

Tema 5.2 - Retrofit

Descargar estos apuntes [pdf](#) o [html](#)

Índice

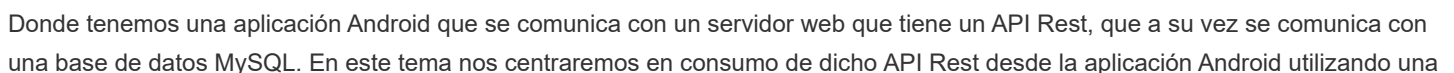
- ▼ Introducción
 - ▼ ¿Qué es JSON?
 - Tipos de datos en JSON
- ▼ Definición de un Servidor Rest rápido para pruebas
 - Creando la Base de Datos con phpMyAdmin
- ▼ Consumo de un Servicio Rest desde Android
 - Configuración del proyecto
 - ▼ Crear los servicios con Retrofit
 - Definiendo los tipos a serializar a JSON
 - Definiendo las peticiones para consumo del '*endpoint*'
 - Preparando los objetos de Retrofit con Hilt
 - Implementaciones de la gestión del '*consumo*' de nuestro endpoint
 - ▼ Usando nuestra implementación del servicio en el patrón Repository
 - Acceder a la API iniciando una sesión en el servidor

Introducción

- **Persistencia remota consumiendo un API Rest con Retrofit2**
 - Página oficial librería: [Retrofit](#)
 - Guía usuario: [Gson](#)
 - Video Tutorial (Castellano): [Martin Kiperszmid](#)
 - Video Tutorial **Interceptores** (Castellano): [Martin Kiperszmid](#)
 - Video Tutorial (Castellano): [DevExperto](#)
 - Video Tutorial (Castellano): [AristiDevs](#)

En la gran mayoría de aplicaciones móviles, se necesita acceder a datos que no están en el dispositivo, sino que están en un servidor remoto. Para ello se utilizan que permiten el acceso a los datos a través de internet. Estos servicios pueden ser **API Rest**, **GraphQL**, **WebSockets**, **gRPC**, etc. En este tema nos centraremos en los servicios API Rest, que son los más utilizados.

Los Servicios web REST, utilizan el protocolo HTTP para intercambiar información entre el cliente y el servidor. Nosotros en el curso vamos a utilizar una arquitectura similar a la siguiente:



Información

Uno de los inconvenientes de **Retrofit2** es que al funcionar sobre la librería de Java **OkHttp**, no es compatible con la programación multiplataforma. Disponemos de una alternativa muy potente y extendida denominada **Ktor Client** que además es una muy buena documentación dispone de una montón de [vídeos de la comunidad explicando su uso en Android](#). Existen incluso librerías como **Kotrfit** que imitan el funcionamiento de Retrofit2 pero utilizando Ktor Client por debajo y así facilitar la migración.

El formato de los datos que se intercambian entre el cliente y el servidor es muy importante. En este tema nos centraremos en el formato **JSON**.

¿Qué es JSON?

Aunque seguramente ya lo has visto en el módulo de Acceso a Datos. Vamos a realizar un resumen rápido sobre dicho formato a modo de recordatorio.

JSON (Javascript Object Notation) es un formato ligero de intercambio de datos entre clientes y servidores, basado en la sintaxis de Javascript para representar estructuras en forma organizada. Es un formato en texto plano independiente de todo lenguaje de programación, es más, soporta el intercambio de datos en gran variedad de lenguajes

Tipos de datos en JSON

Similar a la estructuración de datos primitivos y complejos en los lenguajes de programación, JSON establece varios tipos de datos: **cadenas**, **números**, **booleanos**, **arrays** y **objetos**. El propósito es crear objetos que contengan varios atributos compuestos como pares clave valor. Donde la clave es un nombre que identifique el uso del valor que lo acompaña. Veamos un ejemplo:

```
{
  "id": 101,
  "nombre": "Carlos",
  "estaActivo": true,
  "notas": [2.3, 4.3, 5.0]
}
```

La anterior estructura es un objeto JSON compuesto por los datos de un estudiante. Los objetos JSON contienen sus atributos entre llaves **{ }**, al igual que un bloque de código en Javascript, donde cada atributo debe ir separado por coma **,** para diferenciar cada par.

La sintaxis de los pares debe contener dos puntos **:** para dividir la clave del valor. El nombre del par debe tratarse como cadena y añadirle comillas dobles.

Si te fijas en nuestro ejemplo, este trae un ejemplo de cada tipo de dato:

- **id** es de tipo entero, ya que contiene un número que representa el código del estudiante.
- **nombre** es un string. Usa comillas dobles para definirlas.
- **estaActivo** es un tipo booleano que representa si el estudiante se encuentra en la institución educativa o no. Usa las palabras reservadas **true** y **false** para declarar el valor.
- **notas** es un arreglo de números reales. El conjunto de sus elementos debes incluirlos dentro de corchetes **[]** y separarlos por coma.

La idea es crear un mecanismo que permita recibir la información que contiene la base de datos externa en formato JSON hacia la aplicación. Con ello se parseará cada elemento y será interpretado en forma de objeto Kotlin para integrar correctamente el aspecto en la interfaz de usuario.

Para realizar este proceso podemos utilizar diferentes librerías para la serialización y deserialización de objetos JSON. En este tema nos centraremos en la librería de Google **Gson**, aunque **Retrofit2** es agnóstico a la forma de serializar y también nos permite utilizar otras librerías como **Jackson**, **Moshi**, etc.

Definición de un Servidor Rest rápido para pruebas

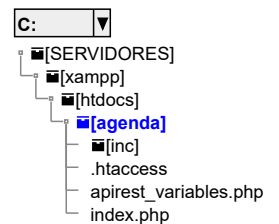
Código de los ejemplos

Si te surge alguna duda o tienes dificultades para completar este tema. Puedes descargar el proyecto con el código de mismo del siguiente enlace: [Proyecto ejemplos](#)

Una forma rápida de **definir una API Rest rápido** para probar desde Android, es mediante la librería de PHP proporcionada en el módulo de Acceso a datos [apiRest-PHP-Session.zip](#).

