

Tema 5.2 - Retrofit

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

Índice

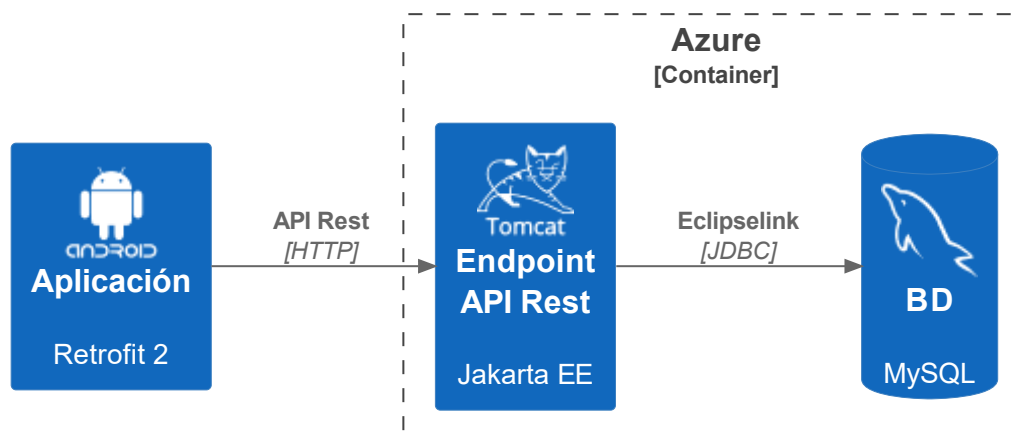
- ▼ Introducción
 - ▼ ¿Qué es JSON?
 - Tipos de datos en JSON
- ▼ Definición de un Servidor Rest rápido para pruebas
 - Creando la Base de Datos con phpMyAdmin
- ▼ Consumo de un Servicio Rest desde Android
 - Configuración del proyecto
 - ▼ Crear los servicios con Retrofit
 - Definiendo los tipos a serializar 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

Introducción

- **Persistencia remota consumiendo un API Rest con Retrofit2**
 - Página oficial librería: [Retrofit](#)
 - Página oficial librería base: [OkHttp](#)
 - 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:



Donde tenemos una aplicación Android que se comunica con un servidor web que tiene un API Rest, que a su vez se comunica con una base de datos MySQL. En este tema nos centraremos en consumo de dicho API Rest desde la aplicación Android utilizando una librería llamada **Retrofit2**.

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 definirlos.
- **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

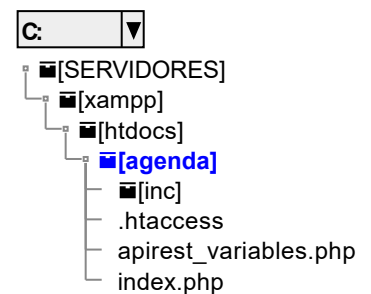
Código de los ejemplos

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";

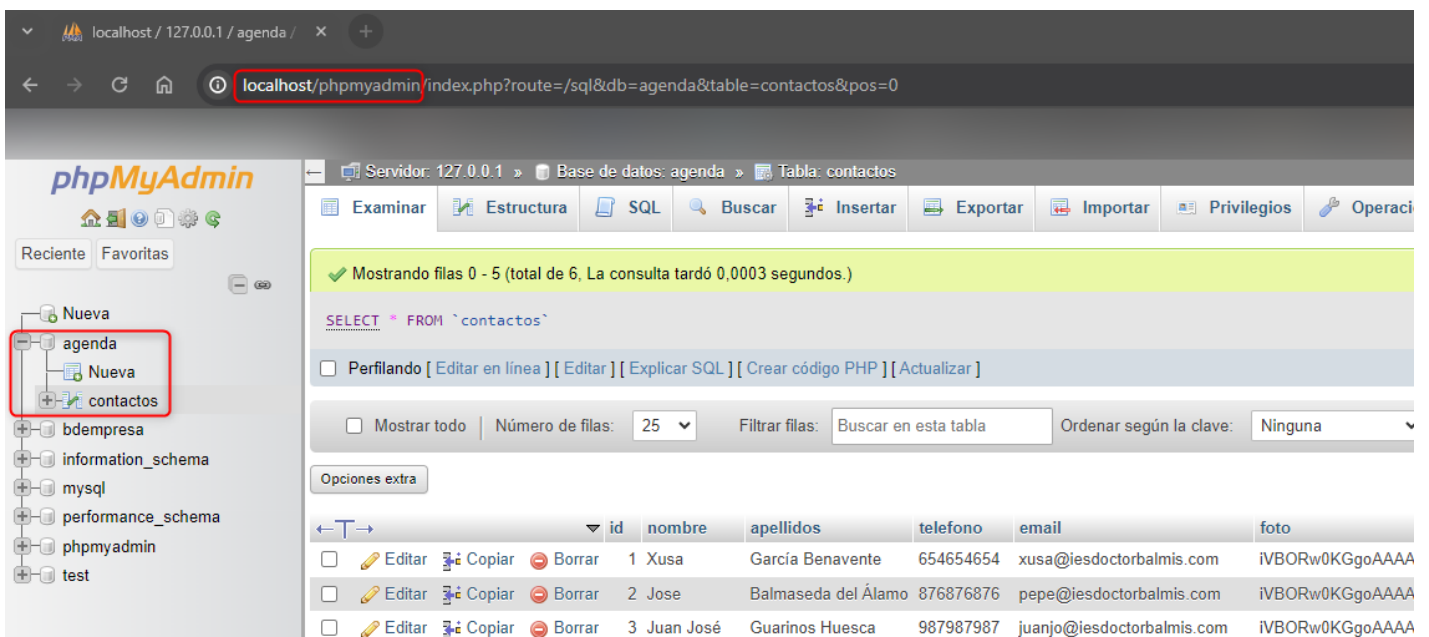
// ACCESO USUARIOS (si está vacío funciona sin usuarios)
$usuarios = array();
// $usuarios["maria"]="m1234";
// $usuarios["juan"]="j1234";

