

Módulo 8 – Backend con Spring Boot

Índice

Framework Spring Boot

¿Qué es Spring Boot?	2
¿Qué es Maven?	3
Crear un Proyecto en Spring Boot	4
Patrón MVC en Spring Boot.....	10

API Con Spring Boot

API REST con Spring Boot.....	12
Creamos una API	14
Más sobre @annotation.....	20

Probando la API

¿Qué es Postman?.....	22
Respuestas con @ResponseBody	23

Aplicando buenas prácticas de diseño

Patrón DTO en Spring	26
Arquitectura Multicapa	29
Inyección de Dependencias e Inversión de Control	30
Bases de Datos con SpringBoot, JPA e Hibernate	36
ABML con SpringBoot + JPA + Hibernate	39
Probando el ABML con Posman	47

¿Qué es Spring Boot?

Spring framework es un conjunto de proyectos de código abierto desarrollados en Java, con el objetivo de agilizar el desarrollo de aplicaciones en este lenguaje. Todo su pack de aplicaciones es conocido como Spring platform e incluye herramientas para el desarrollo web, microservicios, manejo de base de datos, seguridad, entre otros. Algunas de las características de Spring Platform pueden ser vistas en la siguiente imagen:

¿Qué puede hacer Spring?



Cada una de estas características de Spring fueron desarrolladas en una serie de proyectos independientes, donde cada uno es utilizado para diferentes fines. Entre los principales proyectos se encuentran:

- **Spring Boot:** Facilita la creación y configuración inicial de proyectos de Spring para generar aplicaciones de fácil y rápida puesta en marcha.
- **Spring Data:** Utilizado para la administración, manejo y comunicación con bases de datos, tanto relacionales como no-relacionales.
- **Spring Security:** Utilizado para las cuestiones de seguridad que puede necesitar todo proyecto.
- **Spring Web Services:** Utilizado para facilitar el desarrollo de Web Services SOAP.

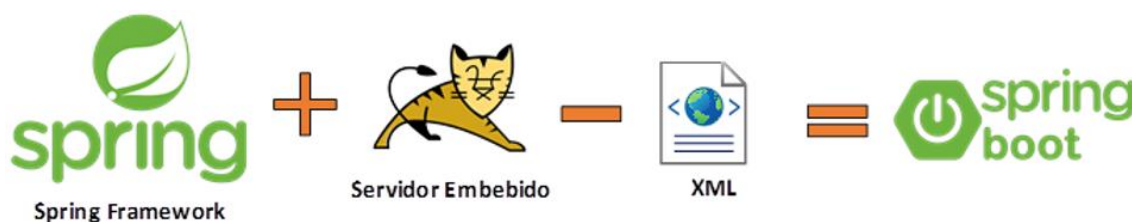
La lista completa de todos los proyectos de Spring está disponible en: <https://spring.io/projects>, ¿Sabías que Netflix usa este Spring?

Spring Framework vs Spring Boot

Spring Boot es una extensión de Spring Framework y se encuentra dentro de su lista de proyectos. Fue creado con la finalidad de facilitar la creación de aplicaciones web listas para salir a producción, es decir, bajo el concepto “Just Run” (solo ejecutar).

Anteriormente, realizar las configuraciones iniciales para llevar a cabo una aplicación en Spring llevaba mucho tiempo a los desarrolladores. Esto, se realizaba mediante una configuración manual de un archivo xml y de un servidor de aplicaciones web, consumiendo gran parte del tiempo de desarrollo del proyecto en realizar configuraciones. Dada esta problemática y con la finalidad de resolverla, fue desarrollado Spring Boot, que requiere una configuración mínima y que puede ser integrado con otros proyectos de Spring o librerías externas. En la siguiente Ilustración se pueden observar las características incorporadas y quitadas de Spring Framework que lograron la implementación de Spring Boot.

Veamos en la imagen los componentes de Spring Boot:



¿Qué es Maven?

Maven es una herramienta de software para la gestión y construcción de proyectos Java que se caracteriza por tener un modelo de configuración muy simple, basado en el formato XML.

Maven utiliza el conocido archivo POM.xml (Project Object Model) para dentro de él especificar las diferentes dependencias o librerías que serán necesarias incluir en el proyecto que se esté desarrollando. A partir de esta definición, Maven se encarga de buscar todas las dependencias especificadas en las versiones correspondientes y que sean compatibles entre sí para el desarrollo de la aplicación en cuestión, ahorrando gran cantidad de tiempo en lo que respecta a la organización de los complementos necesarios.

Muchos entornos de desarrollo (IDE) ya tienen incorporado un motor como Maven para permitir la sincronización y descarga rápida de los complementos necesarios. Algunos ejemplos son Apache Netbeans o también IntelliJ Idea.

Maven vs Gradle

Gradle es una herramienta muy similar a Maven, sin embargo, se caracteriza por ser utilizada principalmente para procesos de automatización de compilación. Se basa en conceptos similares a Maven, pero con la particularidad de que fue diseñado principalmente para realizar trabajos multiproyecto o que requieran de un gran grado de personalización.

Entre algunas de sus principales características se encuentran:

1. Flexibilidad.
2. Integración entre varios proyectos.
3. Gestión de dependencias.
4. Utiliza DSL.
5. Groovy.
6. Soporta varios lenguajes (no solo Java).

A continuación, se puede visualizar una pequeña tabla comparativa entre Gradle y Maven:

Gradle	Maven
Utiliza un lenguaje específico de dominio (DSL)	Utiliza XML
Se basa en la tarea mediante la cual se realiza el trabajo	En Maven se definen objetivos vinculados al proyecto
Admite compilaciones incrementales de la clase java	No admite compilaciones incrementales
Pensado para mayor nivel de detalle y personalización de los proyectos	Pensado para cuando existe poca personalización

Conociendo un poco las diferencias entre ambos, cabe destacar que en los ejercicios y ejemplos que veremos a continuación utilizaremos Maven. Esto se debe a que los proyectos que serán desarrollados cumplirán con los estándares y sin mucho nivel de personalización.

¿Con cuál trabajaremos?

Nosotros nos enfocaremos en Maven para las prácticas y para el proyecto.

