

Apuntes

[Descargar estos apuntes](#)

Tema 11. Persistencia de Datos I

Índice

1. [Acceso a bases de datos locales. SQLite](#)
 1. [Introducción](#)
 2. [Creación y apertura de la BD](#)
 3. [Acceso y modificación de los datos en la BD](#)
 1. [Acceso a la BD mediante sentencias sql](#)
 2. [Acceso a la BD mediante SQLiteDataBase](#)
 3. [Recuperación de datos con rawQuery](#)
 4. [Recuperación de datos con query](#)
2. [Acceso a bases de datos remotas](#)
 1. [Introducción](#)
 2. [Acceso a bases de datos con ApiRest y Retrofit](#)
 1. [Definición del Servicio Rest](#)
 2. [Consumo de un Servicio Rest desde Android](#)

Acceso a bases de datos locales. SQLite

Introducción

La plataforma Android proporciona dos herramientas principales para el almacenamiento y consulta de datos estructurados: la base de datos **SQLite** y **Content Providers** (lo trabajaremos en temas posteriores).

Vamos a centrarnos en **SQLite**, aunque no entraremos ni en el diseño de BBDD relacionales ni en el uso avanzado de la BBDD. Para conocer toda la funcionalidad de SQLite se recomienda usar la documentación oficial.

Para el acceso a las bases de datos tendremos tres clases que tenemos que conocer. La clase **SQLiteOpenHelper**, que encapsula todas las funciones de creación de la base de datos y versiones de la misma. La clase **SQLiteDatabase**, que incorpora la gestión de tablas y datos. Y por último la clase **Cursor**, que usaremos para movernos por el recordset que nos devuelve una consulta *SELECT*.

Creación y apertura de la BD

El método recomendado para la creación de la base de datos es extender la clase **SQLiteOpenHelper** y sobrescribir los métodos **onCreate** y **onUpgrade**. El primer método se utiliza para crear la base de datos por primera vez y el segundo para actualizaciones de la misma. Si la base de datos ya existe y su versión actual coincide con la solicitada, se realizará la conexión a ella.

Para ejecutar la sentencia SQL utilizaremos el método **execSQL** proporcionado por la API para SQLite de Android.

```
internal inner class BDClientes(  
    context: Context?,  
    name: String?,  
    factory: CursorFactory?,  
    version: Int) : SQLiteOpenHelper(context, name, factory, version)  
{  
    var sentencia =  
        "create table if not exists clientes" +  
        "(dni VARCHAR PRIMARY KEY NOT NULL, nombre TEXT, apellidos TEXT);"  
    override fun onCreate(sqliteDatabase: SQLiteDatabase) {  
        sqliteDatabase.execSQL(sentencia)  
    }  
    override fun onUpgrade(sqliteDatabase: SQLiteDatabase, i: Int, i2: Int) {}  
}
```

✎ Si la base de datos existe pero su versión actual es anterior a la solicitada, se llamará automáticamente al método `onUpgrade()`. Si la base de datos no existe se llama automáticamente a `onCreate()`.

Acceso y modificación de los datos en la BD

Una vez creada la clase, instanciaremos un objeto del tipo creado y usaremos uno de los dos siguientes métodos para tener acceso a la gestión de datos:

`getWritableDatabase()` para acceso de escritura y `getReadableDatabase()` para acceso de solo lectura. En ambos casos se devolverá un objeto de la clase `SQLiteDatabase` para el acceso a los datos.

Para realizar una codificación limpia y reutilizable, lo mejor es crear una clase con todos los métodos que necesitemos para trabajar con la Base de Datos, además es útil tener la clase derivada de `SQLiteOpenHelper` interna a esta.

```
class BDAAdapter(context: Context?)
{
    private var clientes: BDClientes

    init {
        clientes = BDClientes(context, "BDClientes", null, 1)
    }

    ...

    internal inner class BDClientes(
        context: Context?,
        name: String?,
        factory: CursorFactory?,
        version: Int) : SQLiteOpenHelper(context, name, factory, version)
    {
        var sentencia =
            "create table if not exists clientes" +
            "(dni VARCHAR PRIMARY KEY NOT NULL, nombre TEXT, apellidos TEXT);"
        override fun onCreate(sqliteDatabase: SQLiteDatabase) {
            sqliteDatabase.execSQL(sentencia)
        }
        override fun onUpgrade(sqliteDatabase: SQLiteDatabase, i: Int, i2: Int)
    }
}
```

✎ Como podemos ver en el código, creamos un objeto del tipo de `BDClientes` con el que vamos a trabajar llamando al constructor `BDClientes()`, a este método le pasamos el contexto, el nombre de la BBDD, un objeto cursor (con valor null) y la versión de la BD que necesitamos.