Por si no la has visto en el módulo de Acceso a Datos, vamos a realizar un resumen rápido sobre el uso dicha librería...

Debes instalar un servidor web Apache con PHP y MySQL. En el módulo de Acceso a Datos se ha utilizado **xampp**. En la carpeta `xampp\htdocs`, o en la carpeta pública de tu servidor web, debes descomprimir el fichero **apiRest-PHP-Session.zip** y renombrar su contenido por ejemplo por la BD que quieras utilizar. Supongamos que queremos crear el servicio de API Rest para la Agenda que hemos ido creando durante el curso. En este caso renombraremos la carpeta **apiRest-PHP-Session** por **agenda** como se aprecia a la derecha **agenda**.



La configuración de nuestro API consta de dos archivos:

1. **.htaccess** En este archivo se configuran las reglas de acceso, y solo deberemos modificar la línea **RewriteBase** para indicar la ruta de acceso a la carpeta de nuestro API. En nuestro caso cambiaremos el valor de la cadena **"/apirest-session/"** por **"/agenda/"**.

```
RewriteEngine On
2 RewriteBase "/apirest-session/"
...

RewriteEngine On
6 RewriteBase "/agenda/"
...
```

2. **apiREST_variables.php** , en este archivo se definen los datos de conexión a la base de datos. se indican las tablas que tiene esta y el nombre del identificador de cada una de ellas.

```
<?php

// CONFIGURACIÓN BASE DE DATOS MYSQL
$servername = "127.0.0.1";
$username = "root";
$password = "";

// BASE DE DATOS
$dbname = "agenda";

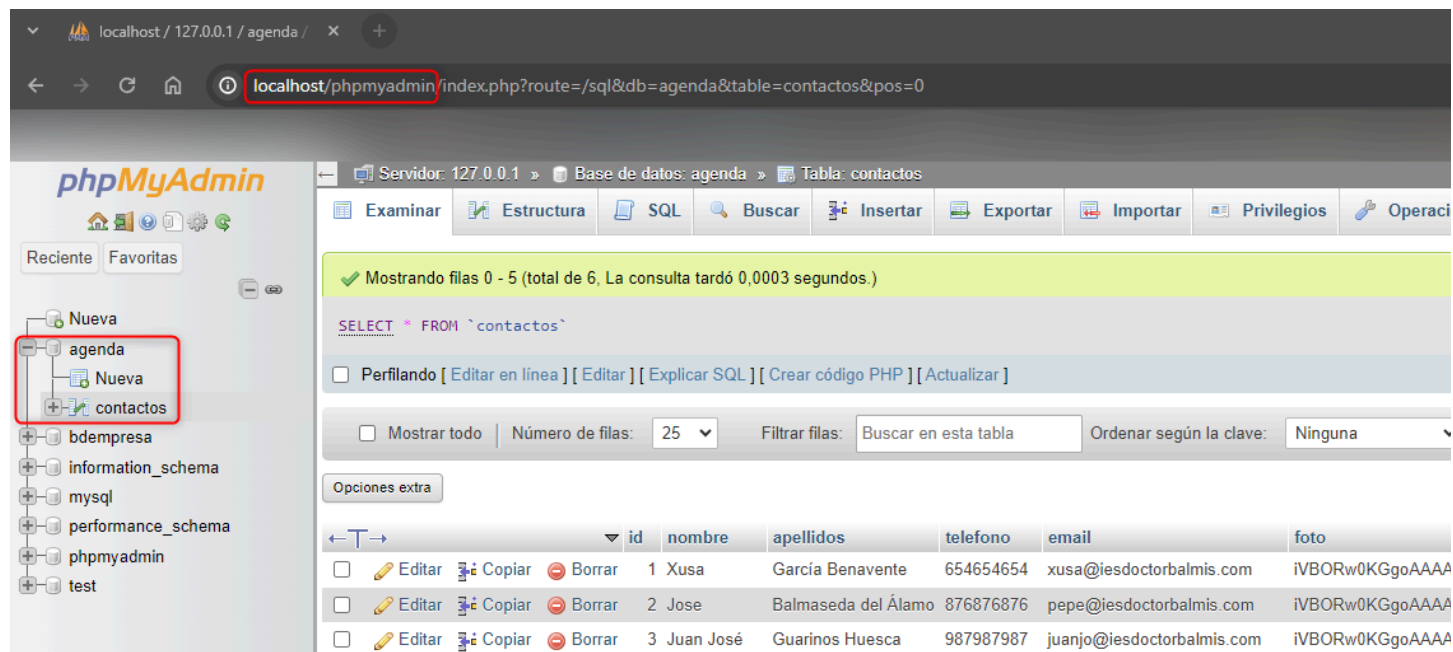
// ACCESO USUARIOS (si está vacío funciona sin usuarios)
$usuarios = array();
// $usuarios["juanjo"]="_IesBalmis1";
// $usuarios["xusa"]="_IesBalmis2";
// $usuarios["pepe"]="_IesBalmis3";

// TABLAS Y SU CLAVE
$tablas = array();
$tablas["contactos"]="id";
```

El resto de archivos del ApiREst **no tendrán que modificarse**, ya que está construida de forma genérica con las necesidades más comunes para estos casos.

Creando la Base de Datos con phpMyAdmin

Para crear la base de datos con **phpMyAdmin**, deberemos crear una base de datos y para ello puedes bajarte el siguiente **recurso** en el que encontrarás el archivo **agenda_mysql_datos.sql** que contiene el script de creación de la base de datos, incluyendo las imágenes de ejemplo en formato **blob** (base64).



