

# Internet Connection

---

1TEL05 - Servicios y Aplicaciones para IoT

# Permisos

---

# Permisos → Manifest

- Cuando una aplicación desea acceder a características del sistema Android como: la cámara, sms o consultas a internet; es necesario agregar permisos adicionales. Estos son registrados en el **AndroidManifest.xml**

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.snazzyapp">

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

    <application ...>
        ...
    </application>
</manifest>
```

- Para que pueda hacer solicitudes a internet, debe solicitar el permiso:

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

- Y para verificar el estado de la red:

→ `<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />`

★ **Nota:** Tanto el permiso Internet como ACCESS\_NETWORK\_STATE son permisos normales, lo que significa que se otorgan al momento de la instalación y no requieren solicitarse en el tiempo de ejecución.

# Verificar el estado de la conexión

---

# Estado de la conexión

Es importante ***verificar el estado de la conexión (o si existe alguna)*** antes de hacer alguna solicitud a un servidor externo. Para este fin puede utilizar la clase:



```
ConnectivityManager connectivityManager =  
    (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);  
  
boolean tieneInternet = false;  
if (connectivityManager != null) {  
    NetworkCapabilities capabilities =  
        connectivityManager.getNetworkCapabilities(connectivityManager.getActiveNetwork());  
    if (capabilities != null) {  
        if (capabilities.hasTransport(NetworkCapabilities.TRANSPORT_CELLULAR)) {  
            Log.i( tag: "msg-Internet", msg: "NetworkCapabilities.TRANSPORT_CELLULAR");  
            tieneInternet = true;  
        } else if (capabilities.hasTransport(NetworkCapabilities.TRANSPORT_WIFI)) {  
            Log.i( tag: "msg-Internet", msg: "NetworkCapabilities.TRANSPORT_WIFI");  
            tieneInternet = true;  
        } else if (capabilities.hasTransport(NetworkCapabilities.TRANSPORT_ETHERNET)) {  
            Log.i( tag: "msg-Internet", msg: "NetworkCapabilities.TRANSPORT_ETHERNET");  
            tieneInternet = true;  
        }  
    }  
}
```

# Tipo de conexión

Es posible **validar el tipo de conexión**, si tiene conexión wifi, red celular, ethernet, bluetooth entre otras.

