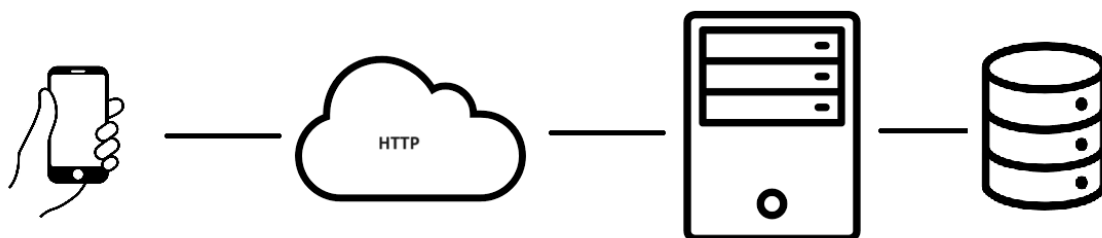


Web Services y Retrofit

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 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 conectan programas a través de la red. Por ejemplo, un cliente en una localización remota pide cierta cosa a un Web Service, éste consulta con la Base de Datos, y responde 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.

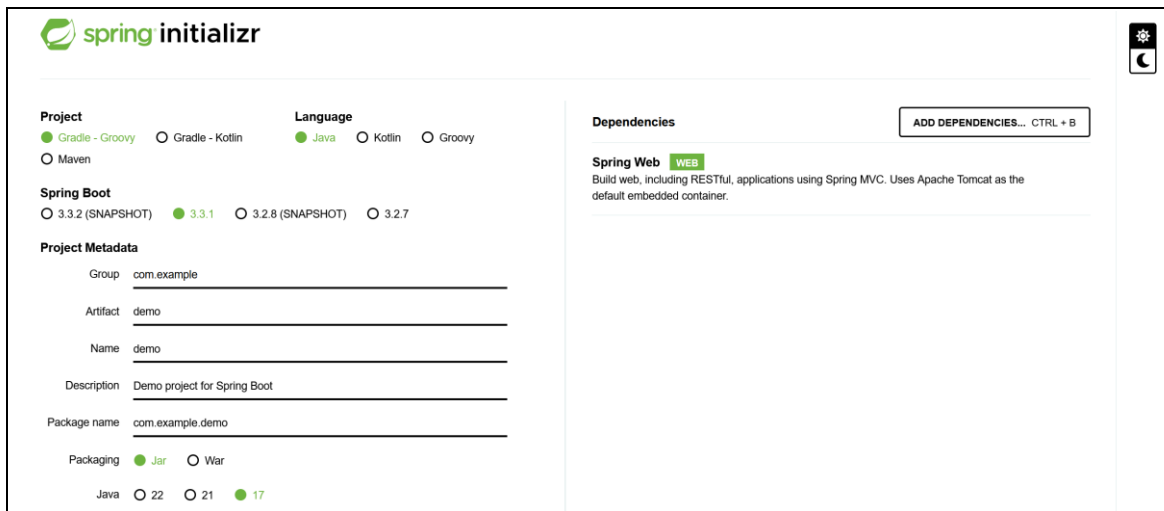
Spring Boot es una herramienta para desarrollar aplicaciones web que utiliza el framework Spring. En pocas palabras, nos genera automáticamente un programa que puede funcionar como servidor de Web Services. Podemos customizarlo y programarlo a nuestro gusto añadiéndole funcionalidades. Obviamente, lo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

Montando el Servidor [Java]

Vamos a montar un pequeño servidor con **Spring Boot** que exponga unos pocos servicios a la red. Probaremos estos servicios desde **Postman**, y después prepararemos otros tantos para ser accesibles desde una App utilizando **Retrofit**.

Spring Initializr [Java]

Accedemos a la página <https://start.spring.io/> donde podremos configurar los parámetros que tendrá nuestro servidor. Una vez hecho, esta página nos permitirá descargar el código fuente del servidor con la configuración que le hayamos puesto. Es una forma sencilla y cómoda de empezar un proyecto, en comparación a hacerlo de forma manual.

