad se visualiza como lo muestran las siguientes imágenes:

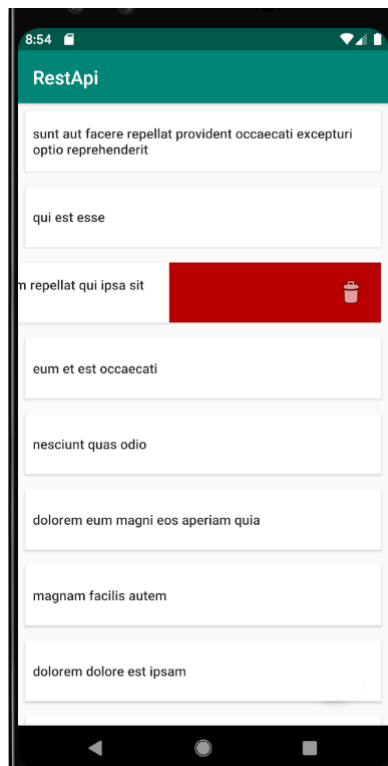


Imagen 16. Acción Swipe To Delete.

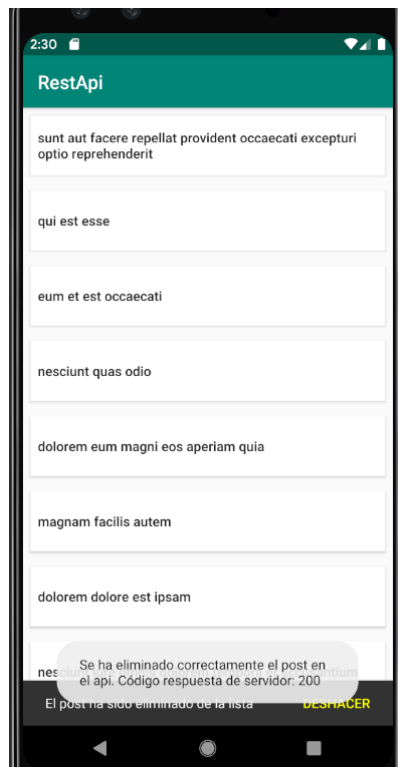


Imagen 17. Mensaje respuesta del api a través del servicio rest /posts/{id} , método http DELETE, y barra de acción “deshacer” para recuperar el post eliminado en la UI.

## Volley HTTP Client

Volley es una biblioteca HTTP-Client al igual que Retrofit, que hace que la creación de redes para aplicaciones de Android sea más fácil y, lo que es más importante, más rápida.

Volley está disponible en [GitHub](https://github.com/google/volley). Volley ofrece los siguientes beneficios:

- Programación automática de solicitudes de red.
- Múltiples conexiones de red concurrentes.
- Disco transparente y memoria caché de respuesta con [coherencia de caché](#) HTTP estándar .
- Soporte para solicitud de priorización.
- Solicitud de cancelación API. Puede cancelar una sola solicitud o puede establecer bloques o ámbitos de solicitudes para cancelar.
- Facilidad de personalización, por ejemplo, para reintentos y retrocesos.
- Ordenación sólida que facilita el llenado correcto de la interfaz de usuario con los datos obtenidos de forma asincrónica de la red.
- Herramientas de depuración y rastreo.

Volley se destaca en las operaciones de tipo RPC que se utilizan para completar una interfaz de usuario, como buscar una página de resultados de búsqueda como datos estructurados.

Se integra fácilmente con cualquier protocolo y viene de fábrica con soporte para cadenas sin procesar, imágenes y JSON. Al proporcionar soporte integrado para las funciones que necesita, Volley lo libera de escribir código repetitivo y le permite concentrarse en la lógica específica de su aplicación.



Volley no es adecuado para grandes operaciones de descarga o transmisión, ya que Volley guarda todas las respuestas en la memoria durante el análisis. Para operaciones de descarga grandes, considere usar una alternativa como [DownloadManager](#).