Para poder usar los métodos creados en la clase, tan solo instanciaremos un objeto de esta pasándole el contexto (de tipo activity). Con esta instancia podremos llamar a cualquiera de los métodos que crearemos en la clase de utilidad.

```
var bdAdapter = BDAdapter(this)
bdAdapter.insertarDatos()
```

Acceso a la BD mediante sentencias sql

El acceso a la BBDD se hace en dos fases, primero se consigue un objeto del tipo **SQLiteDatabase** y posteriormente usamos sus funciones para gestionar los datos. Si hemos abierto correctamente la base de datos entonces podremos empezar con las inserciones, actualizaciones, borrado, etc. No deberemos olvidar cerrar el acceso a la BD, siempre y cuando no se esté usando un cursor.

*Por ejemplo, vamos a insertar 10 registros en nuestra tabla clientes. Para ello podremos crear un método en el **BDAdapter** parecido al siguiente:*

```
fun insertarDatos() {
    var dbClientes = clientes.writableDatabase
    if (dbClientes != null) {
        for (i in 0..9) {
            val sentencia = "INSERT INTO Clientes (dni, nombre, apellidos) V
                            " ('" + i + "', 'nombre" + i + "', 'apellido" + i + "');"
            dbClientes.execSQL(sentencia)
        }
        dbClientes.close()
    }
}
```

👉 Vale, ¿y ahora qué? ¿Dónde está la base de datos que acabamos de crear? ¿Cómo podemos comprobar que todo ha ido bien y que los registros se han insertado correctamente?, en primer lugar veamos dónde se ha creado nuestra base de datos. Todas las bases de datos SQLite creadas por aplicaciones Android se almacenan en la memoria del teléfono en un fichero con el mismo nombre de la base de datos situado en una ruta que sigue el siguiente patrón: **/data/data/paquete.java.de.la.aplicacion/databases/nombre_base_datos**

En el caso de nuestro ejemplo, la base de datos se almacenaría por tanto en la ruta siguiente: **/data/data/ejemplo.basedatos/databases/DBClientes**

Para comprobar que los datos se han grabado podemos acceder de forma remota al emulador a través de su consola de comandos (shell). Pero es mas sencillo abrir el archivo creado con un bloc de notas y echar un vistazo a su contenido.

Acceso a la BD mediante SQLiteDatabase

La API de Android nos ofrece dos métodos para acceder a los datos. El primero de ellos ya lo hemos visto, se trata de ejecutar sentencias SQL a través de **execSql**. Este método tiene como parámetro una cadena con cualquier instrucción SQL válida. La otra forma es utilizar los métodos específicos **insert()**, **update()** y **delete()** de la clase **SQLiteDatabase**.

Veamos a continuación cada uno de estos métodos.

- **insert()** : recibe tres parámetros `insert(table, nullColumnHack, values)` , el primero es el nombre de la tabla, el segundo se utiliza en caso de que necesitemos insertar valores nulos en la tabla "nullColumnHack" en este caso lo dejaremos pasar ya que no lo vamos a usar y por lo tanto lo ponemos a null y el tercero son los valores del registro a insertar. Los valores a insertar los pasaremos a su vez como elementos de una colección de tipo **ContentValues** . Estos elementos se almacenan como parejas **clave-valor**, donde la clave será el nombre de cada campo y el valor será el dato que se va a insertar.
- **update()** : Prácticamente es igual que el anterior método pero con la excepción de que aquí estamos usando el método `update(table, values, whereClause, whereArgs)` para actualizar/modificar registros de nuestra tabla. Este método nos pide el nombre de la tabla "table", los valores a modificar/actualizar "values" (ContentValues), una condición WHERE "whereClause" que nos sirve para indicarle que valor queremos que actualice y como último parámetro "whereArgs" podemos pasarle los valores nuevos a insertar, en este caso no lo vamos a necesitar por lo tanto lo ponemos a null.
- **delete()** : el método `delete(table, whereClause, whereArgs)` nos pide el nombre de la tabla "table", el registro a borrar "whereClause" que tomaremos como referencia su id y como último parámetro "whereArgs" los valores a borrar.