Para acceder a **phpMyAdmin** debes ir a la url **`http://localhost/phpmyadmin/`** y ejecutar **agenda_mysql_datos.sql** en la pestaña SQL. Tras hacerlo, te debe aparecer la base de datos **agenda** con la tabla **contactos** como se muestra en la imagen de ejemplo.

Puedes probar que el API está funcionando correctamente, abriendo un navegador y accediendo a la url **`http://localhost/agenda/`**. Si todo ha ido bien, deberías ver una página como la siguiente:

API REST en PHP

ESPECIFICACIÓN DE INTERFAZ

Tabla	Clave	Campos
contactos	id	nombre apellidos telefono email foto categorias

Método	Query String	Descripción
GET	/	Muestra las especificaciones del API Rest
GET PUT POST DELETE	/datos	Muestra en formato HTML todos los datos implicados en la llamada al API Rest incluidos los recibidos.
GET	/tabla	Muestra los datos de todos los registros de tabla
GET	/tabla/count	Muestra el número de registros de tabla
GET	/tabla/valor	Muestra los datos del registro cuya clave de la tabla es igual a valor , teniendo en cuenta el campo clave de las variables definidas
GET	/tabla/campo/?desde=valor	Muestra los datos de los registros de la tabla cuyo campo sea mayor o igual que el valor
GET	/tabla/campo/?hasta=valor	Muestra los datos de los registros de la tabla cuyo campo sea menor o igual que el valor
GET	/tabla/campo/?desde=valor1&hasta=valor2	Muestra los datos de los registros de la tabla cuyo campo sea mayor o igual que el valor1 y menor o igual que el valor2
DELETE	/tabla/valor	Elimina el registro cuya clave de la tabla es igual a valor , teniendo en cuenta el campo clave de las variables definidas
PUT	/tabla/valor	Actualiza el registro cuya clave de la tabla es igual a valor , teniendo en cuenta el campo clave de las variables definidas
POST	/tabla	Inserta en tabla un registro nuevo con los datos recibidos

Consumo de un Servicio Rest desde Android

Como ya hemos comentado, existen diferentes librerías que nos permitirían consumir los servicios desde la App de Android, pero dada su facilidad vamos a utilizar las librerías: **Retrofit2** y **Gson**.

Retrofit la utilizaremos para hacer peticiones http y procesar las respuestas del API Rest, mientras que con **Gson** transformaremos los datos de JSON a los propios que utilice la aplicación.

Configuración del proyecto

Para poder utilizar Retrofit y Gson en nuestro proyecto, deberemos añadir:

En el catálogo de versiones `lib.versions.toml` deberemos comprobar que hemos definido tener:

```
[versions]
retorfit = "2.11.0"
okhttp3 = "4.12.0"

[libraries]
com-squareup-retrofit2-converter-gson = {
    group = "com.squareup.retrofit2", name = "converter-gson", version.ref = "retorfit"
}
com-squareup-retrofit2-retrofit = {
    group = "com.squareup.retrofit2", name = "retrofit", version.ref = "retorfit"
}
com-squareup-okhttp3-okhttp-bom = {
    group = "com.squareup.okhttp3", name = "okhttp-bom", version.ref = "okhttp3"
}
com-squareup-okhttp3-okhttp = {
    group = "com.squareup.okhttp3", name = "okhttp"
}
com-squareup-okhttp3-logging-interceptor = {
    group = "com.squareup.okhttp3", name = "logging-interceptor"
}
```

Acuérdate de escribir en una sola línea la definición de las librerías.

En el fichero `build.gradle.kts` del módulo de la aplicación:

```
dependencies {
    ...
    implementation(libs.com.squareup.retrofit2.converter.gson)
    implementation(libs.com.squareup.retrofit2.retrofit)
    implementation(platform(libs.com.squareup.okhttp3.okhttp.bom))
    implementation(libs.com.squareup.okhttp3.okhttp)
    implementation(libs.com.squareup.okhttp3.logging.interceptor)
}
```

En el archivo `AndroidManifest.xml` deberemos añadir el permiso de acceso a internet para que el servicio pueda acceder al API.

```
<manifest ...>
...
3 <uses-permission android:name="android.permission.INTERNET"/>
  <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
5 <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
...
<application ...>
  <!-- Permitir tráfico http en lugar de https -->
9  android:usesCleartextTraffic="true"
  ...
</application>
</manifest>
```

Crear los servicios con Retrofit

En la paquete `data` crearemos un nuevo paquete llamado `data.services` donde definiremos los API de cara uno de nuestro 'endpoint'.

Definiendo los tipos a serializar a JSON

Para cada una de las clases que se van a transferir en las peticiones crearemos un fichero `<Tipo>Api.kt`.

1. Definiremos la clase `ContactoApi` que será la que se utilizará para transferir los datos de los contactos entre el servidor y la aplicación. Fíjate que los atributos de la clase tienen que tener el mismo nombre que los campos que se devuelven en el JSON del API Rest. Si quisiéramos cambiar el nombre de alguna propiedad, deberemos utilizar la anotación `@SerializedName` justo antes de la propiedad para indicar el nombre que tiene en el JSON. Puedes obtener más información sobre transformaciones de en la [documentación de la librería](#).

```
// ContactoApi.kt
data class ContactoApi(
    val id: Int,
    4 @SerializedName(value = "nombre")
    val nombre: String,
    val apellidos: String,
    val telefono: String,
    val email: String,
    val foto: String?,
    val categorias: String
)
```

2. La librería de PHP que estamos usando, en la peticiones de tipo **POST**, **PUT** y **DELETE**, nos devuelve una respuesta en JSON con campos de información sobre la petición, por tanto, deberemos definir un objetos para su deserialización.

```
// RespuestaApi.kt
data class RespuestaApi (
    val respuesta : Int,
    val metodo: String? = null,
    val tabla: String? = null,
    val mensaje: String? = null,
    val sqlQuery: String? = null,
    val sqlError: String? = null
)
```

