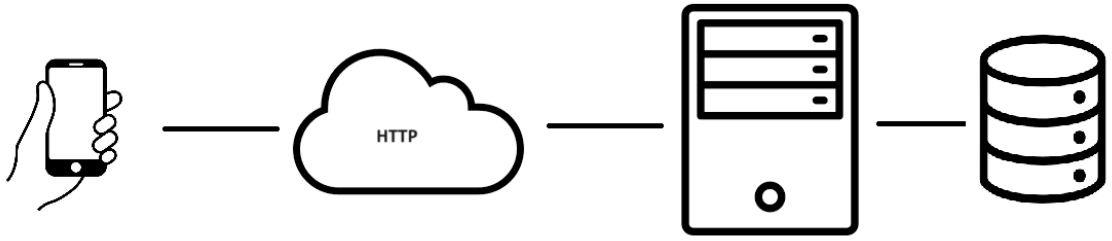
Web Services y Retrofit

Una de las utilidades más obvias y utilizadas en las App móviles consiste en alimentarlas con información obtenida de los **Web Services** (Servicios Web). Por ejemplo, podemos disponer de un servidor que haga de intermediario de una Base de Datos de alumnos y notas. Un alumno enciende su app móvil, solicita ver el resultado de un examen, esa petición llega al servidor y éste le responde con la nota.

Este documento hace referencia a esta lógica empleando para ello varias tecnologías, como son Retrofit, HTTP, Spring Boot, etc. Ciertos contenidos no pertenecen estrictamente a la asignatura, de forma que se detalla únicamente una base mínima para poder comprender la parte de la App Android.



Los **Web Services** son es una tecnología que utiliza un conjunto de protocolos y estándares que sirven para intercambiar información entre aplicaciones. En líneas generales, pueden describirse como programas que esperan una solicitud de un cliente para responderle con información. Lo habitual es que tengan una Base de Datos detrás y se le conecten programas a través de la red. Por ejemplo, un cliente en una localización remota pide cierta cosa a un Web Service, éste consulte con la Base de Datos, y responda al cliente con el resultado. Su arquitectura suele ser, por tanto, MVC.

Retrofit es un cliente de servidores REST para Android y Java. Permite crear una API de red de forma sencilla que permita recuperar información de un Web Service. Internamente hace cosas verdaderamente complejas, como convertir los tipos de datos, enrutamiento, etc. En pocas palabras, abstrae al programador de las complicaciones del manejo de HTTP, y es capaz de convertir un objeto a formato JSON/XML (y viceversa), que suele ser el formato de respuesta de los Web Services.

Spring Boot es una herramienta para desarrollar aplicaciones web que utiliza el framework Spring. En pocas palabras, nos genera automáticamente un programa que puede funcionar como servidor de Web Services. Podemos customizarlo y programarlo a nuestro gusto añadiéndole funcionalidades. Obviamente, lo que se le suele añadir es lo necesario para acceder a una Base de Datos mediante tecnologías como Hibernate y JTA.

Montando un Servidor [Java]

No vamos a explicar en este documento cómo montar una Api REST. Para ello, consulta el documento **UD4 - Generación de Servicios en Red – 2** de la asignatura de **PSP**. Ahí tienes toda la información necesaria para montar una Api REST con **Spring Boot**.

Montando un Cliente [Kotlin]

Si has seguido los apuntes del apartado anterior, ahora tendrás una Api REST con dos métodos expuestos. Deberían ser funcionales, ya que los habremos probado desde **Postman**. Vamos ahora a crear el cliente usando **Retrofit**.

Nótese que seguiremos añadiendo más endpoints a la Api Rest para completar el ejemplo.

Retocando el Servidor Springboot

Antes de seguir, vamos a retocar un par de cosas del Servidor Springboot. Para empezar, añadiremos un endpoint nuevo y modificaremos otro para evitar que devuelva una cadena de texto, que no tiene sentido.

```
@GetMapping("/alumnos")
public List<Alumno> getAllPotatoes() {
    List<Alumno> ret = null;
    FakeDataBase fake = FakeDataBase.getInstance();
    ret = fake.getAllAlumnos();
    return ret;
}

@GetMapping("/alumno/find/{name}")
public ResponseEntity<Alumno> getByName(@PathVariable String name) {
    Alumno ret = null;
    FakeDataBase fake = FakeDataBase.getInstance();
    List<Alumno> alumnos = fake.getAllAlumnos();
    for (Alumno alumno : alumnos) {
        if (alumno.getNombre().equals(name)) {
            // Found!
            ret = alumno;
            break;
        }
    }
    return ret != null ? ResponseEntity.ok(ret) :
        ResponseEntity.notFound().build();

}

@PostMapping("alumno/new")
public void addAlumno(@RequestBody Alumno alumno) {
    FakeDataBase fake = FakeDataBase.getInstance();
    fake.addAlumno(alumno);
}
```