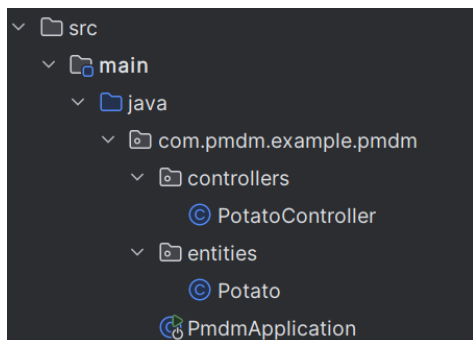


The screenshot shows the Spring Initializr web interface. It includes sections for Project, Language, Spring Boot, Project Metadata, and Dependencies. The Project section has radio buttons for Gradle - Groovy, Gradle - Kotlin, and Maven. The Language section has radio buttons for Java, Kotlin, and Groovy. The Spring Boot section has radio buttons for 3.3.2 (SNAPSHOT), 3.3.1, 3.2.8 (SNAPSHOT), and 3.2.7. The Project Metadata section has input fields for Group (com.example), Artifact (demo), Name (demo), Description (Demo project for Spring Boot), and Package name (com.example.demo). The Packaging section has radio buttons for Jar and War. The Java section has radio buttons for 22, 21, and 17. The Dependencies section has a button to add dependencies and a list of dependencies including Spring Web.

En nuestro caso, podemos utilizar la configuración que ves en la imagen. No te olvides de indicar la dependencia de Spring Web, que incluye las librerías necesarias para los Web Services.

Importación del servidor [Java]

Ahora, despliega el código fuente del servidor en un IDE. Puedes hacerlo en Eclipse o en cualquier otro. En las imágenes que tienes a continuación se está utilizando IntelliJ. Al hacerlo, veremos que tenemos un proyecto en blanco con una única clase localizada en la carpeta **src**.



Asumiendo que hayas llamado al servidor **PmdmApplication**, verás una estructura de carpetas similar a esta.

Las carpetas controllers y entities las añadiremos a continuación, pues son las que contienen la funcionalidad de nuestros Web Services.

No es necesario modificar la clase **PmdmApplication** que contiene el main del proyecto. Si ejecutas el código verás que funciona correctamente, aunque obviamente, no hará nada. Le faltan los **endpoints** de los Web Services. Vuelve a apagar el proyecto.

Los servidores que ofrecen servicios web son pasivos. Esto significa que (sin entrar en detalles) se quedan siempre en espera de que otros programas (los clientes) tomen la iniciativa de comunicarse con ellos. Esto se hace configurando una serie de **métodos** que, si están **correctamente anotados**, serán expuestos al exterior.

Es decir: estamos dejando que **otros programas** llamen a métodos de nuestro programa.

Añadiendo Endpoints [Java]

Genera la carpeta **controllers**. Crea ahí la clase **PotatoController**. Vamos a definir un servicio que va a ofrecer a nuestra App acceso a nuestra Base de Datos de patatas. Información vital en importante.

```
@RestController
public class PotatoController{

    private final List <Potato> potatoList; 10 usages

    public PotatoController(){
        potatoList = new ArrayList<>();
        potatoList.add(getDefaultPotato ());
    }

    private Potato getDefaultPotato (){ 1 usage
        Potato potato = new Potato();
        potato.setId(0);
        potato.setName("Potato");
        potato.setAmount(10);
        return potato;
    }
}
```

Dado que nuestra App va a querer consultar y añadir patatas a la Base de Datos del servidor, vamos a emular una añadiendo una lista de patatas en la que guardarlas. Obviamente, en un proyecto real se accedería a una Base de Datos mediante Hibernate, JTA, etc. pero para nosotros nos basta con esto.