3. Por último, las petición <http://localhost/agenda/contactos/count> devuelve un número de contactos en un JSON especial, por lo que deberemos definir un objeto para su deserialización.

```
// TotalRegistrosApi.kt
data class TotalRegistrosApi(
    @SerializedName("tabla")
    val tabla: String,
    @SerializedName("total_registros")
    val totalRegistros: Int
)
```

Definiendo las peticiones para consumo del 'endpoint'

Para cada una de las peticiones que se vayan a realizar al API Rest, crearemos una interfaz con el nombre de la clase del API, en nuestro caso `ContactoApi` y le añadiremos el sufijo `Service`. Pot tanto, vamos a definir un interface llamado `ContactoService`.

Definiremos pues la signature de cada uno delos métodos que se van a utilizar para realizar las peticiones Para ello utilizaremos diferentes anotaciones para indicar el tipo de petición, la url del API Rest, el tipo de datos que se envía y el tipo de datos que se recibe.

Por ejemplo, para la petición `http://localhost/agenda/contactos` que devuelve un listado de contactos, deberemos añadir...

```
interface ContactoService {
    @GET("contactos")
    @Headers("Accept: application/json", "Content-Type: application/json")
    suspend fun contactos(): Response<List<ContactoApi>>
}
```

1. La anotación `@GET` con la url `contactos` que completaría la URL base al definir el objeto de Retrofit que será `http://localhost/agenda/`.
2. La anotación `@Headers` que incluirá en la cabecera de la petición HTTP los valores `Accept` y `Content-Type` para indicar que se envía y se espera recibir JSON en el **body**.
3. Por último, definiremos la signatura del método que será de suspensión `suspend` y devolverá un objeto de tipo `Response` que contendrá una lista de objetos de tipo `ContactoApi` ya deserializados si la respuesta es correcta.

En el caso de que la respuesta en el body no sea un objeto del tipo `ContactoApi` como el caso de la petición `http://localhost/agenda/contactos/count` que devuelve un objeto del tipo `TotalRegistrosApi`, `Response` irá parametrizado con este tipo para la correcta deserialización.

```
interface ContactoService {
    // ...

    @GET("contactos/count")
    @Headers("Accept: application/json", "Content-Type: application/json")
    suspend fun count(): Response<TotalRegistrosApi>
}
```

Podremos poner también el valor de un parámetro en la URL con la anotación `@Path`, así como serializar en el 'body' de la petición un objeto con `@Body`. Por ejemplo, para la petición `@PUT` a nuestro Api con PHP tenemos que indicar el **id** del contacto a actualizar `http://localhost/agenda/contactos/1` y en el cuerpo de la petición el objeto `ContactoApi` con los datos a actualizar. Fíjate además que la respuesta que parametrizamos en el objeto `Response` es del tipo `RespuestaApi` que definimos anteriormente. Nuestro prototipo de método `update` quedaría de la siguiente manera...

```
interface ContactoService {
    // ...
    @PUT("contactos/{id}")
    @Headers("Accept: application/json", "Content-Type: application/json")
    suspend fun update(@Path("id") id: Int, @Body c : ContactoApi): Response<RespuestaApi>
}
```

También podemos establecer parámetros en el **QUERYSTRING** de la URL con la anotación `@Query`. Por ejemplo, para la petición `http://localhost/agenda/contactos/id/?desde=3&hasta=5` que devuelve un **listado de contactos con id entre 3 y 5**. Nuestro prototipo de método `contactosDesdeHasta` quedaría de la siguiente manera...

```
interface ContactoService {
    // ...
    @GET("contactos/id/")
    @Headers("Accept: application/json", "Content-Type: application/json")
    suspend fun contactosDesdeHasta(
        @Query("desde") desde: Int,
        @Query("hasta") hasta: Int
    ): Response<List<ContactoApi>>
}
```

Con lo visto ya podemos definir el resto de métodos HTTP que necesitemos para nuestro API Rest. En nuestro caso, nos quedan por definir los siguientes los siguientes ...

```
interface ContactoService {  
    // ...  
    @GET("contactos/{id}")  
    @Headers("Accept: application/json", "Content-Type: application/json")  
    suspend fun contacto(@Path("id") id: Int): Response<ContactoApi>  
  
    @POST("contactos")  
    @Headers("Accept: application/json", "Content-Type: application/json")  
    suspend fun insert(@Body c : ContactoApi): Response<RespuestaApi>  
  
    @DELETE("contactos/{id}")  
    @Headers("Accept: application/json", "Content-Type: application/json")  
    suspend fun delete(@Path("id") id: Int): Response<RespuestaApi>  
}
```

Preparando los objetos de Retrofit con Hilt

En el fichero `.di/AppModule.kt` deberemos definir como crear los objetos a inyectar de Retrofit.

1. Primero definimos como crear el objeto `OkHttpClient`, para ello usaremos `OkHttpClient.Builder()`. En este caso le añadiremos un **interceptor** o ('hook') para poder depurar, a través de Logcat de Android Studio, el contenido de las peticiones y respuestas HTTP que se realizan. Fíjate que el nivel de log que le hemos indicado es **HEADERS**, esto es porque no queremos que se muestre la cabecera sin el cuerpo de la petición. Además, le hemos indicado un tiempo de espera ('*TIMEOUT*') de **10 segundos** para las peticiones después de los cuales se cancelará la petición y se producirá una excepción de tipo `SocketTimeoutException`.

```
@Provides
@Singleton
fun provideOkHttpClient() : OkHttpClient {
    val loggingInterceptor = HttpLoggingInterceptor()
    loggingInterceptor.level = HttpLoggingInterceptor.Level.HEADERS

    val timeout = 10L
    return OkHttpClient.Builder()
        .addInterceptor(loggingInterceptor)
        .connectTimeout(timeout, TimeUnit.SECONDS)
        .readTimeout(timeout, TimeUnit.SECONDS)
        .writeTimeout(timeout, TimeUnit.SECONDS)
        .build()
}
```