Crear un Proyecto en Spring Boot

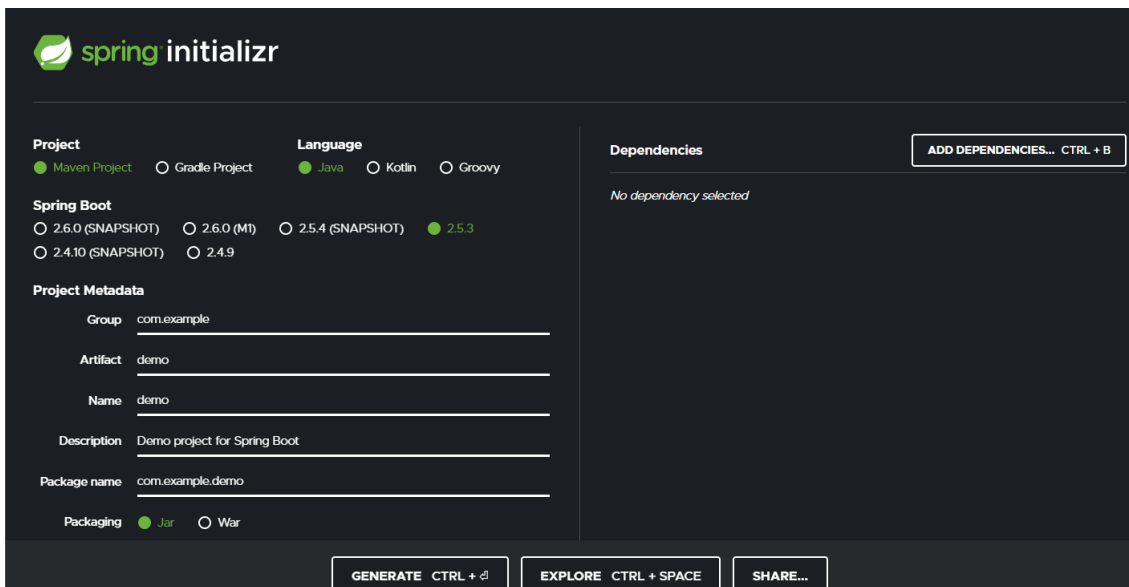
¿Por dónde comenzamos?

Para facilitar la creación de proyectos de forma genérica y que éstos puedan ser levantados desde cualquier IDE, Spring Boot ofrece una herramienta conocida como Initializr.

Para crear un proyecto con Initializr es necesario llevar a cabo una serie de pasos que los vamos a especificar a continuación:

PASO 1

Ingresar a la web oficial de initializr: <https://start.spring.io/> en donde visualizaremos un apartado similar al siguiente:

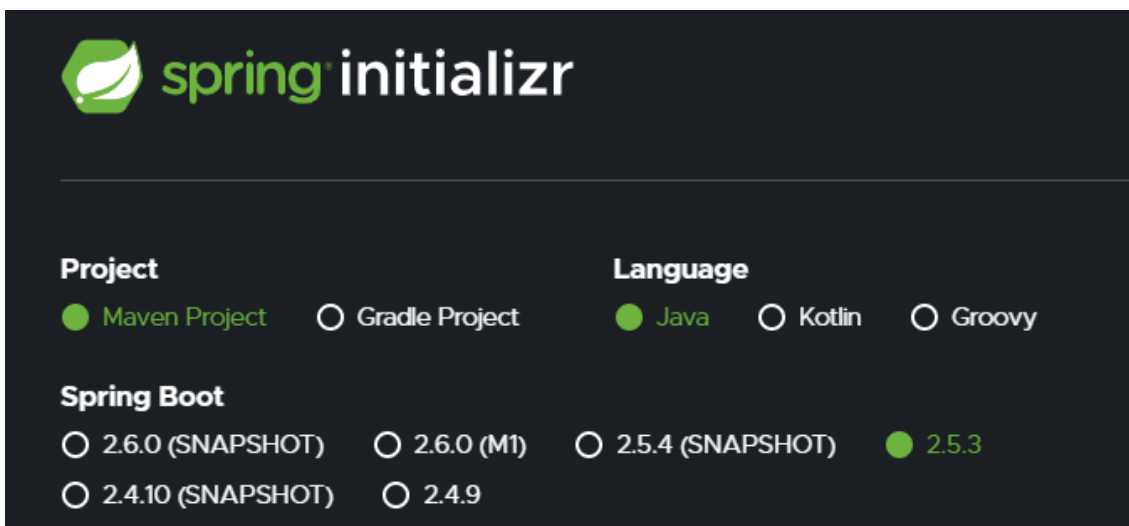


The image shows the Spring Initializr web interface. It has a dark theme with the Spring logo and 'spring initializr' text at the top. The interface is divided into several sections: 'Project' with radio buttons for 'Maven Project' (selected) and 'Gradle Project'; 'Language' with radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'; 'Spring Boot' with radio buttons for versions '2.6.0 (SNAPSHOT)', '2.6.0 (M1)', '2.5.4 (SNAPSHOT)', '2.5.3' (selected), '2.4.10 (SNAPSHOT)', and '2.4.9'; 'Project Metadata' with input fields for 'Group' (com.example), 'Artifact' (demo), 'Name' (demo), 'Description' (Demo project for Spring Boot), and 'Package name' (com.example.demo); and 'Packaging' with radio buttons for 'Jar' (selected) and 'War'. A 'Dependencies' section on the right says 'No dependency selected' with a button 'ADD DEPENDENCIES... CTRL + B'. At the bottom are three buttons: 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'.

PASO 2

Una vez en la web, vamos a seleccionar una serie de parámetros, entre ellos: el tipo de proyecto, el lenguaje, la versión de Spring Boot, las configuraciones de los datos del proyecto y las dependencias (librerías, etc) que estarán asociadas.

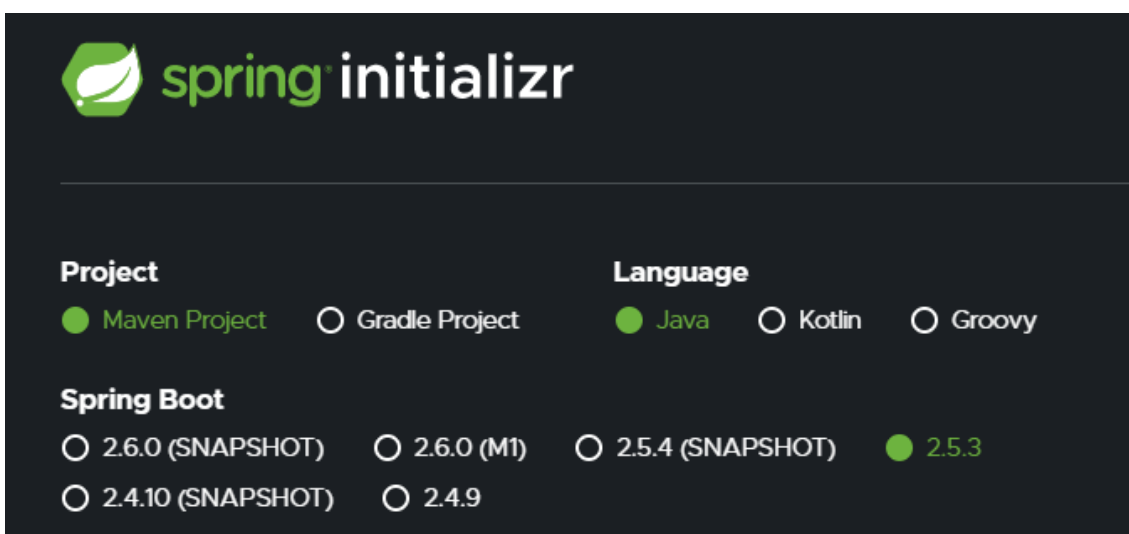
Para el tipo de proyecto seleccionaremos Maven, como lenguaje Java y como versión de Spring Boot la más estable actualmente que nos sugiere Spring Initializr (en este caso la 2.5.3), tal como puede verse a continuación:



This image shows the same Spring Initializr web interface as before, but with the selections made in Step 2. 'Maven Project' is selected under Project, 'Java' is selected under Language, and '2.5.3' is selected under Spring Boot. The other fields and buttons remain the same.

PASO 3

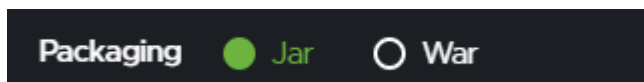
Debemos especificar una serie de detalles para el proyecto a crear: En primer lugar tenemos el “grupo” (o group), el mismo suele coincidir con la url de una dirección web pero de forma inversa, es decir, por ejemplo, si vamos a desarrollar una aplicación cuya url va a ser “miaplicacion.com”, el group va a ser “com.miaplicacion”. Además de esto establecemos los nombres de artefacto y del proyecto. Como resultado, se obtiene el nombre del paquete principal del proyecto que es la concatenación de los apartados anteriores. También es posible establecer una descripción para el proyecto con la finalidad de especificar en mayor detalle sobre qué tratará el mismo. te mostramos algunas configuraciones de ejemplo:



The screenshot shows the Spring Initializr configuration interface. At the top is the 'spring initializr' logo. Below it, there are two main sections: 'Project' and 'Language'. In the 'Project' section, 'Maven Project' is selected with a green radio button, while 'Gradle Project' is unselected. In the 'Language' section, 'Java' is selected with a green radio button, while 'Kotlin' and 'Groovy' are unselected. Below these sections is the 'Spring Boot' section, which contains several version options: '2.6.0 (SNAPSHOT)', '2.6.0 (M1)', '2.5.4 (SNAPSHOT)', '2.5.3', '2.4.10 (SNAPSHOT)', and '2.4.9'. The '2.5.3' version is selected with a green radio button.

PASO 4

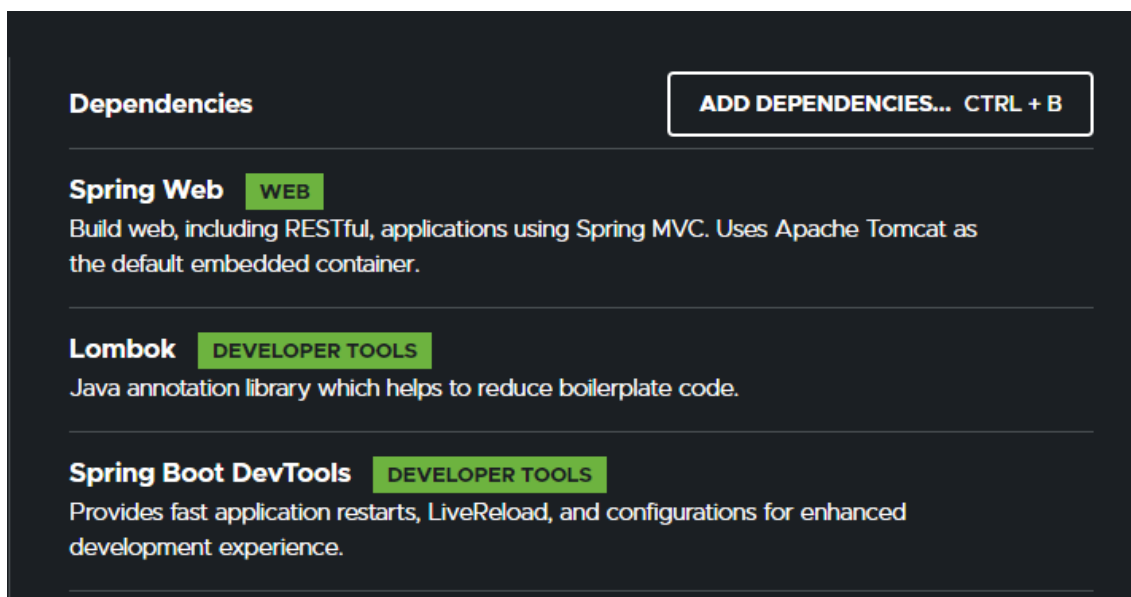
Es necesario seleccionar el tipo de empaquetado que tendrá la aplicación. En este caso para el ejemplo a desarrollar seleccionaremos “Jar”:



The screenshot shows the 'Packaging' section of the Spring Initializr form. It has a dark background with the word 'Packaging' in white. To its right, there are two radio button options: 'Jar' and 'War'. The 'Jar' option is selected, indicated by a green filled circle next to it.