Insertando con ContentValues

```
val dbClientes = clientes.writableDatabase
val valores = ContentValues()
valores.put("nombre", "Xavi")
valores.put("dni", "22222111")
valores.put("apellidos", "Perez Rico")
dbClientes.update("clientes", valores, "dni=4", null)
dbClientes.close()
```

Borrando con ContentValues

```
val dbClientes = clientes.writableDatabase
if (dbClientes != null) {
    val valores = ContentValues()
    valores.put("nombre", cliente.nombre)
    valores.put("dni", cliente.dni)
    valores.put("apellidos", cliente.apellidos)
    dbClientes.insert("clientes", null, valores)
    dbClientes.close()
}
```


Modificando con ContentValues

```
val dbClientes = clientes.writableDatabase
val arg = arrayOf("1")
dbClientes.delete("clientes", "dni=?", arg)
dbClientes.close()
```

Los valores que hemos dejado anteriormente como null son realmente argumentos que podemos utilizar en la sentencia SQL. Veámoslo con un ejemplo:

```
val dbClientes = clientes.writableDatabase
val valores = ContentValues()
valores.put("nombre", "Carla")
val arg = arrayOf("6", "7")
dbClientes.update("clientes", valores, "dni=? OR dni=?", arg)
dbClientes.close()
```

Donde las `?` indican los emplazamientos de los argumentos.

 **Crea un ejercicio EjercicioResueltoBD en el que pruebes el código visto anteriormente: Creación de la BD, insertar elementos con SQL y modificar, eliminar e insertar mediante ContentValues.**

Recuperación de datos con.rawQuery

Existen dos formas de recuperar información (SELECT de la base de datos), aunque ambas se apoyan en el concepto de **cursor**, que es en definitiva el objeto que recoge los resultados de la consulta. Vamos a completar el ejercicio anterior, para que nos permita ir insertando registros en la BD y visualizándolos en un listView.

La primera forma de obtener datos es utilizar el método `rawQuery()` de la clase `SQLiteDatabase`. Este método recibe como parámetro la sentencia SQL, donde se

indican los campos a recuperar y los criterios de selección.

EjemploBD

dni

nombre

apellidos

ANADIR

MOSTRAR

nombre5	apellido5	5
nombre0	apellido0	0
nombre3	apellido3	3
nombre1	apellido1	1

```
fun seleccionarDatosSelect(sentencia: String?): Boolean {  
    val listaCliente: ArrayList<Clientes>?  
    val dbClientes = clientes.readableDatabase  
    if (dbClientes != null) {  
        val cursor: Cursor = dbClientes.rawQuery(sentencia, null)  
        listaCliente = getClientes(cursor)  
        dbClientes.close()  
        return if (listaCliente == null) false else true  
    }  
    return false  
}
```

Evidentemente, ahora tendremos que recorrer el cursor y visualizar los datos devueltos. Para ello se dispone de los métodos `moveToFirst()`, `moveToNext()`, `moveToLast()`, `moveToPrevious()`, `isFirst()` y `isLast()`.

Existen métodos específicos para la recuperación de datos `getXXX(indice)`, donde XXX indica el tipo de dato (String, Blob, Float,...) y el parámetro índice permite recuperar la columna indicada en el mismo, teniendo en cuenta que comienza en 0.