// 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 él encontrará 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).



The screenshot shows the phpMyAdmin web interface. The browser address bar displays `localhost/phpmyadmin/index.php?route=/sql&db=agenda&table=contactos&pos=0`. The left sidebar shows a tree of databases, with 'agenda' selected and its 'contactos' table highlighted. The main panel shows the 'contactos' table structure and data. A message at the top indicates 'Mostrando filas 0 - 5 (total de 6, La consulta tardó 0,0003 segundos.)'. Below this, the SQL query `SELECT * FROM `contactos`` is shown. The table has columns: id, nombre, apellidos, telefono, email, and foto. The data rows are:

	id	nombre	apellidos	telefono	email	foto
<input type="checkbox"/>	1	Xusa	García Benavente	654654654	xusa@iesdoctorbalmis.com	iVBORw0KGgoAAAA
<input type="checkbox"/>	2	Jose	Balmaseda del Álamo	876876876	pepe@iesdoctorbalmis.com	iVBORw0KGgoAAAA
<input type="checkbox"/>	3	Juan José	Guarinos Huesca	987987987	juanjo@iesdoctorbalmis.com	iVBORw0KGgoAAAA

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
<code>contactos</code>	<code>id</code>	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 las siguientes dependencias en el fichero `build.gradle.kts` del módulo de la aplicación:

```
dependencies {  
    ...  
    implementation("com.squareup.retrofit2:converter-gson:2.9.0")  
    implementation("com.squareup.retrofit2:retrofit:2.9.0")  
    implementation("com.squareup.okhttp3:logging-interceptor:4.2.2")  
}
```

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

```
<manifest ...>  
    ...  
    <uses-permission android:name="android.permission.INTERNET"/>  
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>  
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>  
    ...  
</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 de los 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 signature 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 {
    // ...
    // Caso especial del API Rest de PHP que en lugar de
    // devolver un único objeto, devuelve un array con un único objeto
    @GET("contactos/{id}")
    @Headers("Accept: application/json", "Content-Type: application/json")
    suspend fun contacto(@Path("id") id: Int): Response<List<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 ApiServiceException(mensaje: String) : Exception(mensaje)
```

Posteriormente, definiremos para ello una clase denominada **ContactoServiceImplementation** 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...

```

class ContactoServiceImpl @Inject constructor(
    private val contactoService: ContactoService
) {
    4 // Propiedad privada cte. donde definimos el tag para los logs
    5 // 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 {
            10 // Obtenemos la respuesta HTTP Response<List<ContactoApi>>
            val response = contactoService.contactos()
            if (response.isSuccessful) {
                Log.d(logTag, response.toString())
                14 // 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
                19 // error indicando que no hay datos.
                val dato = response.body()
                return dato ?: throw ApiServiceException("No hay datos")
            } else {
                23 // 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
                26 // mensaje generado al Log.
                val body = response.errorBody()?.string()
                Log.e(logTag, "$mensajeError (código ${response.code()}) : $this\n${body}")
                throw ApiServiceException(mensajeError)
            }
        } catch (e: Exception) {
            32 // 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
            35 // producido un TIMEOUT, etc.
            Log.e(logTag, "Error: ${e.localizedMessage}")
            throw ApiServiceException(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 = $id"
    try {
        4      val response = contactoService.contacto(id)
              if (response.isSuccessful) {
                  Log.d(logTag, response.toString())
                  val dato = response.body()
              8      // Puesto que el Api de PHP devuelve una lista con un solo contacto, si todo
                  // ha ido correctamente, devolveremos el primer elemento de la lista.
              10     return dato?.get(0) ?: throw ApiServiceException("No hay datos")
            } else {
                val body = response.errorBody()?.string()
                Log.e(logTag, "$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 {
        // En este método el API de PHP espera recibir el id del contacto
        // que también lo podemos obtener del objeto contacto que le pasamos
        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. Debemos 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.

✦ **Nota:** 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)
    }
}
```