

HLC

2º DAM

**I.E.S. POLITÉCNICO H. LANZ
JOSÉ MARÍA MOLINA**



TEMA 4 – APIREST DESDE JAVAFX: RETROFIT

TEMA 4 - APIREST DESDE JAVAFX



Vamos a crear un proyecto JAVAFX para explotar la **APIRest** hecha en temas anteriores.

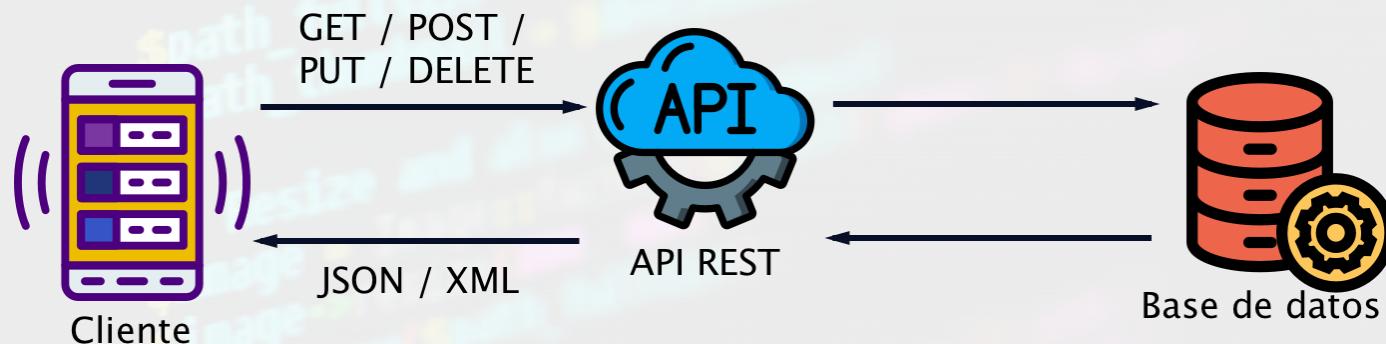
La tendencia es “publicar” BBDD vía una API para que podamos interactuar con los datos, por tanto también vamos a explotar APIRest públicas

- ✓ 1 APIREST Y ENDPOINTS
- ✓ 2 RETROFIT
- ✓ 3 APIs Externas (API Keys)

1 API REST Y ENDPOINTS



- ✓ **API Rest:** API es una Interfaz de Programación de Aplicaciones, es decir, una forma de que otras aplicaciones interactúen con los datos de la primera. **Rest** significa que es una arquitectura que se ejecuta sobre HTTP/HTTPS.
- ✓ Formas de probar una API:
 - Conjunto de Formularios (**DAW**)
 - APP hecha con Móvil(Android/IOS) o **APP de Escritorio (RETROFIT)**
 - POSTMAN ✓



1 API REST Y ENDPOINTS



- ✓ **API Rest:** Recordamos que nuestra API de ejemplo (Actores) gestionaba (CRUD) estos datos:

id	int(11)
nombre	varchar(100)
edad	int(3)
activo	tinyint(1)
fotourl	varchar(100)
fotocodif	varchar(10000)

Lo bueno de una API es que se puede reutilizar total o parcialmente según nos interese. Aquí utilizaremos un campo que NO se ha usado antes y desecharemos uno que se usa para WEB

- ✓ Fotourl: **SOLO** se utilizaba este campo para “visualizar” fotos subidas al servidor vía multipart (Envío de archivo), por lo que aquí **NO LO USAREMOS!**
- ✓ Fotocodif: Se usará para gestionar imágenes en Base64 (no usamo BLOB porque B64 es mucho más versátil: web/escrit/móvil)