El método `getClientes` (implementado por nosotros) es el que nos permite leer los datos existentes en la BD y llevarlos al `arrayList`:

```

fun getClientes(cursor: Cursor): ArrayList<Clientes>? {
    val clientes: ArrayList<Clientes>
    var cliente: Clientes
    cursor.moveToFirst()
    if (!cursor.isAfterLast()) {
        clientes = ArrayList()
        while (!cursor.isAfterLast()) {
            cliente =
                Clientes(cursor.getString(0), cursor.getString(1), cursor.getStrin
            clientes.add(cliente)
            cursor.moveToNext()
        }
        return clientes
    }
    return null
}

```

Recuperación de datos con query

La segunda forma de recuperar información de la BD es utilizar el método `query()`. Recibe como parámetros el nombre de la tabla, un string con los campos a recuperar, un string donde especificar las condiciones del WHERE, otro para los argumentos si los hubiera, otro para el GROUP BY, otro para HAVING y finalmente otro para ORDER BY.


```

fun seleccionarDatosCodigo(
    columnas: Array<String?>?,
    where: String?,
    valores: Array<String?>?,
    orderBy: String?): ArrayList<Clientes>?
{
    var listaCliente: ArrayList<Clientes>? = ArrayList()
    val dbClientes = clientes.readableDatabase
    if (dbClientes != null) {
        val cursor: Cursor =
            dbClientes.query("clientes", columnas, where, valores, null, null,
            listaCliente = getClientes(cursor)
            dbClientes.close()
            return listaCliente
        }
    }
    return null
}

```

Donde la llamada al método podría ser la siguiente:


```
val listaCliente = bdAdapter.seleccionarDatosCodigo(  
    arrayOf("dni", "nombre", "apellidos"),  
    null,  
    null,  
    "apellidos")
```

 Crea un ejercicio ejemploBD en el que pruebes el código visto anteriormente: Recuperación de los datos con rawQuery y con query. Para ello crea la aplicación como en la imagen, de forma que con un botón se guarde la información en la BD y con el otro se extraiga y se muestre en una lista o en un recycler. Si vais a utilizar una lista como se muestra en la imagen, debereis crear una clase adaptador como la de la siguiente imagen y asociarla a la lista con el setAdapter:

```
binding.listView.setAdapter(  
    AdaptadorClientes(  
        this@MainActivity,  
        R.layout.list_layout,  
        listaCliente!! ))
```

Clase Adaptador para un ListView:

```

class AdaptadorClientes(var activitycontext: Activity,
                           resource: Int,
                           objects: ArrayList<Clientes>
                           ) :
    ArrayAdapter<Any>(activitycontext, resource, objects as List<Any>) {
    var objects: ArrayList<Clientes>
    override fun getView(position: Int,
                           convertView: View?,
                           parent: ViewGroup): View {
        var vista: View? = convertView
        if (vista == null) {
            val inflater = activitycontext.layoutInflater
            vista = inflater.inflate(R.layout.list_layout, null)
            (vista?.findViewById(R.id.apellidolist) as TextView)
                .setText(objects[position].apellidos)
            (vista?.findViewById(R.id.nombrelist) as TextView)
                .setText(objects[position].nombre)
            (vista?.findViewById(R.id.dnilist) as TextView)
                .setText(objects[position].dni)
        }
        return vista
    }

    init {
        this.objects = objects
    }
}

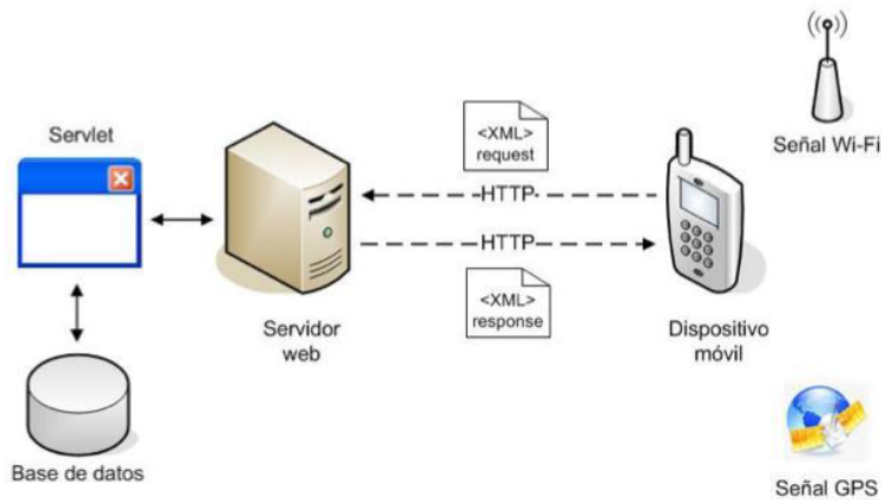
```