2. Ahora definiremos el objeto **Retrofit** que es el que usaremos realmente para realizar las peticiones HTTP y procesar las respuestas del API Rest. Al mismo le pasaremos:

- El objeto **OkHttpClient** que hemos creado anteriormente y que le llega a través de la inyección.
- La url base del API Rest, en nuestro caso **http://10.0.2.2/agenda/** . Fíjate que la dirección no es **localhost** o **127.0.0.1** ya que, cómo estamos accediendo desde el dispositivo emulador para él el **localhost** es el propio dispositivo Android emulado y no el equipo donde está el servidor web. AVD (Android Virtual Device) proporciona una dirección IP especial **10.0.2.2** que nos permite acceder al equipo donde está el servidor web.

```
@Provides
@Singleton
fun provideRetrofit(
    okHttpClient: OkHttpClient
) : Retrofit = Retrofit.Builder()
    .client(okHttpClient)
    .baseUrl("http://10.0.2.2/agenda/")
    .addConverterFactory(GsonConverterFactory.create())
    .build()
```

3. Por último, **indicaremos a Hilt como instanciar el o los objetos de <endpoint>Service** que es el que realmente utilizaremos para realizar las peticiones al API Rest. Para ello le pasaremos el objeto **Retrofit** que hemos creado anteriormente y que le llega a través de la inyección. En nuestro caso solo vamos a definir como instanciar un objeto que implemente la interfaz **ContactoService** que utilizaremos para realizar las peticiones al API Rest de la Agenda que es el único '*endpoint*' definido.

```
@Provides
@Singleton
fun provideContactoService(
    retrofit: Retrofit
) : ContactoService = retrofit.create(ContactoService::class.java)
```

Implementaciones de la gestión del '*consumo*' de nuestro endpoint

Aunque **este paso intermedio no es de todo necesario** y no lo vamos a ver en muchos ejemplos de uso de Retrofit. Si que **es recomendable para gestionar correctamente los errores y los logs de depuración que se puedan producir** al consumir nuestro API Rest y simplificar el código de uso de Retrofit en nuestro patrón Repository.

Primero definiremos la clase `ApiServicesException` que será la que utilizaremos para lanzar las excepciones que se produzcan al consumir el API Rest.

```
class ApiServicesException(mensaje: String) : Exception(mensaje)
```

Posteriormente, definiremos para ello una clase denominada `ContactoServiceImplmentation` a la que le inyectaremos una instancia de `ContactoService` que es la que realmente utilizaremos para realizar las peticiones al API Rest.

Veamos la anatomía de uso de Retrofit para **obtener la lista de contactos** del API Rest en esta clase comentado paso por paso...

```
@Singleton
class ContactoServiceImplmentation @Inject constructor(
    private val contactoService: ContactoService
) {
    // Propiedad privada cte. donde definimos el tag para los logs
    // de depuración de las peticiones.
    private val logTag: String = "OkHttp"
    suspend fun get(): List<ContactoApi> {
        val mensajeError = "No se han podido obtener los contactos"
        try {
            // Obtenemos la respuesta HTTP Response<List<ContactoApi>>
            val response = contactoService.contactos()
            if (response.isSuccessful) {
                Log.d(logTag, response.toString())
                // Si la respuesta tiene un estatus 2xx (200, 201, 202, etc.)
                // Obtenemos con response.body los datos List<ContactoApi>
                // ya deserializados de JSON contenidos en el cuerpo de la misma.
                // Si no hay datos porque el resultado de la serialización
                // es null o el cuerpo estaba vacío. Entonces, lanzamos un
                // error indicando que no hay datos.
                val dato = response.body()
                return dato ?: throw ApiServicesException("No hay datos")
            } else {
                // sino entonces la respuesta HTTP tiene un estatus de error y por
                // tanto obtendré el mensaje de error del body de la respuesta
                // y lanzaremos un error genérico, enviando al al mismo tiempo el
                // mensaje generado al Log.
                val body = response.errorBody()?.string()
                Log.e(logTag, "$mensajeError (código ${response.code()}) : $this\n${body}")
                throw ApiServicesException(mensajeError)
            }
        } catch (e: Exception) {
            // Si ha habido algún error al deserializar el JSON
            // o también si ha habido algún error al realizar la petición por
            // ejemplo por falta de conexión a internet, o se ha
            // producido un TIMEOUT, etc.
            Log.e(logTag, "Error: ${e.localizedMessage}")
            throw ApiServicesException(mensajeError)
        }
    }

    // ... resto de la implementación de las llamadas al Servicio Rest
}
```

Siguiendo el esquema anterior, **obtener un contacto por ID** quedaría...


```

suspend fun get(id: Int): ContactoApi {
    val mensajeError = "No se han podido obtener el contacto con id = $id"
    try {
        4      val response = contactoService.contacto(id)
        if (response.isSuccessful) {
            Log.d(logTag, response.toString())
            val dato = response.body()
            8      return dato ?: throw ApiServiceException("No hay datos")
        } else {
            val body = response.errorBody()?.string()
            Log.e(logTag, "$mensajeError (código ${response.code()}) : $this\n${body}")
            throw ApiServiceException(mensajeError)
        }
    } catch (e: Exception) {
        Log.e(logTag, "Error: ${e.localizedMessage}")
        throw ApiServiceException(mensajeError)
    }
}