1 APIREST Y ENDPOINTS

php

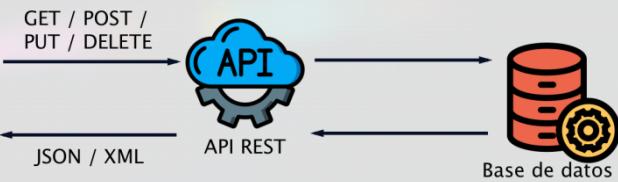
WEB

INSERTAR

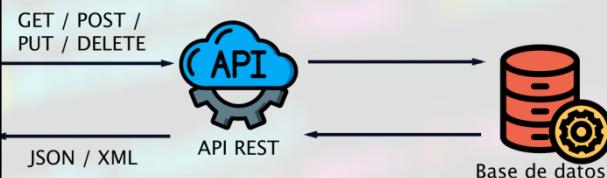
ID	Nombre	Edad	Activo	Foto URL	Foto Base64	Acciones
92	Tom Hanks	65	Sí			Editar Subir foto Borrar
93	Al Pacino	81	Sí			Editar Subir foto Borrar
94	Brad Pitt	58	Sí			Editar Subir foto Borrar
95	Tim Robbins	63	Sí			Editar Subir foto Borrar
96	Morgan Freeman	84	Sí			Editar Subir foto Borrar
97	Marlon Brando	80	No			Editar Subir foto Borrar
99	Robert De Niro	78	Sí			Editar Subir foto Borrar
101	Leonardo DiCaprio	47	Sí			Editar Subir foto Borrar
102	Keanu Reeves	57	Sí			Editar Subir foto Borrar
103	Matt Damon	51	Sí			Editar Subir foto Borrar
104	Anne Hathaway	39	Sí			Editar Subir foto Borrar
105	Cate Blanchett	52	Sí			Editar Subir foto Borrar
106	Natalie Portman	40	Sí			Editar Subir foto Borrar
107	Michelle Pfeiffer	63	Sí			Editar Subir foto Borrar
108	Julia Roberts	55	Sí			Editar Subir foto Borrar

localhost/APIRest_BBDDActoresV8/crud/leer.php

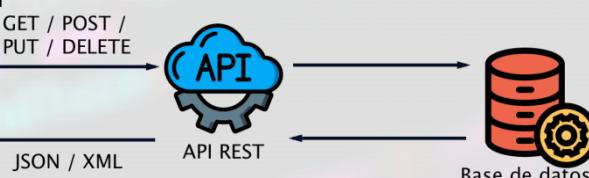
```
JSON Datos sin procesar Cabeceras  
Guardar Copiar Contraseña Expander todo Filtrar JSON  
▼ 8:  
  id: 92  
  nombre: "Tom Hanks"  
  edad: 65  
  activo: 1  
  fotourl: ""  
  fotocodif: ""  
  
  ▼ 9:  
  id: 93  
  nombre: "Al Pacino"  
  edad: 81  
  activo: 1  
  fotourl: ""  
  fotocodif: ""  
  
  ▼ 10:  
  id: 94  
  nombre: "Brad Pitt"  
  edad: 58  
  activo: 1  
  fotourl: ""  
  fotocodif: ""
```



ESCRITORIO



PMDM



1 API REST Y ENDPOINTS



✓ Un **ENDPOINT** (o punto final) es una **URL específica** que recibe solicitudes para interactuar con una API.

✓ En nuestro caso, un ejemplo de endpoint sería:

http://localhost/APIRest_BBDDActoresV9/crud/leer.php

✓ Otros ejemplo de Endpoints:

• <https://stapi.co/api/v1/rest/episode/search> En este caso concreto, este endpoint permite listar todos los episodios en el servidor de STAPI. Un endpoint se divide en:

• <https://stapi.co/api> : **Dominio base y raíz de api STAPI**, digamos que es el origen desde donde se ven todos los servicios.

• /v1/rest/episode/search: **Versión, ruta y recurso específico** que queremos acceder, en este caso, listar episodios.