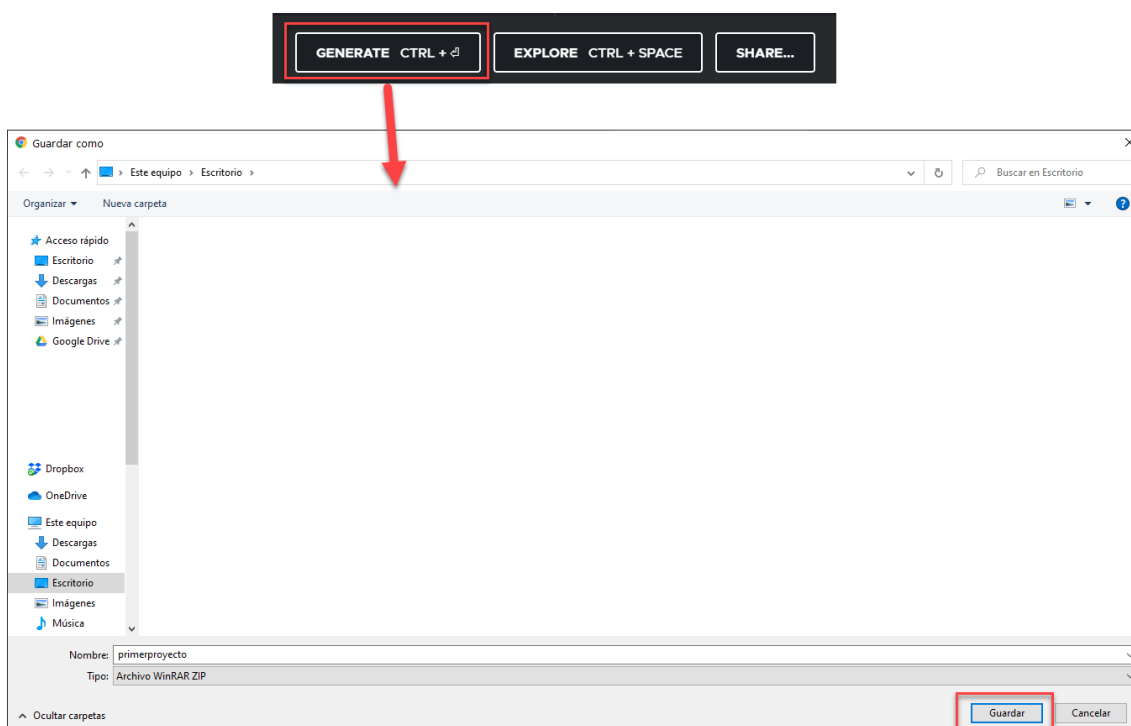
PASO 5

Como última configuración, es necesario seleccionar todas las dependencias / librerías que serán necesarias para el proyecto. Para ello es necesario hacer clic en el botón Add Dependencies y seleccionar las que correspondan. En la siguiente ilustración se muestran algunas de las más importantes para el ambiente web y que utilizaremos para este ejemplo:



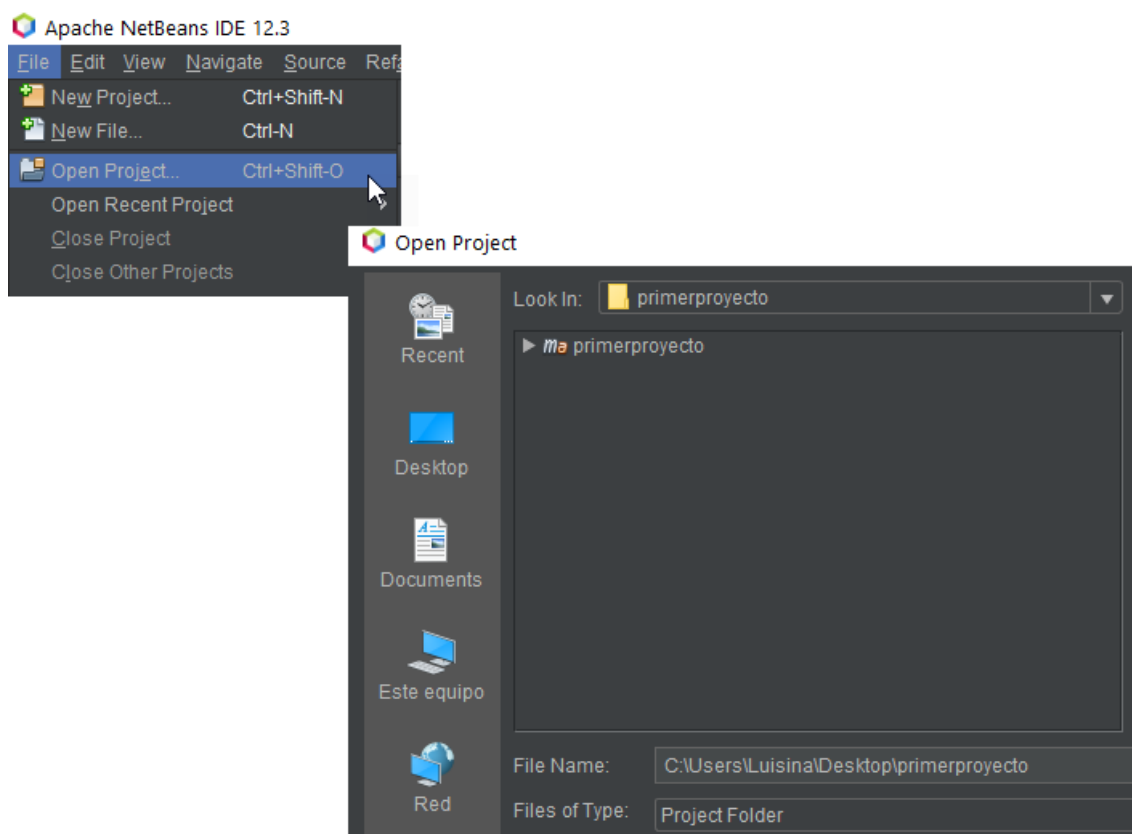
PASO 6

Una vez seleccionadas todas las configuraciones, se debe hacer clic en el botón “Generate”. Esto generará un archivo comprimido en donde se encontrará el proyecto que se podrá levantar en el IDE que utilizemos. Tal como puede visualizarse a continuación:



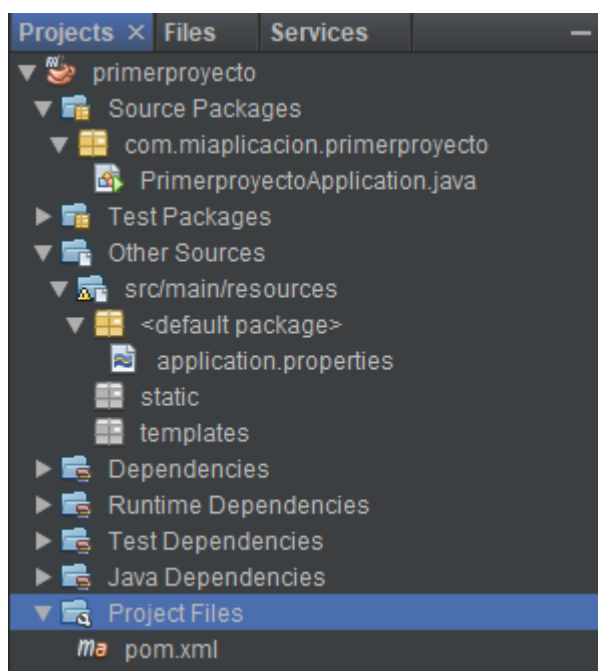
PASO 7

Una vez hecho esto, es necesario descomprimir el proyecto realizado para levantarlo en el IDE que estemos utilizando. En la siguiente ilustración se puede ver un ejemplo de cómo sería el proceso con Apache Netbeans v 12 (LTS):



Una vez realizado esto, veremos nuestro proyecto. Tendremos el paquete que fue configurado en Initializr donde se encuentra el “ejecutable” de la aplicación. Al mismo tiempo, tendremos el archivo `application.properties` en donde se pueden realizar diferentes configuraciones específicas del proyecto y el archivo `POM.xml` en donde se especifican las dependencias.