```

Para **insertar un contacto** en el API Rest tendremos ...

```

suspend fun insert(contacto: ContactoApi) {
    val mensajeError = "No se ha podido añadir el contacto"
    try {
        4      val response = contactoService.insert(contacto)
        if (response.isSuccessful) {
            Log.d(logTag, response.toString())
            7      // Aquí response.body() es un objeto de tipo RespuestaApi
            // que simplemente logeamos si no es null.
            9      Log.d(logTag, response.body()?.toString() ?: "No hay respuesta")
        } else {
            val body = response.errorBody()?.string()
            Log.e(logTag, "$mensajeError (código ${response.code()}) : $this\n${body}")
            throw ApiServiceException(mensajeError)
        }
    } catch (e: Exception) {
        Log.e(logTag, "Error: ${e.localizedMessage}")
        throw ApiServiceException(mensajeError)
    }
}

```

Para **actualizar un contacto** en el API Rest tendremos ...

```

suspend fun update(contacto: ContactoApi) {
    val mensajeError = "No se ha podido actualizar el contacto"
    try {
        4      // En este método el API de PHP espera recibir el id del contacto
        // que también lo podemos obtener del objeto contacto que le pasamos
        6      val response = contactoService.update(contacto.id, contacto)
        if (response.isSuccessful) {
            Log.d(logTag, response.toString())
            Log.d(logTag, response.body()?.toString() ?: "No hay respuesta")
        } else {
            val body = response.errorBody()?.string()
            Log.e(logTag, "$mensajeError (código ${response.code()}) : $this\n${body}")
            throw ApiServiceException(mensajeError)
        }
    } catch (e: Exception) {
        Log.e(logTag, "Error: ${e.localizedMessage}")
        throw ApiServiceException(mensajeError)
    }
}

```

Para **borrar un contacto** en el API Rest tendremos ...

```
suspend fun delete(id: Int) {
    val mensajeError = "No se ha podido borrar el contacto"
    try {
        val response = contactoService.delete(id)
        if (response.isSuccessful) {
            Log.d(logTag, response.toString())
            Log.d(logTag, response.body()?.toString() ?: "No hay respuesta")
        } else {
            val body = response.errorBody()?.string()
            Log.e(logTag, "$mensajeError (código ${response.code()}) : $this\n${body}")
            throw ApiServiceException(mensajeError)
        }
    } catch (e: Exception) {
        Log.e(logTag, "Error: ${e.localizedMessage}")
        throw ApiServiceException(mensajeError)
    }
}
```

Usando nuestra implementación del servicio en el patrón Repository

Al igual que sucedía con las anteriores fuentes como los objetos Mock de prueba o las entidades de room. Deberemos definir en `RepositoryConverters.kt` las funciones de extensión para convertir los objetos de tipo `ContactoApi` en objetos de tipo `Contacto` y viceversa.

```
fun Contacto.toContactoApi() = ContactoApi(...)
fun ContactoApi.toContacto() = Contacto(...)
```

Por último, en el fichero `ContactoRepository.kt` deberemos inyectar la implementación de nuestro servicio `ContactoServiceImplementation` y utilizarlo en las funciones de nuestro patrón Repository.



Importante

Cualquier error que se produzca ya lo resolveremos en el **ViewModel** como sucedía con **room**.

```
class ContactoRepository @Inject constructor(
    private val contactoService: ContactoServiceImplementation
) {
    suspend fun get(): List<Contacto> = withContext(Dispatchers.IO) {
        contactoService.get().map { it.toContacto() }
    }
    suspend fun get(id: Int): Contacto = withContext(Dispatchers.IO) {
        contactoService.get(id).toContacto()
    }
    suspend fun insert(contacto: Contacto) = withContext(Dispatchers.IO) {
        contactoService.insert(contacto.toContactoApi())
    }
    suspend fun update(contacto: Contacto) = withContext(Dispatchers.IO) {
        contactoService.update(contacto.toContactoApi())
    }
    suspend fun delete(id: Int) = withContext(Dispatchers.IO) {
        contactoService.delete(id)
    }
}
```

Acceder a la API iniciando una sesión en el servidor

Si queremos incrementar la seguridad del acceso a nuestra APIRest, podemos **iniciar una sesión en el servidor** con determinadas credenciales. Tran enviar la petición de *'login'*, capturaremos la *'cookie'* que nos devuelve con el id de la sesión y usaremos esta misma cookie para poder acceder al resto de funcionalidades del APIRest. Para habilitar la autenticación a traves del uso de sesiones en nuestro APIRest con PHP, volveremos a editar el archivo `apirest_variables.php` y descomentaremos las siguientes líneas donde definimos los usuarios y sus contraseñas.

```
<?php
...

// ACCESO USUARIOS (si está vacío funciona sin usuarios)
$usuarios = array();
6 $usuarios["juanjo"]="_IesBalmis1";
  $usuarios["xusa"]="_IesBalmis2";
8 $usuarios["pepe"]="_IesBalmis3";

...
```

Tras ello, entremos donde entremos a nuestro API en `http://localhost/agenda/` nos pedirá que iniciemos sesión con un usuario y contraseña indicándonos como debemos hacerlo con la siguiente pantalla...