• <https://swapi.py4e.com/api/> **Dominio base y raíz de api SWAPI**

1 API REST Y ENDPOINTS



- ✓ Cada API es de su padre y de su madre!!! → Revisar documentación. Ej: documentación de STAPI está en <https://stapi.co/api-documentation>
- ✓ Utilidades: **SWAGGER EDITOR**, permite describir la documentación de las APIs (usando yaml) de forma que se definen endpoints, parámetros, respuestas posibles, etc. Permite PROBAR!
- ✓ STAPI tiene un enlace para poder usar con este programa.
<https://editor.swagger.io/?url=https://stapi.co/api/v1/rest/common/download/stapi.yaml>

2 RETROFIT

✓ **RETROFIT:** Librería de cliente HTTP para Android y JAVA que permite explotar APIs de forma muy sencilla (desde Android se puede utilizar de una forma muy similar)

✓ **PASOS:**

Lo vemos en apirest_ejemplo_javafx_actorescine

1. Declarar librería Retrofit y Gson en build.gradle

```
implementation 'com.squareup.retrofit2:retrofit:2.11.0'
```

```
implementation 'com.squareup.retrofit2:converter-gson:2.11.0'
```

2. Crear una **interface** pública por cada endpoint (donde se definen cómo se le pasan los datos). Es decir, creamos una **interface** por cada operación CRUD

✓ **CLASE INTERFACE** (define llamada pero NO cómo se procesa):

- Se indica el método con una @ seguido de la página php o endpoint.

- Se declara el método de la interfaz junto a los datos de entrada. Además, se utilizan modificadores según “la forma de enviar datos a la API”:

2 RETROFIT

✓ @Query:

- ✓ Se utiliza para agregar **parámetros** a la URL en RETROFIT, normalmente se utiliza con GET. Los parámetros de la cadena de consulta se añaden después del signo de interrogación (?) y se separan por ampersands (&).
Ej: .getActor(93) → leer.php?id=93

```
public interface MiAPIServicioLeer {  
    @GET("leer.php") //Usa el método GET  
    Call<Actor> getActor(@Query("id") int id);  
}
```

✓ @Body:

- ✓ Se utiliza para especificar que el parámetro debe enviarse en el cuerpo de la solicitud HTTP. Normalmente se utiliza con POST o PUT. Ejemplo más típico: envío de datos en formato **JSON**.

```
public interface MiAPIServicioInsertar {  
    @Headers({  
        "Content-Type: application/json",  
        "Accept: application/json"  
    })  
    @POST("insertar.php")  
    Call<Actor> insertarData(@Body Actor a);  
}
```

Además, @Headers permite indicar modificadores de cabecera, en este caso se indica que “los datos que se envían irán en JSON” y que también se “espera una respuesta en JSON”

2 RETROFIT

✓ CLASE INTERFACE:

- ✓ En APIs como SWAPI, se utiliza **IRI** para las consultas, de esta forma, para consultar la persona con id=3 haríamos esto:

<https://swapi.py4e.com/api/people/3/> ya que lo siguiente NO funcionaría:
<https://swapi.py4e.com/api/people?id=3/>

- ✓ En este caso NO podemos utilizar **@Query**, deberemos utilizar **@Path** y que sustituya en tiempo de ejecución una variable para modificar parte de la URL (no tiene por qué ser la parte última)

- ✓ **@Path**: La anotación @Path se utiliza para especificar **partes variables de la URL** que se reemplazarán por valores concretos en tiempo de ejecución en cualquier parte de la URL. Esto es muy útil en APIs donde la forma de consulta no utiliza variables sino una construcción tipo **IRI**:

```
public interface MiAPIServicioLeer {  
    @GET("people/{id}")  
    Call<Person> getPersonById(@Path("id") int mid);  
}
```

2 RETROFIT

✓ PASOS:

- Crear un **modelo** que soporte los datos según estructura JSON!!

Ejemplo API Actores:

```
id:      92
nombre:  "Tom Hanks"
edad:    65
activo:   1
fotourl:  ""
fotocodif: ""
```

```
public class Actor {

    Integer id;
    String nombre;
    Integer edad;
    Integer activo;
    String fotourl;
    String fotocodif;

    public Actor(Integer id, String nombre, Integer edad, Integer activo, String fotourl, String fotocodif) {
        this.id = id;
        this.nombre = nombre;
        this.edad = edad;
        this.activo = activo;
        this.fotourl = fotourl;
        this.fotocodif = fotocodif;
    }

    public Integer getId() {
        return id;
    }
}
```

2 RETROFIT

✓ PASOS:

- Crear un **modelo** que soporte los datos según estructura JSON!!

Normalmente los modelos son DISTINTOS, como por ejemplo SWAPI VS STAPI.