Por otro lado, como Maven ya se encuentra integrado en Apache Netbeans 12.3, comenzará a descargar las dependencias necesarias de forma automática a partir de las configuraciones que recibió de initializr:



PASO 8

Como último paso, para verificar la correcta configuración del proyecto creado, vamos a ejecutarlo. Spring Boot ya tiene incorporado un servidor Web (en este caso Apache Tomcat) que será el encargado de levantar la aplicación que desarrollaremos directamente desde el Main de la misma. Esto se lleva a cabo mediante la anotación `@SpringBootApplication`, ya que de esta manera, el IDE identifica que se trata de una aplicación de Spring:

```
package com.miaplicacion.primerproyecto;

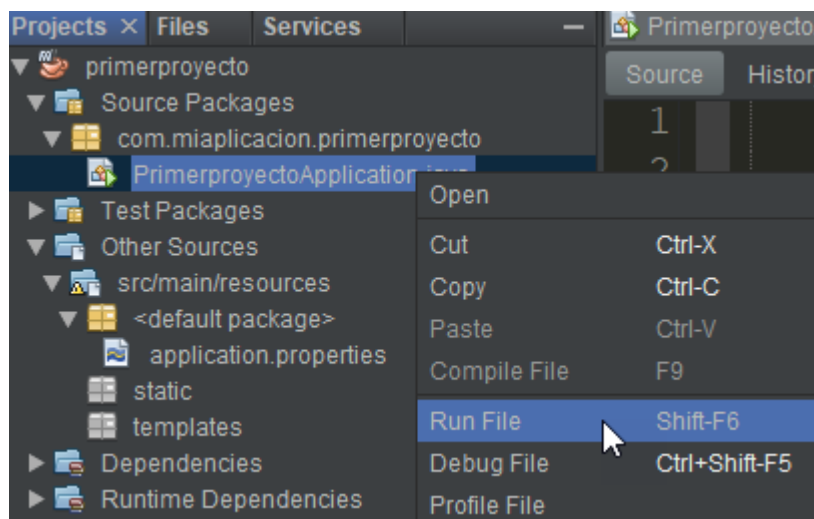
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class PrimerproyectoApplication {

    public static void main(String[] args) {
        SpringApplication.run(PrimerproyectoApplication.class, args);
    }

}
```

Una vez posicionados en la clase Main, vamos a ejecutarla y a verificar en la consola de salida sobre qué URL y puerto la levanta el servidor web:



A partir de los datos que se observan en la consola, vamos a dirigirnos a la url (en este caso <https://localhost:8080>) y verificar que la aplicación esté corriendo correctamente. Si se despliega el mensaje que se observa a continuación, significa que la aplicación se ejecutó correctamente:



Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Fri Aug 13 20:21:25 ART 2021

There was an unexpected error (type=Not Found, status=404).

No message available

Patrón MVC en Spring Boot

¿Recuerdas este patrón?

Este patrón MVC (Modelo Vista Controlador) lo hemos visto cuando trabajamos con el Front End con Angular, como verás, Spring Boot también lo utiliza, recordemos que permite una separación entre la lógica de negocio de una aplicación y la vista que se le presenta al usuario, utilizando como intermediario a un controlador que se encarga de tomar la decisión de cómo interactúan la vista y el modelo entre sí.

En este patrón, el usuario realiza una petición, un controlador la recibe y decide hacia dónde debe ir la misma o qué acciones deben realizarse para emitir una respuesta. Veamos como Spring Boot utiliza el Patrón MVC:

Cada una de las partes del patrón cumple con su funcionalidad específica:

- **El Controlador:** Se encarga de “controlar” o hacer de intermediario; recibe las órdenes del usuario, solicita los datos al modelo y se los comunica a la vista. Trabaja como si se tratara de un “pivote” o un apoyo que se encarga de distribuir las tareas. En SpringBoot se determina la clase controladora mediante la Annotation `@RestController`, es decir, si creas una clase y le pones `@RestController` tomara el comportamiento como un controlador. Vemos el siguiente código como se ve:

```
@RestController public class PersonaController { private static final String template = "Hello, %s!"; }
```

- **El Modelo:** Se encarga del modelado de los datos. En él se encuentra generalmente la lógica de usuario y las fuentes de datos, como por ejemplo, el consumo de datos desde una base de datos en particular. En las líneas de código que te mostramos podrás identificar el acceso a datos que se realiza por medio de `productoService.obtenerTodosProductos()` que está dentro de la clase que representa el modelo:

```
@GetMapping(value = "/lista")
public String listar(Model model) {
    model.addAttribute("titulo", nombreAplicacion);
    model.addAttribute("productos", productoService.obtenerTodosProductos());
    return "lista"; }

@GetMapping(value = "/listaModelMap")
public String listarModelMap(ModelMap model) {
    model.addAttribute("titulo", nombreAplicacion);
    model.addAttribute("productos", productoService.obtenerTodosProductos());
    return "lista"; }

@GetMapping(value = "/listaModelAndView")
public ModelAndView listarModelAndView() {
    ModelAndView mav = new ModelAndView();
    mav.addObject("titulo", nombreAplicacion);
    mav.addObject("productos", productoService.obtenerTodosProductos());
    mav.setViewName("lista"); return mav;
}
```