API REST en PHP

ESPECIFICACIÓN DE INTERFAZ

Usuarios	Password
juanjo	_IesBalmis1
xusa	_IesBalmis2
pepe	_IesBalmis3

Método	Query String	Descripción
GET	?usu={usuario}&pass={contraseña}	Login
GET	?logout	Logout
GET	?user	Datos del usurio conectado
GET	/datos	Datos de la SESIÓN almacenados en el servidor

Si nos fijamos en la ayuda para autenticarnos necesitamos enviar una petición **GET** a la url **http://localhost/agenda/?usu=<usuario>&pass=<password>** . Por tanto, deberemos definir un nuevo servicio en nuestro APIRest llamado **AutenticacionService** en el paquete **.data.services.autenticacion** que nos permita realizar esta petición y la de cerrar la sesión como mínimo.

```
interface AutenticacionService {
    @GET(".")
    @Headers("Accept: application/json", "Content-Type: application/json")
    suspend fun login(
        @Query("usu") usuario : String,
        @Query("pass") password : String): Response<RespuestaAutenticacionApi>

    @GET(".")
    @Headers("Accept: application/json", "Content-Type: application/json")
    suspend fun logout(
        @Query("logout") usuario : String = ""): Response<RespuestaAutenticacionApi>
}
```

En ambos casos la respuesta del APIRest será un objeto del tipo **RespuestaAutenticacionApi** que definiremos como...

```
data class RespuestaAutenticacionApi(
    val mensaje : String,
    val usuario : String
)
```

y que nos devolverá un mensaje de error o de éxito y el usuario que ha iniciado la sesión o null si no se ha podido iniciar la sesión.

A la hora de realizar la implementación de este servicio de **login**...

```
1 --> GET http://10.0.2.2/agenda/?usu=juanjo&pass=_IesBalmis1
Accept: application/json
Content-Type: application/json
--> END GET
```

deberemos tener en cuenta que en la cabecera de la respuesta del APIRest **se nos devuelve una cookie con el id de la sesión que deberemos almacenar** del algún modo para poder acceder al resto de funcionalidades del APIRest. En nuestro caso el valor de esa cookie se encuentra en la cabecera **Set-Cookie** y es **PHPSESSID=obqu9co5rqhgdocspnqs4n1v5o** . Sin tener en cuenta que podrían llegar otras cookies en en esta misma cabecera.

```
<-- 200 OK http://10.0.2.2/agenda/?usu=juanjo&pass=_IesBalmis1 (18ms)
Date: ...
Server: Apache/2.4.56 (Win64) OpenSSL/1.1.1t PHP/8.2.4
X-Powered-By: PHP/8.2.4
5 Set-Cookie: PHPSESSID=obqu9co5rqhgdocspnqs4n1v5o; path=/
6 Expires: Fecha de expiración de la Cookie
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Content-Length: 59
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: application/json; charset=utf-8
<-- END HTTP
```

Para ello, podemos definir **una propiedad estática pública** en la clase **AutenticacionServiceImplementation** que almacenará la cookie de la sesión.

```

class AutenticacionServiceImpl @Inject constructor(
    private val autenticacionService: AutenticacionService
) {
    private val logTag: String = "OkHttp"

    companion object {
        var cookie: String? = null
    }

    suspend fun login(
        userName : String,
        password : String
    ): RespuestaAutenticacionApi {
        val mensajeError = "Error al loguear a $userName"
        try {
            val response = autenticacionService.login(
                usuario = userName,
                password = password
            )
            if (response.isSuccessful) {
                Log.d(logTag, response.toString())
                val dato = response.body()
                cookie = response.headers().get("Set-Cookie")
                return dato ?: throw ApiServiceException("No hay datos")
            } else {
                ...
            }
        } catch (e: Exception) {...}
    }
}

```

Esta cookie la utilizaremos en el resto de peticiones que realicemos al APIRest. Para ello, deberemos añadir la cabecera `Cookie` en las peticiones que realicemos al APIRest. Por ejemplo, para la petición `http://localhost/agenda/contactos` que devuelve un listado de contactos la petición HTTP debería ser...

```
--> GET http://10.0.2.2/agenda/contactos
Accept: application/json
Content-Type: application/json
4 Cookie: PHPSESSID=obqu9co5rqhgdocspnqs4n1v5o; path=/
--> END GET
```

Para ello, deberíamos modificar la interfaz `ContactoService` y añadir la cabecera `Cookie` en las peticiones que realicemos al APIRest. Por ejemplo, para la petición GET a esa ruta será ...

```
@GET("contactos")
@Headers("Accept: application/json", "Content-Type: application/json")
suspend fun contactos(@Header("Cookie") cookie : String): Response<List<ContactoApi>>
```

Fíjate que hemos añadido la anotación `@Header` para indicar que el valor de la cabecera `Cookie` se obtendrá del parámetro `cookie` que le pasamos al método y por tanto deberemos añadirlo en la llamada al método que hacíamos desde el

`ContactoServiceImpltation` .

El problema de esto es que, como hemos mencionado, **deberemos hacer lo mismo en todas las peticiones de todos servicios a 'endpoints' que hayamos definido** y pasarlo en todas las llamadas a los métodos de los servicios en las implementaciones de los mismos.

Para evitar hacer esto, podemos definir un **interceptor** o ('*hook*') que se ejecutará antes de realizar la petición y que añadirá la cabecera `Cookie` a la petición. Para ello, podemos seguir los siguientes pasos:

1. En el paquete `services` vamos a crear un nuevo paquete llamado `services.interceptors` donde definiremos el interceptor que se encargará de guardar la cookie de la sesión y el de añadirla a las peticiones que se realicen al APIRest.
2. En el paquete `services.interceptors` definiremos una clase llamada `AlmacenDeCookies` guardará en un `HashSet` las cookies que se vayan recibiendo en las respuestas del APIRest. Hilts nos permitirá inyectar un objeto único esta clase en los interceptores que definamos.

```
@Singleton
class AlmacenDeCookies @Inject constructor() {
    private var cookies: HashSet<String>? = null
    fun getCookies(): HashSet<String>? = cookies
    fun setCookies(cookies: HashSet<String>) {
        this.cookies = cookies
    }
}
```