La forma más fácil de agregar Volley a su proyecto es agregar la siguiente dependencia al archivo build.gradle de su aplicación:

```
dependencies { ... implementation 'com.android.volley: volley: 1.1.1' }
```

También puede clonar el repositorio de Volley y configurarlo como un proyecto de biblioteca. Git clone el repositorio escribiendo lo siguiente en la línea de comando:

```
git clone https://github.com/google/volley
```

Un ejemplo para devolver una respuesta con Volley como cadena string y asignarla a un textview en una actividad sería el siguiente:

```
// function for network call
fun getUsers() {
    // Instantiate the RequestQueue.
    val queue = Volley.newRequestQueue(this)
    val url: String = "https://api.github.com/search/users?q=eyehunt"

    // Request a string response from the provided URL.
    val stringReq = StringRequest(Request.Method.GET, url,
        Response.Listener<String> { response ->

            var strResp = response.toString()
            val jsonObj: JSONObject = JSONObject(strResp)
            val jsonArray: JSONArray = jsonObj.getJSONArray("items")
            var str_user: String = ""
            for (i in 0 until jsonArray.length()) {
                val jsonInner: JSONObject = jsonArray.getJSONObject(i)
                str_user = str_user + "\n" + jsonInner.get("login")
            }
            textView!!.text = "response : $str_user "
        },
        Response.ErrorListener { textView!!.text = "That didn't work!" })
    queue.add(stringReq)
}
```

Importando las siguientes clases a la actividad android:

```
import com.android.volley.Request  
import com.android.volley.Response  
import com.android.volley.toolbox.StringRequest  
import com.android.volley.toolbox.Volley
```

# Intercambio de datos con JSON

---

## Competencias:

- Conocer el formato de intercambio de datos JSON.
- Comprender la estructura que conforma un JSON.
- Identificar la cabecera de una consulta a una api rest.
- Identificar el cuerpo de una consulta a una api rest.

## Introducción

En este capítulo estudiaremos los conceptos asociados a JSON (Javascript Object Notation) como la notación más ligera, eficiente y rápida de toda la web en la actualidad.

JSON ha ido desplazando desde hace algunos años las antiguas formas de intercambio de datos que por ejemplo se hacían utilizando XML, llegando a tal punto que las bases de datos modernas han comenzado a evolucionar hacia el almacenamiento de datos JSON en su forma compilada BSON, como lo son por ejemplo, MongoDB y Oracle 12c+.

Actualmente el estándar de comunicación es a través de Json, Comprender la sintaxis y cómo se ordenan los datos en este formato será de vital importancia para nuestro desarrollador como creadores de apps.

## JSON

JSON (JavaScript Object Notation) es un formato ligero de intercambio de datos. Es fácil para los humanos leer y escribir. Es fácil para las máquinas analizar y generar. Se basa en un subconjunto del lenguaje de programación denominados JavaScript Standard ECMA-262.

JSON es un formato de texto que es completamente independiente del lenguaje pero utiliza convenciones que son familiares para los programadores de la familia de lenguajes C, incluido Kotlin, C, C ++, C#, Java, JavaScript, Perl, Python y muchos otros. Estas propiedades hacen que JSON sea un lenguaje ideal para el intercambio de datos.

## Estructura de un JSON

JSON se basa en dos estructuras:

- Una colección de pares de nombre / valor, en distintos idiomas y lenguajes de programación; esto es conocido como un objeto, registro, estructura, diccionario, tabla hash, lista con clave o arreglo asociativo.
- Una lista ordenada de valores. En la mayoría de los lenguajes de programación, esto se implementa como arreglos, vectores, listas o secuencias.

Estas son estructuras universales; virtualmente todos los lenguajes de programación las soportan de una forma u otra. Es razonable que un formato de intercambio de datos que es independiente del lenguaje de programación se base en estas estructuras.