- **La Vista:** Es la interfaz gráfica que se le presenta al usuario. Generalmente recibe datos provenientes del modelo a través del controlador y se los muestra al usuario en cuestión. Como ya sabemos su estructura es HTML y para imprimir las variables usamos `${objeto.propiedad}` y en caso de ser

una variable `${nombreDeVariable}`. Veamos en la siguiente imagen cómo se ve:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head th:replace="layout/layout::head">
  <meta charset="UTF-8">
</head>
<body>
  <header th:replace="layout/layout::header"></header>
  <div class="container">
    <div class="card bg-dark">
      <div class="card-header">
        <h3>Lista de productos</h3>
      </div>
      <div class="card-body">
        <table class="table table-striped">
          <thead>
            <tr>
              <th>Id</th>
              <th>Código</th>
              <th>Nombre</th>
            </tr>
          </thead>
          <tbody>
            <tr th:each="producto: ${productos}">
              <td th:text="${producto.id}"></td>
              <td th:text="${producto.codigo}"></td>
              <td th:text="${producto.nombre}"></td>
              <td><a th:href="@{eliminar/} + ${producto.id}" th:text="Eliminar" class="btn btn-outline-danger"></a></td>
            </tr>
          </tbody>
          <tfoot>
            <tr>
              <td colspan="5">
                <a th:href="@{crear}" class="btn btn-outline-primary btn-block">Añadir Producto</a>
              </td>
            </tr>
          </tfoot>
        </table>
      </div>
    </div>
  </div>
  <footer th:replace="layout/layout::footer" class="bg-dark"></footer>
</body>
</html>
```

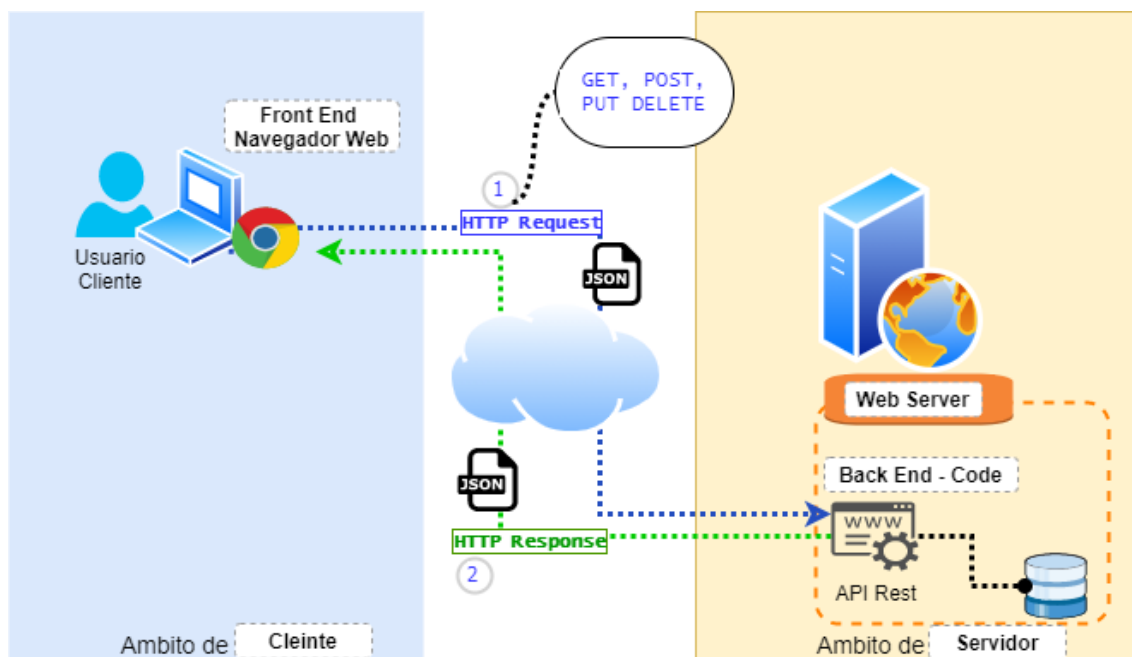
API REST con Spring Boot

¿Qué es una API?

Una **API (application programming interface)** es un conjunto de funciones y procedimientos (métodos) que se usa para diseñar e integrar el software de diferentes aplicaciones. Esto permite que, dos aplicaciones se puedan comunicar entre sí, por más que estén desarrolladas en lenguajes de programación completamente distintos.

Una API es un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones. Suele considerarse como el contrato entre el proveedor de información y el usuario, en donde se establece el contenido que se necesita del consumidor (la llamada) y el que requiere el productor (la respuesta). Por ejemplo, el diseño de una API para un servicio meteorológico podría solicitar que el usuario escribiera un código postal y que el productor diera una respuesta en dos partes: la primera sería la temperatura máxima y la segunda, la mínima.

La forma más común de implementación de una API es mediante REST (Representational State Transfer), el cual es un tipo de servicio que se caracteriza por no tener estado alguno y por lograr interconexiones mediante el protocolo HTTP con mensajes de tipo XML o JSON. Ejemplo de la interacción de una Api:



Cuando hablamos de API REST también estamos hablando de protocolo HTTP, Request, Response, verbos HTTP y mensajes sin estado. Recordemos lo que ya hemos aprendido, pero aplicándolo a las APIs.

¿Qué es eso de sin estado o stateless y por qué es importante para la API REST?

Un mensaje, una aplicación o un proceso **sin estado (stateless)** es algo que no guarda datos ni hace referencia sobre información de operaciones anteriores, es como si no tuviera memoria. Cada operación sin importar cuántas se hayan realizado anteriormente exitosas o erróneas, comienza desde cero, como si nunca pasó nada.

En pocas palabras: En las aplicaciones REST, cada solicitud debe contener toda la información necesaria para que el servidor la entienda y sepa qué hacer, en lugar de depender de que el servidor recuerde solicitudes anteriores. Cuando enviamos una petición al servidor para que realice una acción se le debe enviar toda la información que sea necesaria para que pueda procesar, por ejemplo, datos de seguridad, acciones que necesita realizar, datos necesarios para eliminar o recuperar información.

Para poder recordar cómo debemos usar los verbos HTTP con nuestra API necesitamos asociarlos a las acciones que realiza, en la siguiente imagen se agregó su traducción en palabras y se agrupó para que puedas recordar, veamos:

Metodos seguros sin acción en el server	{	GET	RECUPERA cualquier dato del servidor por Request
		HEAD	INSPECCIONA el recurso HEADERS
Mensajes con Body Envío de datos al Servidor	{	PUT	DEPOSITA datos en el Server - lo contrario de GET
		POST	ENVIA datos de entrada para procesar
		PATCH	MODIFICA PARCIALMENTE el recurso
		TRACE	ECO o RETORNO de nuevo mensaje recibido
		OPTIONS	Recupera las CAPACIDADES del Servidor
		DELETE	Elimina un recurso - NO GARANTIZADO

Ahora hagamos otra asociación, pero esta vez a otras tecnologías que ya conocemos y veamos qué acciones podríamos realizar, por ejemplo, en una base de datos o en objetos de nuestra aplicación, veamos la siguiente imagen:

Verbo HTTP	Operación	en SQL	en Objetos
PUT / POST	Crear	INSERT	Crear
GET	Leer	SELECT	Recuperar
PUT / PATCH	Actualizar	UPDATE	Modificar
DELETE	Eliminar	DELETE	Destruir

En el módulo anterior conociste la [URL](#) (Uniform Resource Locator) este término es correcto para identificar paginas web, pero para trabajar con APIs necesitamos conocer la [URI](#) (Uniform Resource **Identifier**) que es la dirección completa que permite **acceder e identifica el recurso** de una forma única y pasar parámetros, si quieres profundizar en este tema puedes hacerlo [aquí](#).

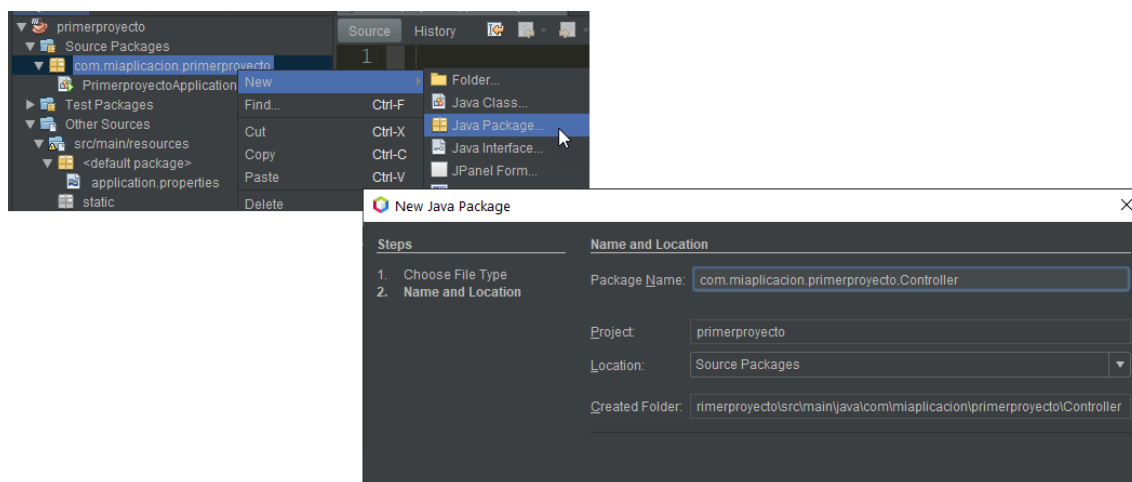
Creamos una API

¿Que necesitamos?

Vamos a necesitar lo siguiente:

1. Tener Instalado el IDE.
2. El navegador web.
3. Curiosidad y ganas de explorar.

Crear una Api con SpringBoot es bastante sencillo. En primera instancia, crearemos un proyecto en Initializr (como se especificó antes). Una vez creado el proyecto y que se encuentre levantado en el IDE, vamos a crear un nuevo paquete llamado Controller:



Creando un @Controller

El nuevo paquete creado representará la capa controladora del proyecto (siguiendo el MVC). Dentro del paquete, se creará una nueva clase que se encargará de recibir las distintas peticiones o solicitudes, es decir, crearemos un controlador. Para ello será necesario mapear la clase con la annotation `@RestController`. El código y detalles de esta clase se pueden ver en la siguiente imagen:



Creando un @GetMapping

Como se mencionó anteriormente, las APIs reciben solicitudes mediante el protocolo HTTP, es decir, pueden recibir solicitudes mediante los diferentes métodos que HTTP provee, como ser por ejemplo GET, POST, PUT, etc. Cuando se desea llevar a cabo un end-point que se ejecuta al recibir una solicitud GET, ésta debe ser etiquetada dentro del controller con la annotation `@GetMapping`. Te mostramos un ejemplo de un HelloWorld:

Annotation