Ejercicio Propuesto Recorrer BD

Acceso a bases de datos remotas

Introducción

Para obtener y guardar información de una base de datos remota, es necesario conectarse a un servidor donde se encontrará la BBDD. El esquema de funcionamiento lo podemos ver en la siguiente imagen:



En el servidor funcionan en realidad tres componentes básicos:

- Una base de datos, que almacena toda la información de los usuarios. Para la BBDD se puede utilizar MySQL, que es de licencia gratuita.



- Un servlet , que atiende la petición recibida, la procesa y envía la respuesta correspondiente. Un servlet no es más que un componente Java, generalmente pequeño e independiente de la plataforma.
- Un servidor web , donde reside y se ejecuta el servlet , y que permanece a la espera de conexiones HTTP entrantes.

Los formatos más utilizados para compartir información mediante estos servicios web son XML (y otros derivados) y JSON .

¿Qué es JSON?

JSON (Javascript Objec t 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 de programación como PHP Python C++ C# Java y Ruby.

XML también puede usarse para el intercambio, pero debido a que su definición

genera un DOM , el parseo se vuelve extenso y pesado. Además de ello XML debe usar Xpath para especificar rutas de elementos y atributos, por lo que demora la reconstrucción de la petición. En cambio JSON no requiere restricciones adicionales, simplemente se obtiene el texto plano y el engine de Javascript en los navegadores hace el trabajo de parsing sin ninguna complicación.

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, objetos y valores null. 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.

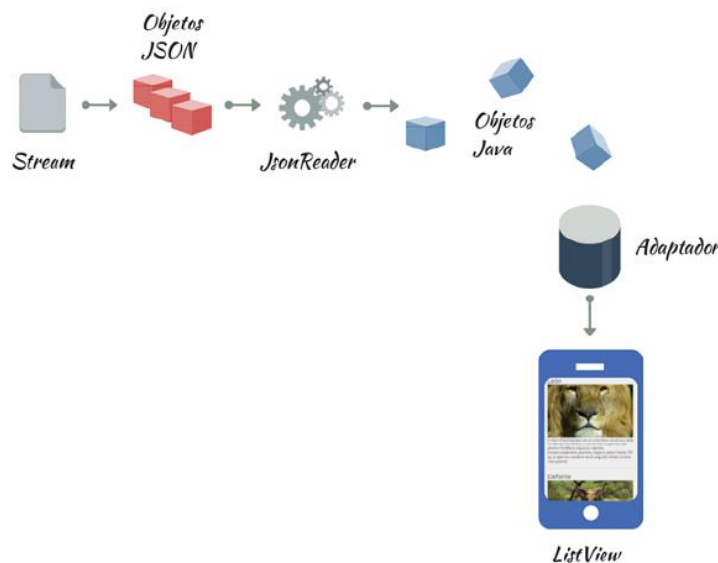
Si notas, este ejemplo trae un ejemplo de cada tipo de dato:

- El *Id* es de tipo entero, ya que contiene un número que representa el código del estudiante.
- El *Nombre* es un string. Usa comillas dobles para identificarlas.
- *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 Java para integrar correctamente el aspecto en la interfaz de usuario.

La clase **JsonObject** de la librería **org.json.JSONObject**, puede interpretar datos con formato JSON y parsearlos a objetos Java o a la inversa.