En JSON, se presentan de estas formas:

Un objeto es un conjunto desordenado de pares de nombre / valor. Un objeto comienza con {llave izquierda y termina con} llave derecha. Cada nombre es seguido por: dos puntos y los pares de nombre / valor están separados por una coma.

### Ejemplo:

```
{
  "glossary": {
    "title": "example glossary",
    "GlossDiv": {
      "title": "S",
      "GlossList": {
        "GlossEntry": {
          "ID": "SGML",
          "SortAs": "SGML",
          "GlossTerm": "Standard Generalized Markup Language",
          "Acronym": "SGML",
          "Abbrev": "ISO 8879:1986",
          "GlossDef": {
            "para": "A meta-markup language, used to create markup languages such as DocBook.",
            "GlossSeeAlso": ["GML", "XML"]
          },
          "GlossSee": "markup"
        }
      }
    }
  }
}
```

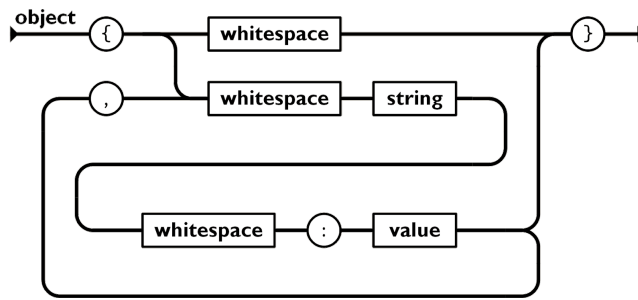


Imagen 18. Representación gráfica de la estructura de un arreglo JSON.

### Ejemplo de array json:

```
[
  { "id": 1, "nombre": "Luis Liberal" },
  { "id": 2, "nombre": "Pamela Martín" }
]
```

Un valor puede ser una cadena entre comillas dobles, o un número, o verdadero o falso o nulo, o un objeto o una matriz. Estas estructuras pueden estar anidadas.

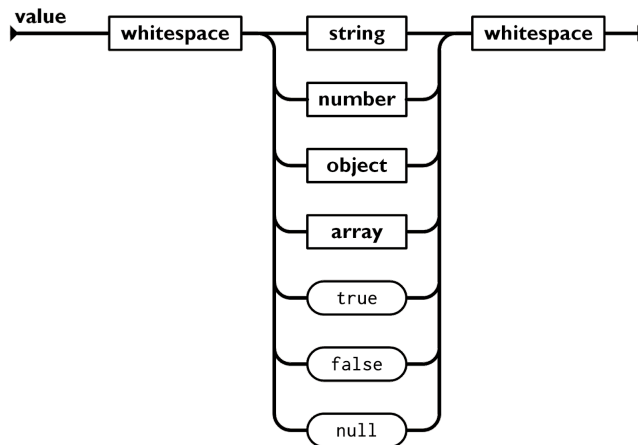


Imagen 19. Representación gráfica de la estructura que tienen los valores en los objetos json.

Una cadena es una secuencia de cero o más caracteres Unicode, entre comillas dobles, usando escapes de barra invertida. Un carácter se representa como una cadena de caracteres única. Una cadena es muy parecida a una cadena C o Java.

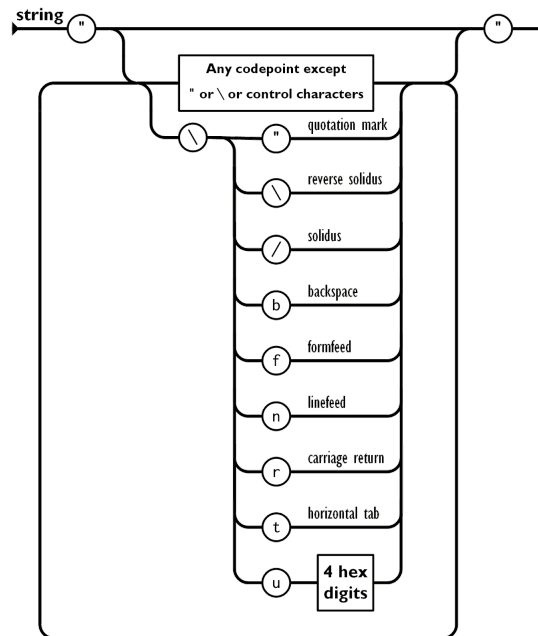


Imagen 20. Representación gráfica de secuencia de tipos de valores en un String o cadena.