Genera ahora la carpeta **entities** y crea la clase **Potato**. Esta clase es un POJO que vamos a utilizar para enviar información desde la App al servidor y viceversa. Este pojo también podría utilizarse para Base de Datos, pero no es buena idea. Este POJO se va a utilizar para la comunicación entre las capas Vista y Controlador (App y el servidor) de nuestro proyecto, y no para las capas Controlador y Modelo (Servidor y BBDD). Pero a veces se hace, por simplicidad.

```
public class Potato implements Serializable { 14 usages

    @Serial no usages
    private static final long serialVersionUID = 1L;

    public int id = 0; 7 usages
    public String name = null;
    public int amount = 0; 7 usages

    public Potato() {} 1 usage
```

La clase **Potato** tiene la habitual colección de getters & setters, equals, hashCode y toString. Es especialmente importante que añadas el **serialVersionUID**. Cuando intentemos enviar un Potato a la App, para poder serializar y deserializar el objeto Potato sin que de error se miran varias cosas: el nombre de la clase, los atributos... y el **serialVersionUID**. Si no coincide, no funcionará.

Vamos a añadir un **endpoint**. En principio, los endpoints están relacionados con los cuatro tipos de operaciones básicas HTTP: Get, Post, Put y Delete. Que a su vez estarán relacionadas con las operaciones SQL de Select, Insert, Update y Delete. Por tanto, vamos a hacer un método que retorne todas las patatas de la lista (un Select * from t_potatoes)

```
// http://localhost:8080/potatoes/
@GetMapping("/potatoes")
public String getPotatoes (){
    return "List of potatoes: " + potatoList;
}
```

A tener en cuenta:

- **GetMapping** es la anotación que indica que el método es para una operación Get. Entre paréntesis se indica el **path** que referenciará al método.
- El nombre del método es irrelevante, sólo vale para no liarnos nosotros.
- En este caso, retornamos un String, pero porque es un ejemplo. Normalmente se devuelven objetos de la base de Datos (como veremos más adelante).
- La **url** que ves arriba en el comentario es la que tendrá nuestro endpoint una vez arranques el servidor. Puedes probarlo desde tu navegador web.

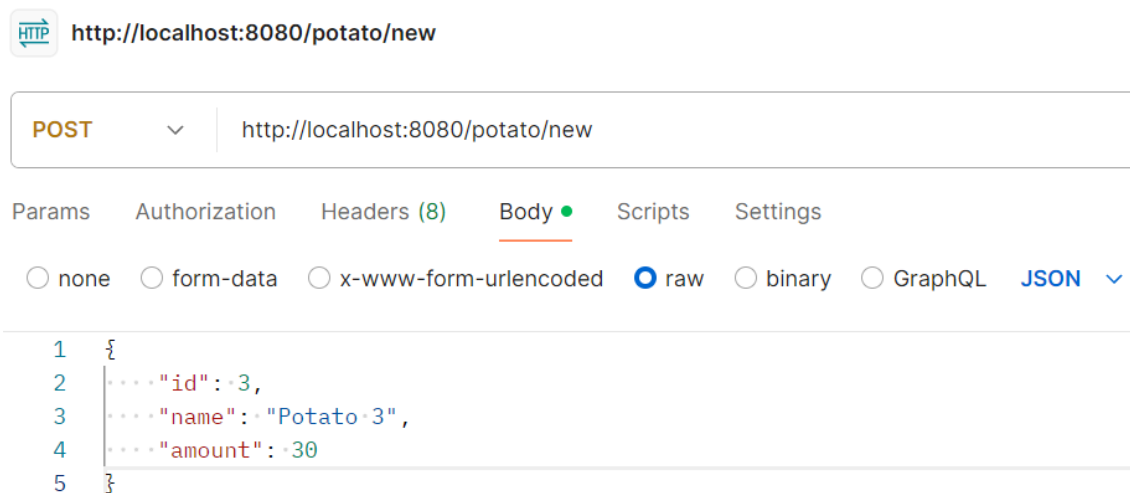
Vamos a añadir un segundo **endpoint**. En este caso, es un endpoint que añade una patata a la lista ((un Insert into t_potatoes)

```
// http://localhost:8080/potato/new
@PostMapping("potato/new")
public String addPotato (@RequestBody Potato potato){
    potatoList.add(potato);
    return "Potato added: " + potato.toString();
}
```