3. En el paquete `services.interceptors` definiremos una clase llamada `ReciveCookiesInterceptor` que implementará el interfaz `Interceptor` y se encargará de **preprocesar cualquier respuesta del servidor y guardar las cookies recibidas** en la cabecera `Set-Cookie` , incluida la de la sesión, en el objeto `AlmacenDeCookies` invalidando el método `intercept` .

```

class ReciveCookiesInterceptor(
    // Inyectamos el objeto AlmacenDeCookies
    private val almacenDeCookies: AlmacenDeCookies
) : Interceptor {

    override fun intercept(chain: Interceptor.Chain): Response {
        // Procesamos la respuesta actual
        val originalResponse = chain.proceed(chain.request())

        // Extraemos de ella las cookies de la cabecera Set-Cookie
        if (originalResponse.headers("Set-Cookie").isNotEmpty()) {
            val cookies = HashSet<String>()
            for (header in originalResponse.headers("Set-Cookie")) {
                cookies.add(header)
            }
            almacenDeCookies.setCookies(cookies)
        }
        // Devolvemos la respuesta original
        return originalResponse
    }
}

```


4. En el paquete `services.interceptors` definiremos una clase llamada `EnviaCookiesInterceptor` que implementará el interfaz `Interceptor` y se encargará de **preprocesar cualquier petición al servidor y añadir las cookies guardadas** en el objeto `AlmacenDeCookies` a la cabecera `Cookie` de la petición, incluida la de la sesión.

```
class EnviaCookiesInterceptor(  
    // Inyectamos el objeto AlmacenDeCookies  
    private val almacenDeCookies: AlmacenDeCookies  
) : Interceptor {  
  
    override fun intercept(chain: Interceptor.Chain): Response {  
        // Builder con el contenido de la petición original  
        val builder = chain.request().newBuilder()  
        val cookies = almacenDeCookies.getCookies()  
        if (cookies != null) {  
            for (cookie in cookies) {  
                // Añadimos las cookies a la cabecera Cookie de la  
                // petición en el builder  
                builder.addHeader("Cookie", cookie)  
            }  
        }  
        // Procesamos la petición con las cookies añadidas  
        return chain.proceed(builder.build())  
    }  
}
```

5. Redefinimos `provideOkHttpClient` inyectando el objeto `AlmacenDeCookies` y añadiendo los interceptores que hemos definido en el builder de nuestro cliente.

```
@Provides
@Singleton
fun provideOkHttpClient(
    4     almacenDeCookies: AlmacenDeCookies
) : OkHttpClient {
    val loggingInterceptor = HttpLoggingInterceptor()
    loggingInterceptor.level = HttpLoggingInterceptor.Level.HEADERS

    val timeout = 10L
    return OkHttpClient.Builder()
    11     .addInterceptor(EnviaCookiesInterceptor(almacenDeCookies))
    .addInterceptor(ReciveCookiesInterceptor(almacenDeCookies))
    // El orden de los interceptores es importante si
    // quiero ver la información de las cookies en el log.
    15     .addInterceptor(loggingInterceptor)
    .connectTimeout(timeout, TimeUnit.SECONDS)
    .readTimeout(timeout, TimeUnit.SECONDS)
    .writeTimeout(timeout, TimeUnit.SECONDS)
    .build()
}
```

Tras esto `.data/services/autenticacion/AutenticacionService.kt` quedará igual que antes, pero su implementación en `.data/services/autenticacion/AutenticacionServiceImplementation.kt` ya no necesitará quedarse con la cookie de la sesión...

```

class AutenticacionServiceImplementation @Inject constructor(
    private val autenticacionService: AutenticacionService
) {
    private val logTag: String = "OkHttp"

    suspend fun login(
        userName : String,
        password : String
    ): RespuestaAutenticacionApi {
        val mensajeError = "Error al loguear a $userName"
        try {
            val response = autenticacionService.login(
                usuario = userName,
                password = password
            )
            if (response.isSuccessful) {
                Log.d(logTag, response.toString())
                val dato = response.body()
                return dato ?: throw ApiServiceException("No hay datos")
            } else {
                val body = response.errorBody()?.string()
                Log.e(logTag, "$mensajeError (código ${response.code()}) : $this\n${body}")
                throw ApiServiceException(mensajeError)
            }
        } catch (e: Exception) {
            Log.e(logTag, "Error: ${e.localizedMessage}")
            throw ApiServiceException(mensajeError)
        }
    }

    suspend fun logout(): RespuestaAutenticacionApi {
        val mensajeError = "Error al cerrar sesión"
        try {
            val response = autenticacionService.logout()
            if (response.isSuccessful) {
                Log.d(logTag, response.toString())
                val dato = response.body()
                return dato ?: throw ApiServiceException("No hay datos")
            } else {
                val body = response.errorBody()?.string()
                Log.e(logTag, "$mensajeError (código ${response.code()}) : $this\n${body}")
                throw ApiServiceException(mensajeError)
            }
        } catch (e: Exception) {
            Log.e(logTag, "Error: ${e.localizedMessage}")
            throw ApiServiceException(mensajeError)
        }
    }
}

```

Además, tras hacer login, ya podremos seguir usándolo nuestro APIRest como lo hacíamos anteriormente pero sin tener que añadir la cabecera **Cookie** en las peticiones que realicemos al APIRest.

Una posible implementación de `./data/AutenticacionRepository.kt` que use la implementación de nuestro servicio podría ser...

```
class AutenticacionRepository @Inject constructor(  
    private val autenticacionService: AutenticacionServiceImplementation  
) {  
    suspend fun login(  
        userName: String,  
        password: String  
    ): Boolean = withContext(Dispatchers.IO) {  
        autenticacionService.login(  
            userName = userName,  
            password = password  
        ).usuario.isNotEmpty()  
    }  
  
    suspend fun logout(): Unit = withContext(Dispatchers.IO) {  
        autenticacionService.logout()  
    }  
}
```

Código de los ejemplos

En el siguiente enlace puedes descargar un proyecto ejemplo con la implementación que acabamos de describir del uso de APIRest en PHP para autenticarnos con la agenda. Para ello, le hemos añadido una pantalla más donde realizar dicha autenticación: [Proyecto ejemplo](#)