Apuntes

Descargar estos apuntes en pdf y html

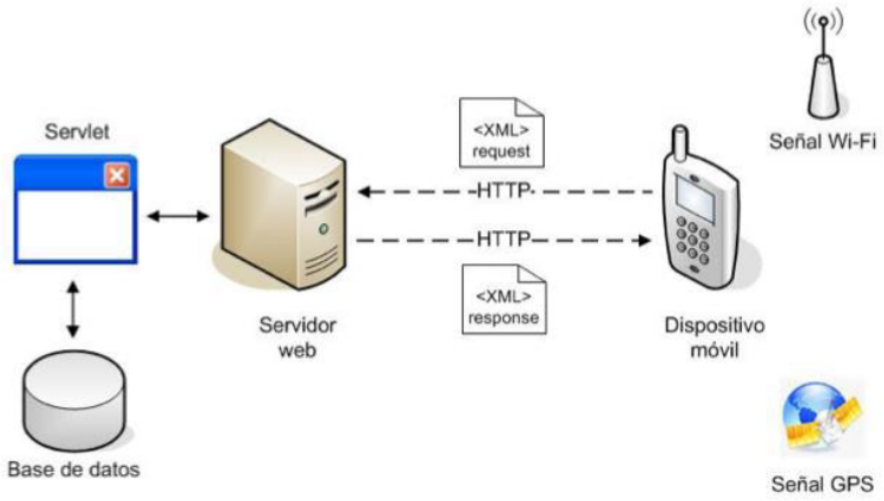
Tema 11.2. Persistencia de Datos II Retrofit

Índice

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]
}
```

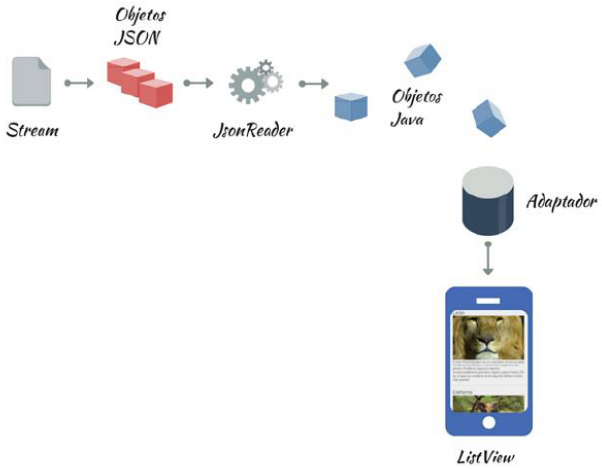
La anterior estructura es un objeto JSON compuesto por los datos de un estudiante. Los objetos JSON contienen sus atributos entre llaves `{}` , al igual que un bloque de código en Javascript, donde cada atributo debe ir separado por coma `,` para diferenciar cada par.

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`.

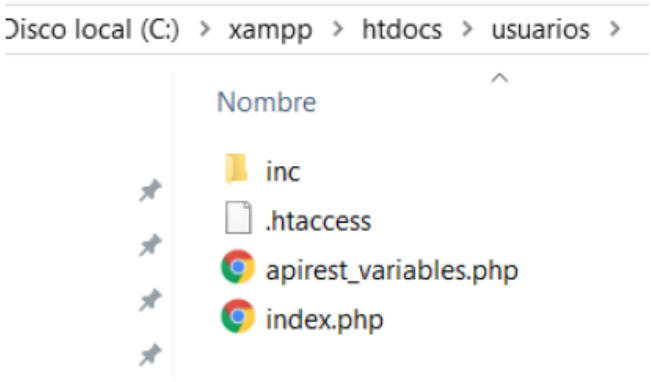
Acceso a bases de datos con Apirest y Retrofit

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.

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 `/`).

```
RewriteEngine On
RewriteBase "/uaurios/"
RewriteRule ^([a-zA-Z0-9_-]*)$ index.php?action=$1&%{QUERY_STRING}
RewriteRule ^([a-zA-Z0-9_-]*)/([a-zA-Z0-9_-]*)$ index.php?action=$1&value=$2&%{QUERY_STRI
RewriteRule ^([a-zA-Z0-9_-]*)/([a-zA-Z0-9_-]*)/([a-zA-Z0-9_-]*) index.php?action=$1&field
```

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

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

    // TABLAS Y SU CLAVE
    $tablas = array();
    $tablas["mensajes"]="_id";
    $tablas["usuarios"]="_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. 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 usarios 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
       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 específicas 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 ProveedorSersvicio {
    @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<Usuari

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

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

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 van 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.

```
1  private fun anyadirUsuario(usuarios: Usuarios) {
2          val context=this
3          var salida:String?
4          val proveedorServicios: ProveedorSersvicio = crearRetrofit()
5
6          CoroutineScope(Dispatchers.IO).launch {
7              val response = proveedorServicios.insertarUsuario(usuarios
8              if (response.isSuccessful) {
9                  val usuariosResponse = response.body()
10                 if (usuariosResponse != null) salida = usuariosRespons
11                 else salida=response.message()            }
12             else {
13                 Log.e("Error", response.errorBody().toString())
14                 salida=response.errorBody().toString()
15             }
16             withContext(Dispatchers.Main) {
17                 Toast.makeText(context, salida, Toast.LENGTH_LONG).sho
18                 if (espera != null) espera?.hide()
19                 limpiaControl()
20             }
21         }
22     }
```

📌 **Nota: 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 enlazará con la url del servidor y con el GSon. **Líneas 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

EjercicioPropuestoAgenda