Veamos una ilustración que muestra el proceso de parseo que será estudiado:



- Como puedes observar el origen de los datos es un servidor externo o hosting que hayas contratado como proveedor para tus servicios web. La aplicación web que realiza la gestión de encriptación de los datos a formato **JSON** puede ser **PHP**, **JavaScript**, **ASP.NET**, etc..
- Tu aplicación Android a través de un cliente realiza una petición a la dirección URL del recurso con el fin de obtener los datos. Ese flujo entrante debe interpretarse con ayuda de un parser personalizado que implementaran las clases que se utilizan para trabajar con **JSON**.
- El resultado final es un conjunto de datos adaptable al **API** de Android. Dependiendo de tus necesidades, puedes convertirlos en una lista de objetos estructurados que alimenten un adaptador que pueble un **ListView** o simplemente actualizar la base de datos local de tu aplicación en **SQLite**.

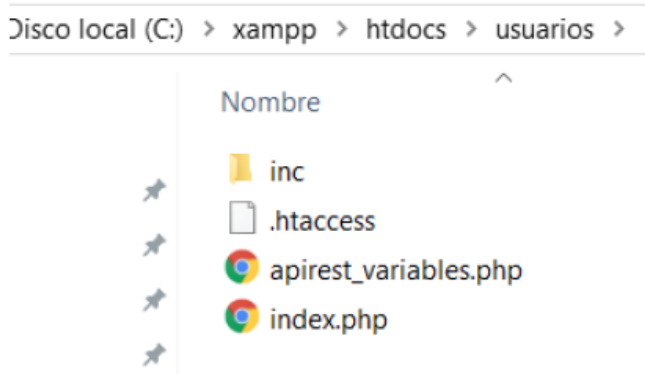
Acceso a bases de datos con Apirest y Retrofit

Retrofit es un cliente *REST* para Android y Java, desarrollado por Square, muy simple y fácil de aprender. Permite hacer peticiones **GET**, **POST**, **PUT**, **PATCH**, **DELETE** y **HEAD**; gestionar diferentes tipos de parámetros y parsear automáticamente la respuesta a un POJO (Plain Old Java Object). Vamos a utilizar esta librería por su facilidad de manejo. P

Definición del Servicio Rest

Como es de suponer, para poder acceder a un servicio Rest desde una de nuestras aplicaciones, este debe de haber sido creado con anterioridad y alojado en un Hosting adecuado.

La parte de creación ya sea con PHP, Java, o cualquiera de los otros lenguajes que lo permiten no corresponde a nuestra asignatura, por lo que la pasaremos por alto y usaremos un ApiRest general que se ha proporcionado en el módulo de Acceso a Datos y que con unas pequeñas modificaciones es valido para la mayoría de casos. La carpeta con el ApiRest está compuesta de los siguientes archivos:



La configuración de nuestro API consta de dos archivos:

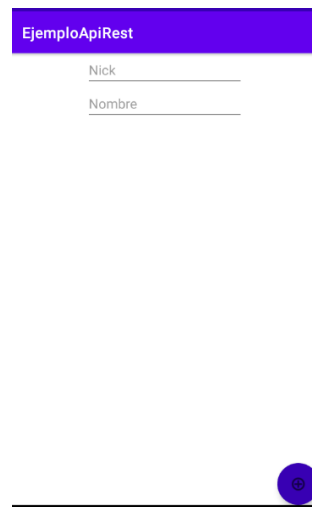
1. **.htaccess** En este archivo se configuran las reglas de acceso, sobre una ruta que le indiquemos en RewriteBase (aquí es donde tendremos que añadir la carpeta en la que habremos alojado nuestra API, comenzando y acabando con `/`).
2. **apirest_variables.php**, en este archivo se definen los datos de conexión a la base de datos. se indican las tablas que tiene esta y el nombre del identificador de cada una de ellas.

<p style="text-align: center;">.htaccess</p> <pre>RewriteEngine On RewriteBase "/usuarios/" RewriteRule ^([a-zA-Z_-]*)\$ index.php?action=\$1&{QUERY_STRING} RewriteRule ^([a-zA-Z_-]*)/([a-zA-Z0-9_-]*)\$ index.php?action=\$1&value=\$2&{Q RewriteRule ^([a-zA-Z_-]*)/([a-zA-Z_-]*)/([a-zA-Z0-9_-]*)\$ index.php?action=\$1</pre>	<p style="text-align: center;">apirest_variables.php</p> <pre><?php // CONFIGURACIÓN BASE DE DATOS MYSQL \$servername = "127.0.0.1"; \$username = "root"; \$password = ""; // BASE DE DATOS \$dbname = "usuariosmensajes"; // TABLAS Y SU CLAVE \$tablas = array(); \$tablas["mensajes"]="_id"; \$tablas["usuarios"]="_id";</pre>
--	--

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. Tendremos que crear la BD y alojar el ApiRest de forma local o en la nube, usando los conocimientos que se tienen del módulo de Acceso a Datos.