No nos referimos a diferencias obvias entre tipos de datos sino a la **ESTRUCTURA** de la respuesta.

name:	"Luke Skywalker"
height:	"172"
mass:	"77"
hair_color:	"blond"
skin_color:	"fair"
eye_color:	"blue"
birth_year:	"19BBY"
gender:	"male"
homeworld:	"https://swapi.py4e.com/api/planets/1/"
films:	
0:	"https://swapi.py4e.com/api/films/1/"
1:	"https://swapi.py4e.com/api/films/2/"

Array

episode:

uid:	"EPMA0000001002"
title:	"Til Death Do Us Part"
titleGerman:	"Bis daß der Tod uns scheide"
titleItalian:	null
titleJapanese:	null
series:	
uid:	"SEMA0000073238"
title:	"Star Trek: Deep Space Nine"

SWAPI devuelve directamente la respuesta sin array, STAPI devuelve un array.

2 RETROFIT

✓ PASOS:

4. Creación de un **convertidor JSON** (Librería de Google para convertir de Java a JSON y viceversa)
5. Creación de **instancia de Retrofit**
6. Creación de **instancia de cada interface pública (cada endpoint)**

```
String baseUrl = "http://localhost/APIRest_BBDDActoresV8/crud/";

Gson gson = new GsonBuilder().setLenient().create();
//Instancia a retrofit agregando la baseURL y el convertidor JSON
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl(baseUrl)
    .addConverterFactory(GsonConverterFactory.create(gson))
    .build();

//Se crea el servicio
MiAPIServicioLeer servicioLeer = retrofit.create(MiAPIServicioLeer.class);
```

2 RETROFIT

✓ PASOS:

7. Construcción de datos y llamada a la interface pública (devuelve un objeto tipo Call (llamada asíncrona). Las llamadas encolaXXXXX lo que hacen es gestionar las posibles respuestas (mensajes o errores)

```
295 if ( ! is_dirt ) {  
296     private Call<Actor> callLect;  
297     mkdirs( s );  
298 }  
299  
300     combo.setOnAction( e -> {  
301         System.out.println("Selección actual: " + combo.getSelectionModel().getSelectedItem());  
302         label.setText( string: "Esperando...." );  
303         callLect = servicioLeer.getActor[ id: combo.getSelectionModel().getSelectedItem() );  
304         this.encolaLectura();  
305     }  
306 }  
307  
308 // Resize and save image  
$image = Image[ width: 1000, height: 1000 ]  
$image->resize( $width, $height )  
->save( $path, $format )
```

- ✓ **Estructura de llamada asíncrona Retrofit.** La estructura es siempre la misma: se encola una llamada asíncrona, esto es, un objeto CallBack. Lógicamente cambiará el modelo y la respuesta. El método ***onFailure*** se dispara cuando hay fallo de conexión, el ***onResponse*** es la respuesta del servidor (mensaje o error de datos). Ej: lectura.

```
public void encolaLectura() {  
    callLect.enqueue(new Callback<Actor>() {  
        @Override  
        public void onFailure(Call<Actor> call, Throwable t) {  
            System.out.println("Network Error :: " + t.getLocalizedMessage());  
        }  
  
        @Override  
        public void onResponse(Call<Actor> call, Response<Actor> response) {  
            Platform.runLater(() -> { //  
                System.out.println("Respuesta LECTURA: " + response.message());  
                if (response.isSuccessful()) {  
                    //TODO OK  
                } else {  
                    //ERROR  
                }  
            });  
        }  
    });  
}
```

2 APIs PÚBLICAS Y API KEYS



- ✓ Lo normal es que sean abiertas (sin API KEY), pero las hay que sí lo requieren.
- ✓ API Key es un STRING que permite conectarnos a la API (autentificación).
- ✓ Ejemplos de APIs con KEY:
 - Nasa: <https://api.nasa.gov/> (apartado Generate Api KEY)
 - Google Maps (API obtenida vía Google Cloud)
- ✓ Ejemplos de APIs sin KEY: SWAPI y STAPI.

Lo vemos en apirest_ejemplo_javafx_swapi y
apirest_ejemplo_javafx_stapi