Configurando mi App: Dependencias

Vamos a lo que nos interesa: la App. Para poder alimentar a nuestra App desde un Servicio Web, necesitamos utilizar **Retrofit**, por lo que tendremos que añadir las dependencias al Gradle:

```
// For Retrofit
implementation("com.squareup.retrofit2:retrofit:3.0.0")
implementation("com.squareup.retrofit2:converter-gson:3.0.0")
implementation("com.google.code.gson:gson:2.13.2")

// If we want coroutines to do asynchronous calls
implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.10.2")
```

Retrofit necesita Gson para poder convertir automáticamente un json en un objeto, y viceversa. Recuerda que la información viaja en el cuerpo de una petición HTTP como un json.

Configurando mi App: Manifest

Nuestra App necesita permisos para acceder a internet, como era de esperar; pero también es necesario incluir otra directiva adicional.

Desde Android 9 (Pie), por **defecto no se permite tráfico HTTP no seguro** (sin SSL) Solo se permiten peticiones **HTTPS** (HTTP cifrado). Por tanto, si tu API Spring Boot está corriendo en http://localhost:8080, el sistema bloqueará la conexión.

La forma de solucionarlo será añadir en nuestra App un **fichero de excepciones**. Primero, le añadimos la línea siguiente al Manifest.xml:

```
android:networkSecurityConfig="@xml/network_security_config"
```

Y luego le añadiremos un fichero llamado **network_security_config.xml** manualmente en **res/xml/**. Tendremos que añadir en él como excepción la IP de nuestro Servidor.

Ten en cuenta que, si el Servidor está en el mismo equipo que el Android Studio, **en lugar de localhost** deberemos utilizar la IP de **loopback** de Android Studio.

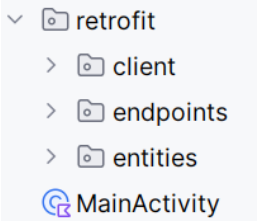
```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config cleartextTrafficPermitted="true">
        <domain includeSubdomains="true">10.0.2.2</domain>
    </domain-config>
</network-security-config>
```

Esto tiene a demás el problema de que permites que circulen json con texto en claro por la red. El cifrado deberías de hacerlo tú manualmente.

Configurando mi App: El código

Ya podemos crear el Servicio que se alimente de nuestra Api Rest. Retrofit necesita de tres clases para funcionar. Un **Interface** por cada Web Service al que se va a acceder, con sus respectivos endpoints; un **Cliente** para generar el acceso al Web Service; y una **Entidad** que describe el objeto que va a recibir o enviar al Servicio Web.

Genera los siguientes paquetes:



Empezamos los la Entidad. Siguiendo nuestro ejemplo, la clase Alumno es la más sencilla de hacer, porque es una copia idéntica de la Entidad Alumno que tiene el Servidor. Así que hacemos:

```
data class Alumno(
    val nombre: String,
    val apellido: String
)
```

Las **Interfaces** describen los **endpoints** de la Api Rest a los que queremos acceder. Habrá un interfaz por cada Web Service diferenciado. En nuestro caso:

```
interface AlumnoInterface {  3 Usages

    @GET( value = "/alumnos")  1 Usage
    fun getAllAlumnos(): Call<List<Alumno>>

    @GET( value = "/alumno/find/{name}")
    fun getByName(@Path( value = "name") name: String): Call<Alumno>

    @POST( value = "/alumno/new")  1 Usage
    fun addAlumno(@Body alumno: Alumno): Call<Void>
}
```

Estos métodos se hacen mirando, literalmente, los endpoints definidos en el servidor. Mira al inicio de este documento.

Nótese que, si un método no devuelve nada, se pone **Void**.