```
package com.miaplicacion.primerproyecto.Controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

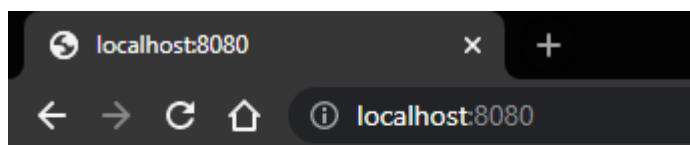
@RestController
public class HolaController {

    @GetMapping
    public String decirHola() {

        return "Hello World";
    }

}
```

De esta manera, cuando ingresemos desde el navegador (que por defecto utiliza el método GET) a la url de la aplicación, obtendremos el resultado de la función decirHola() que se va a ejecutar, tal como puede verse a continuación:



Hello World

De forma genérica, si no se especifica una URL en particular en @GetMapping, el mensaje se devolverá en el directorio raíz de la aplicación; sin embargo, es posible especificar una URL en particular. Supongamos que queremos la devolución del mensaje en el path “/hola”, para ello haremos un pequeño agregado a @GetMapping:


```
package com.miaplicacion.primerproyecto.Controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HolaController {

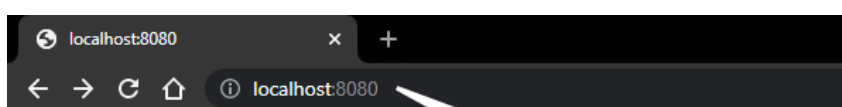
    @GetMapping ("/hola")
    public String decirHola() {

        return "Hello World";
    }

}
```

Se agrega Path

Una vez hecho esto, al ejecutar la aplicación, en el directorio raíz obtendremos un mensaje de error de que el recurso no existe, sin embargo, si colocamos "/hola" obtendremos el mensaje de respuesta:



Whitelabel Error Page

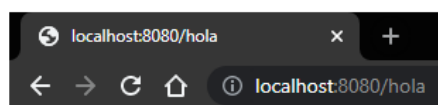
Sin Path

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Fri Aug 13 21:09:19 ART 2021

There was an unexpected error (type=Not Found, status=404).

No message available



Hello World

Con Path

Parámetros con @GetMapping

Así como en un @GetMapping puede establecerse un determinado path, también es posible recibir, tal como el método HTTP GET lo indica, valores o parámetros mediante la URL. Éstos, pueden asignarse a las diferentes funciones que tengamos en el controller a partir de la annotation @PathVariable. Un ejemplo de esto puede verse en la siguiente ilustración:

```
package com.miaplicacion.primerproyecto.Controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HolaController {