Un número es muy parecido a un número C o Java, excepto que los formatos octal y hexadecimal no se usan.

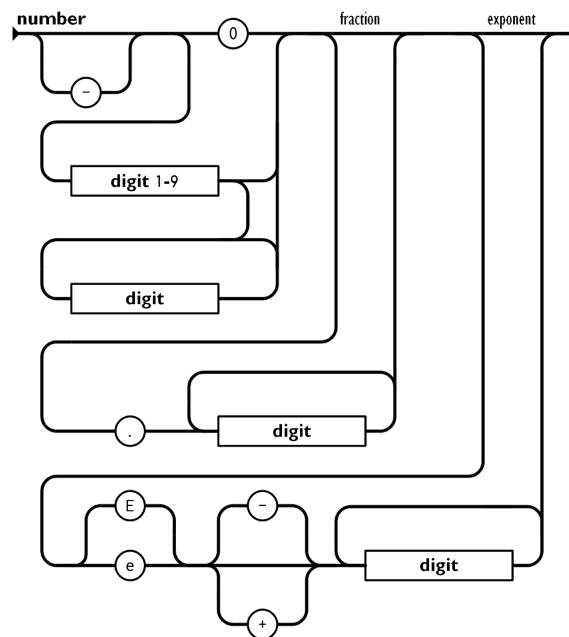


Imagen 21. Representación gráfica tipos de valores numéricos en JSON.

Se puede insertar un espacio en blanco entre cualquier par de tokens. Exceptuando algunos detalles de codificación, que describe completamente el lenguaje.

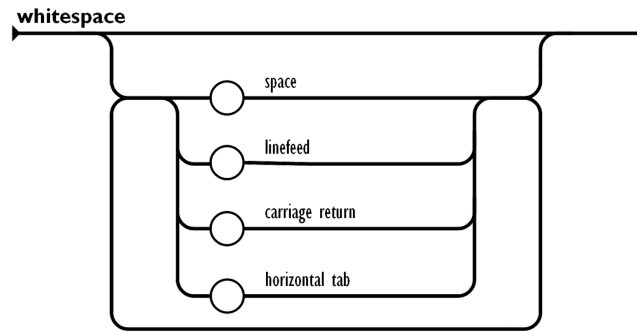


Imagen 22. Representación gráfica del trato de espacios en blanco en JSON.

## JSON - REST HEADER (Cabecera/Encabezado)

Los headers y parámetros REST contienen una gran cantidad de información que puede ayudarlo a localizar problemas cuando los encuentre. Los encabezados HTTP son una parte importante de la solicitud y respuesta de la API, ya que representan los metadatos asociados con la solicitud y la respuesta de la API. Los encabezados llevan información para:

- Request and Response Body (Cuerpos de datos en solicitudes y respuestas)
- Request Authorization (Solicitar autorización)
- Caché de respuesta
- Cookies de respuesta

Además de las categorías anteriores, los encabezados HTTP también contienen otras informaciones sobre los tipos de conexión HTTP, servidores proxy, etc. La mayoría de estos encabezados son para la administración de conexiones entre cliente, servidor y servidores proxy; no requieren validación explícita a través de las pruebas.

Los encabezados se clasifican principalmente como encabezados de solicitud y encabezados de respuesta. Los encabezados de solicitud se configuran cuando enviamos la solicitud para probar una API y debemos establecer la aserción en los encabezados de respuesta para garantizar que se devuelvan los encabezados correctos.