Si te fijas, las anotaciones son similares a las que empleamos en el Servidor. Por ejemplo, el primer método define un acceso a un endpoint:

- **Get**: Se trata de una operación get mapeada como **/alumnos**. Si a este mapeo se la añade la BASE_URL de la Instancia, nos queda la dirección completa del endpoint: **http://10.0.2.2:8080/alumnos**

- **Call <?>**: La llamada al método de Retrofit que envía la llamada y trae la respuesta. Date cuenta que, si no devuelve nada, se debe seguir poniendo.

- En el segundo método, vemos que para enviar un parámetro incluyendo la etiqueta **@Path**. Nótese que usaríamos @Query si estuviésemos usando este sistema en el método de la Api Rest.

- En el tercer método, un Post, vemos que para enviar un objeto se hace incluyendo la etiqueta **@Body**.

Finalmente, nos falta el **Cliente** de Retrofit, que es una instancia única para toda la App (salvo que busques conectarte con varios Servidores). Su formato es similar a éste.

```
object RetrofitClient {
    private val retrofit = Retrofit.Builder()  1 Usage
        .baseUrl( baseUrl = "http://10.0.2.2:8080") // localhost from the Android emulator
        .addConverterFactory( factory = GsonConverterFactory.create())
        .build()

    val alumnoInterface: AlumnoInterface = retrofit.create(AlumnoInterface::class.java)
}
```

Usando Retrofit

Vamos a hacer una llamada sencilla desde un **botón**. La idea es que este botón obtenga todas las patatas de la Base de Datos y las cargue en una tabla. Para ello, primero accedemos a la instancia del servicio:

```
// The Api
val api = RetrofitClient.alumnoInterface
```

Asignamos la petición a un evento de botón, y realizamos la llamada. Por ejemplo, vamos a llamar getAllAlumnos (), que nos retorna de la Api Rest un listado de Alumnos.

```
api.getAllAlumnos().enqueue( callback = object : Callback<List<Alumno>> {

    // This triggers when the Server answers to our call
    override fun onResponse(
        call: Call<List<Alumno>>,
        response: Response<List<Alumno>>
    ) {
        if (response.isSuccessful) {
            // HTTP response code is 200

        } else {
            // HTTP response code is NOT 200. This usually means an error. But NOT ALWAYS!

            // Please, note that you MUST consider the HTTP response code individually!!
        }
    }

    // This triggers when something went wrong. Usually, we couldn't even send the request
    override fun onFailure(call: Call<List<Alumno>>, t: Throwable) {

    }
})
```

Las peticiones se dividen en dos partes:

- onResponse (): que se lanza cuando hay respuesta del Servidor.
- onFailure (): cuando no hay. Normalmente sucede cuando hay fallos de red, etc. la petición ni siquiera ha llegado al Servidor.

Para el onFailure () simplemente mostraremos un Toast informativo. Pero para onResponse (), toca hacer un par de cosas más.

```
override fun onResponse(
    call: Call<List<Alumno>>,
    response: Response<List<Alumno>>
) {
    if (response.isSuccessful) {
        // HTTP response code is 200
        val alumnos = response.body()

        if (alumnos.isNullOrEmpty()) {
            Toast.makeText(
                context = this@MainActivity,
                text = "No hay datos que mostrar",
                duration = Toast.LENGTH_SHORT
            ).show()
        } else {
            val recyclerView = findViewById<RecyclerView>( id = R.id.recyclerView)
            recyclerView.layoutManager = LinearLayoutManager( context = this@MainActivity)
            recyclerView.adapter = AlumnoAdapter( lista = alumnos)
        }
    } else {
        // HTTP response code is NOT 200. This usually means an error. But NOT ALWAYS!
        Toast.makeText(
            context = this@MainActivity,
            text = "Error GET: ${response.code()}",
            duration = Toast.LENGTH_SHORT
        ).show()
        // Please, note that you MUST consider the HTTP response code individually!!
    }
}
```

En este caso, tenemos que diferenciar entre **códigos de respuesta de HTTP**. Nótese que estos códigos son, al fin y al cabo, respuestas válidas del Servidor. Incluso el 404 o el 500. Dependerá de nosotros interpretar los códigos de respuesta.

Obviamente, nosotros cuando diseñemos el Servidor, deberemos incluir estos códigos en nuestras respuestas.

¿Otras operaciones?

Para otro tipo de operaciones, como los POST, el paso de parámetros, tratamiento de códigos de respuesta HTTP, por favor, mira y ejecuta los ejemplos que dispones junto a este documento: **RestServerExample** y **AppRetrofitClientExample**.