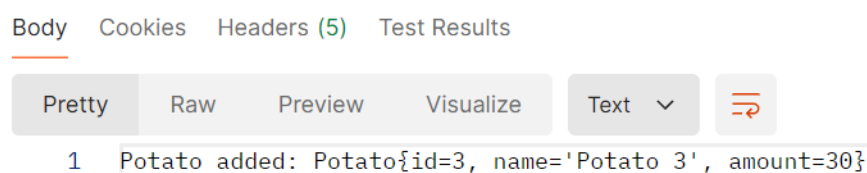
A tener en cuenta:

- **PostMapping** es la anotación que indica que el método es para una operación Post. Entre paréntesis se indica el **path** que referenciará al método.
- El nombre del método es irrelevante, sólo vale para no liarnos nosotros.
- En este caso, retornamos un String, pero el Post de Http no tiene cuerpo de respuesta, por lo tanto, no es habitual que un Post devuelva algo.
- La **url** que ves arriba en el comentario es la que tendrá nuestro endpoint una vez arranques el

Desde el Postman, veremos cómo responde el endpoint. Lanzamos la petición...



... y obtenemos la respuesta.



Añadiendo Endpoints para la App [Java]

Vamos a añadir ahora los métodos para nuestra App. Dado que (se supone) que estamos trabajando contra Base de Datos, con lo que trabajamos es con objetos. Y obviamente, no se trabaja desde navegador, por lo tanto no tiene sentido responde con cadenas de texto.

```
// http://localhost:8080/potato/getAll
@GetMapping("/potato/getAll")
public List<Potato> getAllPotatoes (){
    return new ArrayList<>(potatoList);
}

// http://localhost:8080/potato/getById/
@GetMapping("/potato/getById/")
public Potato getPotatoById (int id){
    Potato ret = null;
    for (Potato potato : potatoList) {
        if (potato.getId() == id) {
            ret = potato;
            break;
        }
    }
    return ret;
}

// http://localhost:8080/potato/addNewP/
@PostMapping("/potato/addNewP/")
public void addPNewPotato (@RequestBody Potato potato){
    potatoList.add(potato);
}
```

Estos endpoint también pueden probarse desde **Postman**. Lo habitual en los Servicios Web reales que ofrecen accesos a Bases de Datos es que exista una clase controller por cada tabla, y en ella, uno o más endpoint por cada operación contra dicha tabla. Normalmente, se utilizan interfaces, abstracción y herencia para definirlos. Pero esto es una generalización, y cada proyecto se diseña de forma independiente.

Adicionalmente, ten en cuenta que esta parte es un resumen de un resumen del contenido de otras asignaturas. Se ha omitido mucha información como, por ejemplo, el token utilizado para mantener sesiones, cifrado HTTPS, y la forma en la que un servidor toma la iniciativa y envía información al cliente.

Montando el Cliente [Java]

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

```
// For Retrofit
implementation(libs.retrofit)
implementation(libs.converter.gson)
implementation(libs.picasso2.okhttp3.downloader)
```

Como ves, necesitamos también el GSON para convertir un objeto a formato JSON y viceversa, y okhttp 3, que es el cliente HTTP.