Vamos a suponer un ejemplo muy sencillo de una base de datos Usuarios en el que tendremos solamente una tabla Usuarios con dos campos de tipo String (nick y nombre). La BD la habremos construido en el servidor con antelación (en este caso con tabla usuarios de tres campos, nick y nombre de tipo cadena y _id de tipo

numérico autoincrementable como clave). También crearemos una aplicación con dos campos de texto que nos permita insertar los datos del usuario y un botón flotante de añadir, como se ve en la imagen siguiente:



Consumo de un Servicio Rest desde Android

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 y procesar las respuestas del APIRest, mientras que con Gson transformaremos los datos de JSON a los propios que utilice la aplicación.

Para ello añadiremos las siguientes líneas en el build.gradle de la app, y no olvides incluir permisos de internet:

```
implementation "com.squareup.retrofit2:converter-gson:2.9.0"
implementation 'com.squareup.retrofit2:retrofit:2.9.0'
```

Podemos decir que los pasos a seguir serán los siguientes:

1. Creación de un builder de Retrofit. Para poder utilizar Retrofit necesitas crear un builder que te servirá para comunicarte con la API elegida. En la imagen se ha utilizado un método para encapsular el código, aunque no es necesario.

```
private fun crearRetrofit(): ProveedorSersvicio {
    //val url = "http://10.0.2.2/usuarios/" //para el AVD de android
    val url="http://xusa.iesdoctorbalmis.info/usuarios/" //para servidor del ins
    val retrofit = Retrofit.Builder()
        .baseUrl(url)
        .addConverterFactory(GsonConverterFactory.create())
        .build()
    return retrofit.create(ProveedorSersvicio::class.java)
}
```

Si te fijas en la segunda línea verás que hay que escribir la url base de la API a la que harás las peticiones. La obtienes de la documentación proporcionada por el creador de la API.

En la siguiente línea hay que indicar la forma de convertir los objetos de la API a los de tu aplicación y viceversa, aquí es donde especificas que vas a utilizar Gson.

Este builder lo necesitarás para hacer las llamadas a la API, así que procura que sea accesible.

2. Creación de las clases Pojo que le servirá al Gson para parsear los resultados.

Pero primero necesitas conocer la estructura del Json que va a devolverte la API, ya que no hay un patrón establecido. Para conocer la estructura previamente, se puede utilizar PostMan y realizar las diferentes peticiones (GET, POST, PUT, etc) desde este.

Existen aplicaciones o webs que facilitan la creación de la clase Pojo a partir de un JSON, como por ejemplo: [<http://www.jsonschema2pojo.org/>]

```
class RespuestaJSON {
    var respuesta = 0
    var metodo: String? = null
    var tabla: String? = null
    var mensaje: String? = null
    var sqlQuery: String? = null
    var sqlError: String? = null
}
```

Después tienes que crear la estructura de clases para almacenar la información que te resulte útil. Clase Usuario en este ejemplo.

```
class Usuarios(var nick: String, var nombre: String, var _id: Int =0)
```