```
ConnectivityManager connMgr = (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);
boolean isWifiConn = false;
boolean isMobileConn = false;
boolean isEthernetConn = false;
for (Network network : connMgr.getAllNetworks()) {
    NetworkInfo networkInfo = connMgr.getNetworkInfo(network);
    if (networkInfo.getType() == ConnectivityManager.TYPE_WIFI) {
        isWifiConn = networkInfo.isConnected();
    }
    if (networkInfo.getType() == ConnectivityManager.TYPE_MOBILE) {
        isMobileConn = networkInfo.isConnected();
    }
    if (networkInfo.getType() == ConnectivityManager.TYPE_ETHERNET) {
        isEthernetConn = networkInfo.isConnected();
    }
}
Log.d( tag: "msg", msg: "Wifi connected: " + isWifiConn);
Log.d( tag: "msg", msg: "Mobile connected: " + isMobileConn);
Log.d( tag: "msg", msg: "Ethernet connected: " + isEthernetConn);
```

# Envío y obtención de información

---

# Envío y obtención de información

Para enviar y recibir información desde Android existen **dos formas**:

- Clase **URLConnection**, la cual permite realizar una solicitud externa.
- Usar **librerías de 3ros**.

**URLConnection** brinda mayor flexibilidad; sin embargo, se le considera una librería low-level pues debe gestionar desde la creación del URL (en formato URI) hasta el parseo de los datos manualmente. Así mismo, debe asegurarse que las solicitudes corran en un worker thread. Más información puede revisar:

<https://developer.android.com/reference/java/net/URLConnection.html>

→ Por este motivo, se utilizarán librerías de 3ros para implementar la funcionalidad.



# Librerías de terceros

Existen **3 librerías** más usadas para obtener información puntual de internet:

- [Volley](#)
- [OkHttp](#)
- [Retrofit](#)

Cada una tiene sus ventajas y sus aplicaciones particulares.

Mostraremos cómo obtener y mandar información de un webservice usando **Retrofit 2**

Si planea descargar o enviar gran cantidad de información > 200MB, Android recomienda usar [DownloadManager](#).

# Retrofit 2

- Librería que permite consumir webservices sobre HTTP en Android y Java.
- Facilita la conexión mediante el uso de interfaces Java.
- No incluye convertidor JSON para analizar la información recibida siendo necesario incluir librería para parsear como:
  - Gson,
  - Jackson o
  - Moshi.

Usaremos Gson: <https://github.com/google/gson/blob/master/UserGuide.md>

→ Dependencias para utilizar Retrofit, se debe agregar al Gradle:

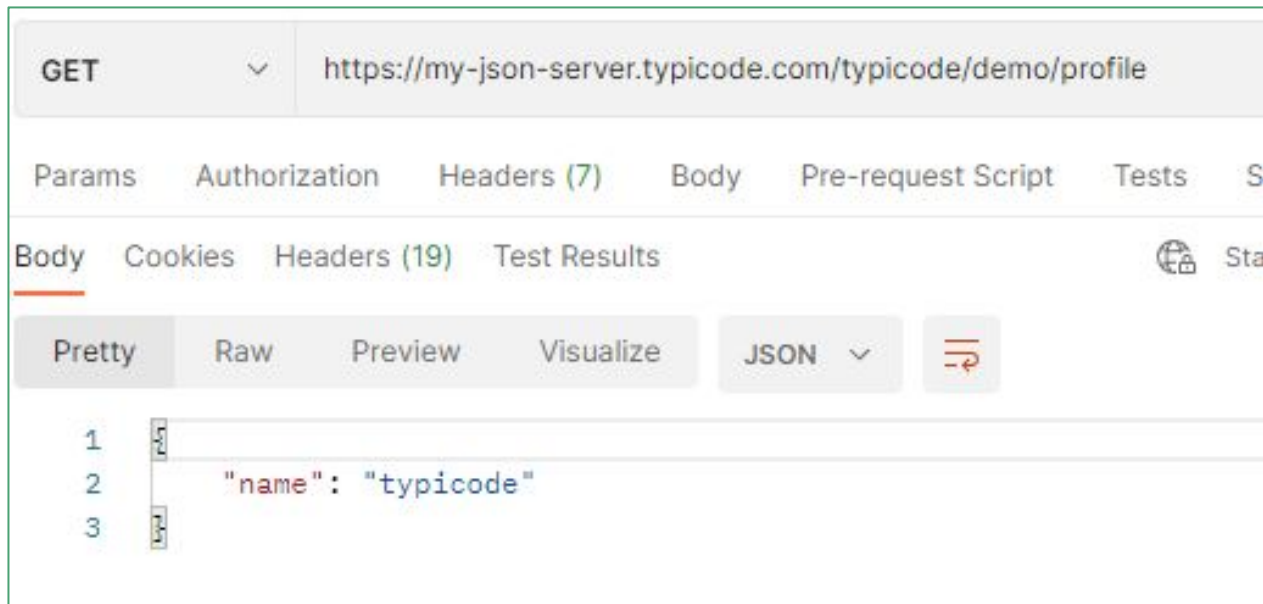
```
implementation 'com.squareup.retrofit2:retrofit:2.9.0'  
implementation 'com.google.code.gson:gson:2.10'  
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
```

GET

# Prueba 1 con Postman

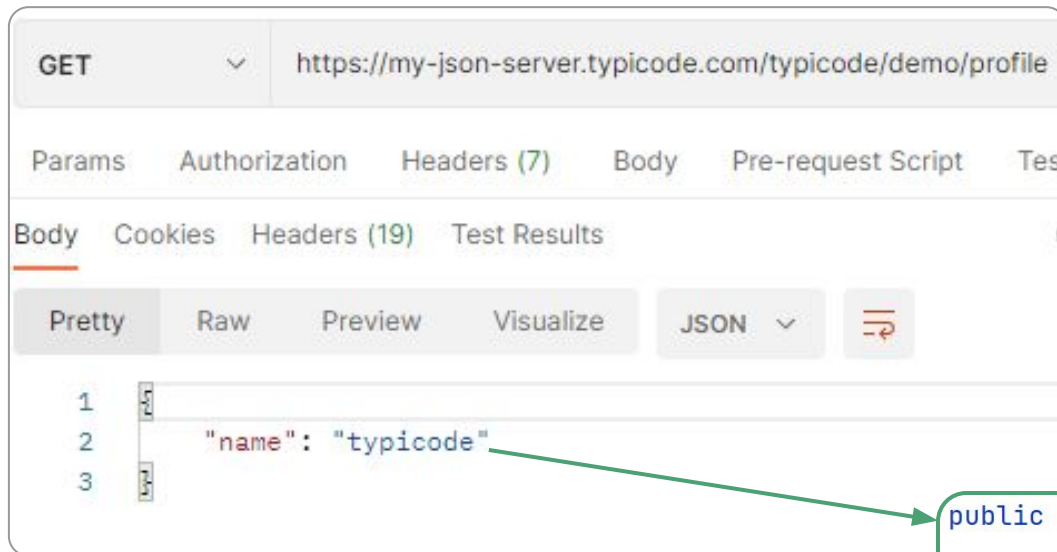
La URL indicada brinda un JSON de prueba:

```
https://my-json-server.typicode.com/typicode/demo/profile
```



# Bean

Para que Retrofit con Gson convierta la respuesta de un webservice en un objeto en Java (POJO), se debe crear una **clase que describa la respuesta**.