Los encabezados con los que nos encontraremos más durante el trabajo con un API son los siguientes:

- **Authorization:** lleva credenciales que contienen la información de autenticación del cliente para el recurso que se solicita.
- **WWW-Authenticate:** el servidor lo envía si necesita una forma de autenticación antes de que pueda responder con el recurso real que se solicita. A menudo se envía junto con un código de respuesta de 401, que significa "no autorizado".
- **Accept-Charset:** este es un encabezado que se establece con la solicitud y le dice al servidor qué conjuntos de caracteres son aceptables para el cliente.
- **Content-Type:** indica el tipo de medio (texto / html o texto / JSON) de la respuesta enviada al cliente por el servidor, esto ayudará al cliente a procesar el cuerpo de la respuesta correctamente.
- **Cache-Control:** esta es la política de caché definida por el servidor para esta respuesta, el cliente puede almacenar una respuesta en caché y volver a usar hasta el momento definido por el encabezado de Control de caché.

## RETROFIT - JSON HEADERS

En retrofit para asignar parámetros a nuestros métodos http de las interfaces, basta con agregar la anotación `@Headers` sobre la anotación del tipo de método http, ya sea get, post, put, delete.

**Ejemplo:**

```
@Headers("Content-Type: application/json; charset=UTF-8")
```

## JSON - REST BODY (Cuerpo)

El cuerpo de un objeto json, no requiere mayor explicación de la que ya hemos dado, el cuerpo json (body) es ese objeto json que puede anexarse a una solicitud (request) o recibirse en la respuesta del servicio. Un ejemplo de un cuerpo json sería el siguiente:

```
{
  "customers":
  {
    "firstName": "Joe",
    "lastName": "Bloggs",
    "fullAddress":
    {
      "streetAddress": "21 2nd Street",
      "city": "New York",
      "state": "NY",
      "postalCode": 10021
    }
  }
}
```



## RETROFIT - JSON BODY

En kotlin utilizando retrofit podemos disponer de una anotación `@Body` que se utiliza para asignar un objeto tipo pojo (clase que contiene instancias de los datos similares a los que contiene el objeto json) a una respuesta o solicitud. Por ejemplo, anteponiendo los siguientes ejercicios, así sería una asignación de un método Post en la interfaz de retrofit para enviar un nuevo post al servidor:

```
@Headers("Content-Type: application/json; charset=UTF-8")
@POST("/posts")
fun createNewPost(@Body post: Post): Call<Post>
```

Dónde "Post" es nuestra clase pojo que contiene los objetos del json body.

### Ejercicio 6:

Crear un post nuevo usando Postman con los datos indicados en la página del api de JsonPlaceholder, enviando cabeceras y cuerpos (headers, body) como se indican en su guía: <https://jsonplaceholder.typicode.com/guide.html>

1. Abrir Postman y asignar la url del método POST para la creación de un nuevo post.

```
https://jsonplaceholder.typicode.com/posts
```

2. Asignar el cuerpo como se indica en la guía de la siguiente manera:



Imagen 23. Asignar el cuerpo.