3. Otro elemento imprescindible, es la gestión de los servicios que se quieran utilizar. Para cada uno de ellos se tendrá que hacer una petición a la API. Necesitaremos crear una interfaz con todos los servicios que quieras utilizar. Aquí tienes unos ejemplos:


```

interface ProveedorServicio {
    @GET("usuarios")
    @Headers("Accept: application/json", "Content-Type: application/json")
    suspend fun usuarios(): Response<List<Usuarios>>

    @GET("mensajes")
    @Headers("Accept: application/json", "Content-Type: application/json")
    suspend fun mensajes(): Response<List<Mensaje>>

    @GET("mensajes/{nick}")
    @Headers("Accept: application/json", "Content-Type: application/json")
    suspend fun getUsuario(@Path("nick") nick: String): Response<List<Usuarios>>

    @POST("usuarios")
    @Headers("Accept: application/json", "Content-Type: application/json")
    suspend fun insertarUsuario(@Body usuarios: Usuarios): Response<RespuestaJSon>

    @POST("mensajes")
    @Headers("Accept: application/json", "Content-Type: application/json")
    suspend fun insertarMensaje(@Body mensaje: Mensaje): Response<RespuestaJSon>
}

```

En el código de arriba hay dos servicios dos son de tipo **GET** sin palabra clave y que servirán para obtener todos los usuarios y todos los mensajes. Luego se tiene otro servicio **GET** con palabra clave (Nick) que servirá para seleccionar los mensajes de un determinado Nick. Y luego dos **POST** a los que se les pasa en el Body los datos a añadir.

Usando **Corrutinas** se realizará la llamada a la API, por lo que los métodos de la interfaz deben ser de tipo **suspend**. En cada método se indica el tipo de dato que espera obtener, en estos ejemplos la respuesta puede ser o una lista del tipo de objeto de la petición o un tipo RespuestaJSon, que tendrá todos los posibles miembros que nos podría dar alguna de las llamadas invocadas.

Las palabras seguidas del símbolo **@** dan funcionalidad a los servicios, haciéndolos dinámicos y reutilizables, para más información [<http://square.github.io/retrofit/>]

💡 Las que gestiona **Retrofit** para invocar la url de la petición son

@GET, **@POST**, **@PUT** y **@DELETE**. El parámetro corresponderá con la url de la petición.

En el builder de Retrofit se incluye la url base del api terminada con **/**, que unida con el parámetro del servicio crearán la url completa de la petición:

@GET -> http://>10.0.2.2/usuarios/usuario

- **@Header** y **@Headers**. Se usan para especificar los valores que vayan en la sección *header* de la petición, como por ejemplo en que formato van a ser

enviados y recibidos los datos.

- **@Path** . Sirve para incluir un identificador en la url de la petición, para obtener información sobre algo específico. El atributo en el método de llamada que sea precedido por **@Path** sustituirá al identificador entre llaves de la ruta, que tenga el mismo nombre.
- **@Fields** . Nos permite pasar variables básicas en las peticiones *Post* y *Put*.
- **@Body** . Es equivalente a Fields pero para variables objeto.
- **@Query** . Se usa cuando la petición va a necesitar parámetros (los valores que van después del **?** en una url).

4. Pedir datos a la Api sería el último paso a realizar. Retrofit nos da la opción de realizarlo de manera síncrona o asíncrona. Vamos a aprovechar nuestros conocimientos de corrutinas para lanzar la petición en segundo plano de forma sencilla.

```
private fun anyadirUsuario(usuarios: Usuarios) {
    val context=this
    var salida:String?
4    val proveedorServicios: ProveedorServicio = crearRetrofit()
5    CoroutineScope(Dispatchers.IO).launch {
        val response = proveedorServicios.insertarUsuario(usuarios)
        if (response.isSuccessful) {
8            val usuariosResponse = response.body()

            if (usuariosResponse != null) salida=usuariosResponse.mensaj
            else salida=response.message()
        }
        else {
            Log.e("Error", response.errorBody().toString())
            salida=response.errorBody().toString()
        }
        withContext(Dispatchers.Main) { Toast.makeText(context, salida,
            if (espera != null) espera?.hide()
            limpiaControl())}
19    }
}
```

📌 **Línea 4** llamamos al método `crearRetrofit`, que es el que nos devuelve un proveedor de servicios del tipo de interface que hemos creado y nos enlazaré con la url del servidor y con el GSON. Línea 5 - 13 se lanza la corrutina con la petición deseada, cuando se obtenga respuesta se filtra para ver si ha sido correcta y recuperar los datos necesarios, **`response.body()`**, en caso contrario se lanza mensaje de error.

🔗 EjercicioPropuestoBDExternas