```
public class Profile {

    private String name;

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

}
```

# Creación de interfaz

- En la interfaz se definen métodos donde le corresponde la porción de URL path.

`https://my-json-server.typicode.com/typicode/demo/profile`

baseUrl

URL path

- Se utilizan anotaciones **@GET** o **@POST** para indicar el método HTTP
- Siempre **devuelve un objeto Call**, conteniendo el tipo de dato que se desea mapear, en este caso, un Profile (ver diapositiva anterior).

```
public interface TypicodeService {  
    @GET("/typicode/demo/profile")  
    Call<Profile> getProfile();  
}
```

# Utilizando Retrofit

Se crea una instancia del Servicio definido en la interfaz, donde se le indica:

- la baseUrl,
- el conversor json y
- la interfaz a instanciar.

```
TypicodeService typicodeService = new Retrofit.Builder()
    .baseUrl("https://my-json-server.typicode.com")
    .addConverterFactory(GsonConverterFactory.create())
    .build()
    .create(TypicodeService.class);
```

# Llamando al método

- Se invoca al método de la interfaz
- Dos formas de ejecución:
  - **enqueue(callback)** → asíncrono (maneja sus propios hilos).
  - **execute** → síncrono (debe ser manejado en un background thread).
- En ambos casos, la respuesta se obtiene con **response.body()**

```
typicodeService.getProfile().enqueue(new Callback<Profile>() {  
    @Override  
    public void onResponse(Call<Profile> call, Response<Profile> response) {  
        //aca estoy en el UI Thread  
        if(response.isSuccessful()){  
            Profile profile = response.body();  
            binding.rptaTextView.setText(profile.getName());  
            Log.d( tag: "msg-test-ws-profile", msg: "name: " + profile.getName());  
        } else{  
            Log.d( tag: "msg-test-ws-profile", msg: "error en la respuesta del webservice");  
        }  
    }  
}  
  
@Override  
public void onFailure(Call<Profile> call, Throwable t) { t.printStackTrace(); }  
});
```



## Ejemplo 2

Usar la URL: <https://my-json-server.typicode.com/typicode/demo/comments>

Y mostrar los resultados:




A screenshot of a web browser window. The address bar shows the URL `my-json-server.typicode.com/typicode/demo/comments`. The main content area displays a JSON array of two comment objects. The first object has `"id": 1`, `"body": "some comment"`, and `"postId": 1`. The second object has `"id": 2`, `"body": "some comment"`, and `"postId": 1`.

```
[
  {
    "id": 1,
    "body": "some comment",
    "postId": 1
  },
  {
    "id": 2,
    "body": "some comment",
    "postId": 1
  }
]
```

# Nuevo bean y método

Se crea un **Bean** que representa la respuesta y un nuevo método en la interfaz.

```
[
  {
    "id": 1,
    "body": "some comment",
    "postId": 1
  },
  {
    "id": 2,
    "body": "some comment",
    "postId": 1
  }
]
```



```
public class Comment {

    private int id;
    private String body;
    private int postId;
}
```

```
public interface TypicodeService {

    @GET("/typicode/demo/profile")
    Call<Profile> getProfile();

    @GET("/typicode/demo/comments")
    Call<List<Comment>> getComments();
}
```

# Utilizar el método

```
typicodeService.getComments().enqueue(new Callback<List<Comment>>() {  
    @Override  
    public void onResponse(Call<List<Comment>> call, Response<List<Comment>> response) {  
        if(response.isSuccessful()){  
            List<Comment> comments = response.body();  
            for(Comment c : comments){  
                Log.d( tag: "msg-test-ws-comments", msg: "id: "  
                    + c.getId() + " | body: " + c.getBody());  
            }  
        } else {  
            Log.d( tag: "msg-test", msg: "error en la respuesta del webservice");  
        }  
    }  
}  
  
@Override  
public void onFailure(Call<List<Comment>> call, Throwable t) {  
    t.printStackTrace();  
}  
});
```

# Parámetros variables por GET

Para enviar parámetros variables por **GET**, utilizar **@Query**

URL/**getChartValues**?**hospitalId**=X&**date**=Y&**patientId**=Z