3. Asignar los parámetros de cabecera Content-Type y Charset de la siguiente forma:

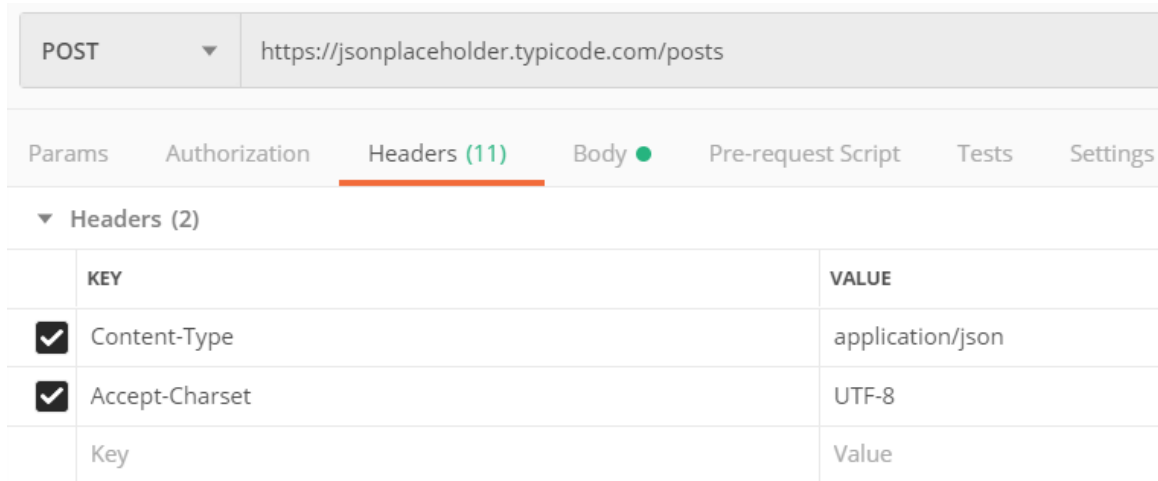


Imagen 24. Asignar parámetros.

4. Visualizar la respuesta con código 201 del servidor, donde se indica su aceptación y retorna un id de post nuevo (estático ya que no lo crea realmente en base de datos por ser pública).

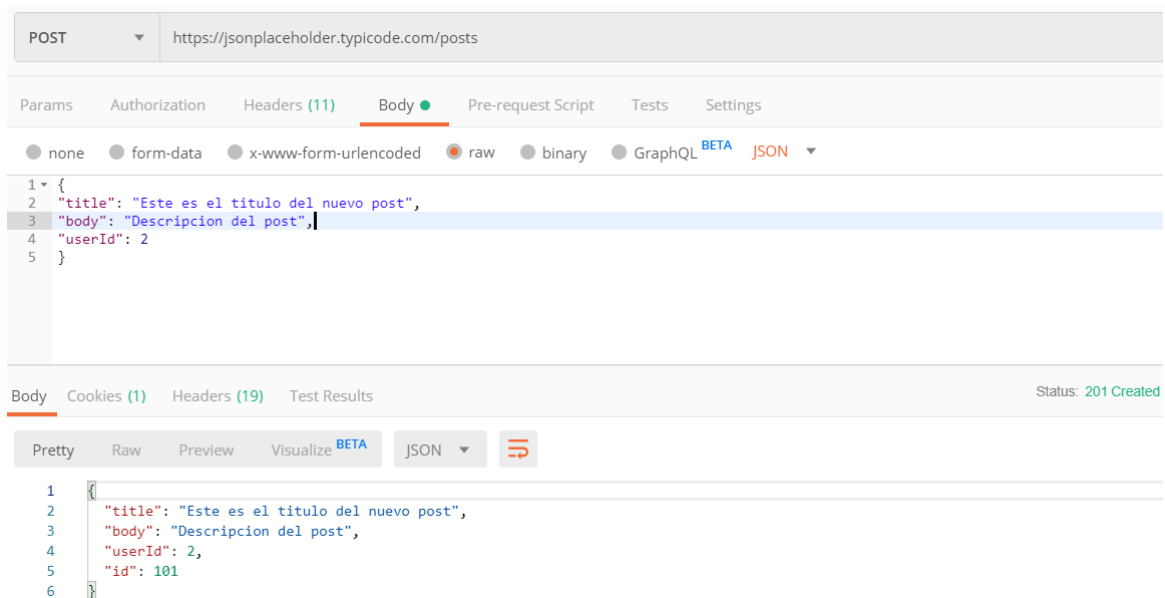


Imagen 25. Visualizar respuesta.