Vamos ahora al Manifest. Le damos acceso a internet, y debido a que vamos a trabajar con HTTP en lugar de HTTPS, tendremos que configurar en nuestra App un fichero de excepciones. Android Studio desaconseja usar HTTP al carecer de seguridad. Añadimos la línea:

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

Añadiremos un fichero **network_security_config.xml** manualmente en **res/xml/** y lo editamos. Tendremos que añadir como excepción la IP de nuestro servidor. En nuestro caso, dado que está en el mismo equipo que el Android Studio, en lugar de localhost deberemos utilizar la IP de loopback de Android Studio cuando usamos el emulador, que es:

```
<?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>
```

Configurando Retrofit [Java]

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

- ▼ retrofit
 - ▼ endpoints
 - 📄 PotatoesInterface
 - ▼ entities
 - 📄 Potato
 - ▼ instances
 - 📄 PotatoInstance

La clase Potato es la más sencilla de hacer, porque es una **copia idéntica** del POJO que tiene el Servidor y que hemos creado antes en este documento. Si se cambia aquel, se cambia éste. No hay más que decir.

La **Instancia** es única para toda la App a menos que quieras conectarte con varias máquinas diferentes. Su formato es estándar, similar a éste.

```
public class PotatoInstance { 2 usages

    private static Retrofit retrofit = null; 3 usages
    private static final String BASE_URL = "http://10.0.2.2:8080"; 1 usage

    public static Retrofit getRetrofitInstance() { 1 usage
        if (retrofit == null) {
            retrofit = new retrofit2.Retrofit.Builder()
                .baseUrl(BASE_URL)
                .addConverterFactory(GsonConverterFactory.create())
                .build();
        }
        return retrofit;
    }
}
```

Nótese que BASE_URL es la IP del servidor donde se encuentran los Web Services a los que deseamos acceder. ¡Cuidado con poner / de más!

Por último, preparamos la **Interfaz** que describe los endpoints a los que queremos acceder. Habrá un interfaz por cada Web Service.

```
public interface PotatoesInterface { 3 usages

    @GET("/potato/getAll") 1 usage
    Call<List<Potato>> getAllPotatoes();

    @POST("/potato/addNewP/") 1 usage
    Call <Void> addPNewPotato(@Body Potato potato);

}
```

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

- **Get:** Se trata de una operación get mapeada como **/potato/getAll**. 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/potato/getAll**
- **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, un Post, vemos que para enviar un objeto se hace incluyendo la etiqueta **@Body**.

Usando Retrofit [Java]

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 generamos la instancia del servicio de tipo PotatoesInterface.

```
// The service
service = null == service?
    PotatoInstance.getRetrofitInstance().create(PotatoesInterface.class) :
    service;
```

Y después lanzamos la petición:

```
Call<List<Potato>> call = service.getAllPotatoes();
call.enqueue(new Callback<List<Potato>>() {

    @Override 2 usages
    public void onResponse(@NonNull Call<List<Potato>> call, @NonNull Response<List<Potato>> response) {

        // Get the potatoes
        List<Potato> potatoes = response.body();

        // Let the adapter do the magic and display the potatoes
        // Add some code yourself to do that, man...
    }

    @Override
    public void onFailure(@NonNull Call<List<Potato>> call, @NonNull Throwable t) {
        Toast.makeText(context: MainActivity.this, text: "Something went wrong...", Toast.LENGTH_SHORT).show();
    }
});
```

La llamada se realiza inmediatamente, pero la respuesta se procesa mediante dos eventos separados. El **onResponse** se va a ejecutar únicamente si la llamada es exitosa, mientras que **onFailure** si es errónea.

No obstante, esto es una simplificación. En realidad, el **onResponse** se lanza cuando la petición ha ido bien, es decir, el servidor ha hecho su trabajo y ha generado una respuesta. Sin embargo, recuerda que HTTP genera códigos de respuesta diferentes y que 404 o 500, pese a ser ‘error’, **siguen siendo respuestas correctas**. En principio, sólo se ejecuta **onFailure** en dos situaciones diferentes: cuando la red falla, o cuando en el servidor hay una excepción sin capturar. Menuda metedura de pata, por cierto.

Para diferenciar el código HTTP de respuesta puede hacerse así:

```
if (response.code() == 400){
    // Do something
} else {
    // Do something else
}
```