```
@GET("/getChartValues")  
Call<Metrics> getMetrics(@Query("hospitalId") String hospitalId,  
                        @Query("date") String date,  
                        @Query("patientId") String patientId);
```

Parámetros  
@SerializedName

# @SerializedName

Si tiene campos en su JSON que no pueden crearse en su clase (comienzan con número o contienen espacios) puede utilizar la notación `@SerializedName`

```
public class Person {  
  
    @SerializedName("name")  
    private String personName;  
  
    @SerializedName("bd")  
    private String birthDate;  
  
}
```

El JSON recibido en este ejemplo, tiene la forma de:

```
{  
    name: "juan"  
    bd: "20/03/1996"  
}
```

POST

# Enviar información POST - Ejemplo 1

Otra opción es enviar información de su dispositivo a un servidor. Esto lo puede realizar con GET (enviando los parámetros en la URL) o con **POST**, enviando los parámetros como parte del cuerpo HTTP. Usaremos este segundo enfoque.

Webservice de prueba:

## Descripción

Validar si el nombre de usuario existe o no existe

Usuarios pre-existentes → carlos.davila / manuel.sanchez / andrea.carrasco / carla.carrillo

## Request

ENDPOINT	https://1a8jit3xd4.execute-api.us-east-1.amazonaws.com/prod				
MÉTODO	POST				
PARÁMETROS	Tipo	Nombre	Descripción	Tipo de Dato	Ejemplo
	GET	accion (obligatorio)	acción a realizar, el valor debe ser "validar"	String	"validar"
	POST	username (obligatorio)	nombre de usuario a validar	String	"carlos.davila"

## Response

RESPUESTA OK 1 (usuario existe)	HEADER	HTTP/ 200 OK
	CONTENT-TYPE	application/json
	BODY	existe
RESPUESTA OK 2 (usuario no existe)	HEADER	HTTP/ 200 OK
	CONTENT-TYPE	application/json
	BODY	no existe



# Una nueva interfaz

Debido a que el nuevo webservice no comparte la misma baseURL que los dos anteriores, es necesario crear una nueva interfaz. Así mismo, se utiliza el método **@FormUrlEncoded** para mandar con el formato **x-www-form-urlencoded** y **@Field** para cada campo enviado por POST.

```
public interface AwsService {  
  
    @FormUrlEncoded  
    @POST("/prod?accion=validar")  
    Call<String> existeUser(@Field("username") String username);  
}
```

# Utilizando la interfaz

```
AwsService awsService = new Retrofit.Builder()
    .baseUrl("https://1a8jit3xd4.execute-api.us-east-1.amazonaws.com")
    .addConverterFactory(GsonConverterFactory.create())
    .build()
    .create(AwsService.class);

awsService.existeUser("carlos.davila").enqueue(new Callback<String>() {
    @Override
    public void onResponse(Call<String> call, Response<String> response) {
        String body = response.body();
        Log.d("msg-test", "aws: " + body);
    }

    @Override
    public void onFailure(Call<String> call, Throwable t) {
        t.printStackTrace();
    }
});
```


# Observaciones

# Tráfico HTTP

Si desea consumir **webservices** usando el protocolo **http**, debe incluir en su **Manifest.xml**:

→ `android:usesCleartextTraffic="true"`

```
<application
    android:name=".AppThreads"
    android:allowBackup="true"
    android:dataExtractionRules="@xml/data_extraction_rules"
    android:fullBackupContent="@xml/backup_rules"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:supportsRtl="true"
    android:theme="@style/Theme.Clase4"
    android:usesCleartextTraffic="true"
    tools:targetApi="31">
```



# Emulator - consume localhost

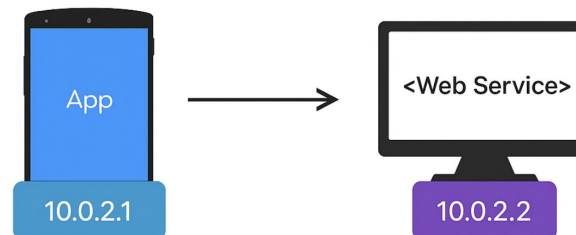
Para acceder a un servicio web desplegado en su computadora (localhost) desde el emulador, se debe tener en cuenta que:

- No se puede usar localhost o 127.0.0.1, ya que desde el emulador esta dirección hace referencia al propio emulador, no a la PC.
- Tampoco se puede usar la dirección IP local de tu computadora (como 192.168.x.x), ya que el emulador se encuentra en una red virtual diferente.

El emulador y la PC crean una red local interna, donde:

- El emulador obtiene el IP: **10.0.2.1**
- La PC obtiene el IP: **10.0.2.2**

Para consumir un webservice localmente, debe usar el segundo IP.



¿Preguntas?

---

Muchas gracias

---