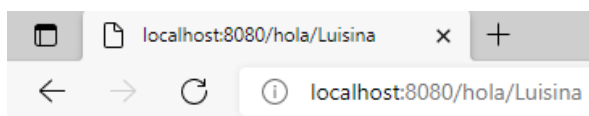
    @GetMapping ("/hola/{nombre}")
    public String decirHola(@PathVariable String nombre) {

        return "Hello World " + nombre;
    }
}
```

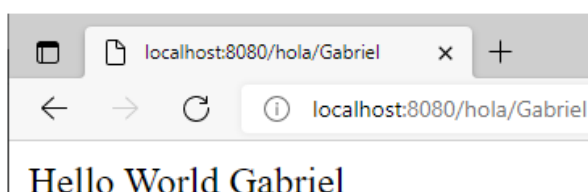
El nombre del parámetro de la url

Tiene que coincidir con el de @PathVariable

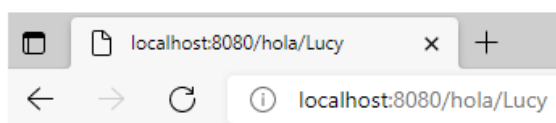
Con este ejemplo, recibiremos mediante el método GET y por la URL un nombre como parámetro. Con @PathVariable asignamos este valor a una variable (que debe tener el mismo nombre) para luego utilizarla en la función en cuestión, en este caso, para agregar el nombre al saludo "Hello World". Ejemplo de la devolución de una solicitud con un nombre:



Hello World Luisina



Hello World Gabriel



Hello World Lucy

El mensaje cambia
según el parámetro
que se pase en la
URL

Así como lo hicimos con el parámetro nombre, podemos agregar nuevos parámetros e ir recibéndolos por la URL de la misma manera. Te mostramos un ejemplo completo con los parámetros edad y profesión:

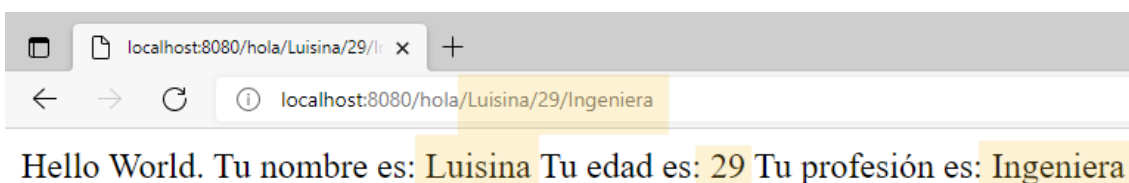
```
package com.miaplicacion.primerproyecto.Controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HolaController {

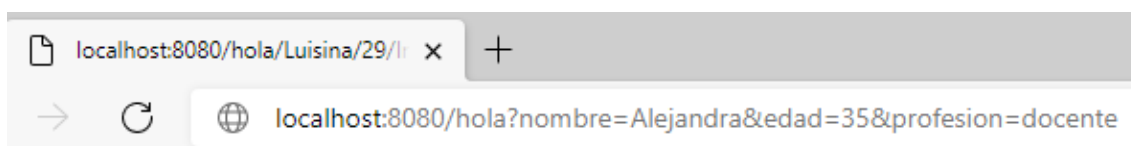
    @GetMapping ("/hola/{nombre}/{edad}/{profesion}")
    public String decirHola(@PathVariable String nombre,
                           @PathVariable int edad,
                           @PathVariable String profesion) {

        return "Hello World. Tu nombre es: " + nombre +
               " Tu edad es: " + edad + " Tu profesión es: " + profesion;
    }
}
```



Creando @RequestParam

Otra forma de recibir múltiples parámetros en un endpoint dentro de un controller, es mediante el uso de la annotation `@RequestParam`. Ésta es una anotación que permite recibir parámetros mediante el método GET. La especificación de los parámetros se manifiesta mediante un signo “?” luego del path en cuestión y luego, cada uno de los parámetros se indica mediante su nombre, un símbolo igual y su valor. En caso de que haya más de un parámetro, se unen entre sí mediante el símbolo “&”. Ejemplo de cómo se vería la URL:



Dentro del endpoint creado, se debe especificar la annotation `@RequestParam` por cada uno de los parámetros que recibiremos con la variable asociada que recibirá dicho valor. Un ejemplo de esto a nivel código puede verse en la siguiente ilustración, mientras que en la segunda ilustración puede verse la respuesta a la solicitud:

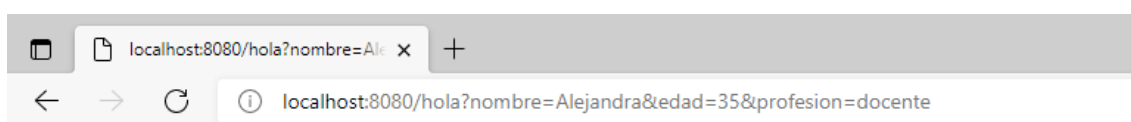
```
package com.miaplicacion.primerproyecto.Controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HolaController {

    @GetMapping ("/hola")
    public String decirHola(@RequestParam String nombre,
                           @RequestParam int edad,
                           @RequestParam String profesion) {

        return "Hello World. Tu nombre es: " + nombre +
               " Tu edad es: " + edad + " Tu profesión es: " + profesion;
    }
}
```



Hello World. Tu nombre es: Alejandro Tu edad es: 35 Tu profesión es: docente

@PathVariable y @RequestParam cumplen funciones muy similares, sin embargo, sus implementaciones son diferentes. En @PathVariable, los parámetros se brindan mediante diferentes apartados path ("/") de la dirección, mientras que en @RequestParam, los datos se especifican dentro de un mismo path mediante el símbolo "?" y separando a cada uno de ellos mediante el símbolo "&".

Más sobre @annotation

Creando @PostMapping

Así como para recibir solicitudes mediante el método GET se utiliza en el controller la annotation "@GetMapping", también existe una que permite la recepción y tratamiento de solicitudes o requests que se envíen mediante el método POST, esta annotation es el @PostMapping.

La annotation se coloca para mapear el método que se ejecutará al recibir la petición POST en la determinada URL que se especifique. Un ejemplo a nivel código puede verse a continuación:

```
package com.miaplicacion.primerproyecto.Controller;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HolaController {

    @PostMapping ("/cliente")
    public void nuevoCliente() {

        //...lo que hará el método

    }
}
```

Recepción de un objeto mediante @RequestBody

@PostMapping tiene un fiel aliado a la hora de recibir valores en una solicitud o request que es el @RequestBody. Esta annotation permite recibir objetos de dominio completos al endpoint creado mediante el cuerpo del mensaje de la solicitud para convertirlos luego en la aplicación en objetos Java.

Si suponemos una clase Cliente y el método nuevoCliente como se ve en la ilustración a continuación, mediante @RequestBody se podrían recibir los datos correspondientes a un cliente y transformarlos en un objeto Cliente Java:

```
public class Cliente {

    private Long id;
    private String nombre;
    private String apellido;
}
```

```
@RestController
public class HolaController {

    @PostMapping ("/cliente")
    public void nuevoCliente(@RequestBody Cliente cli) {

        System.out.println("Datos cliente: " + cli.getNombre() +
                           " Apellido: " + cli.getApellido());

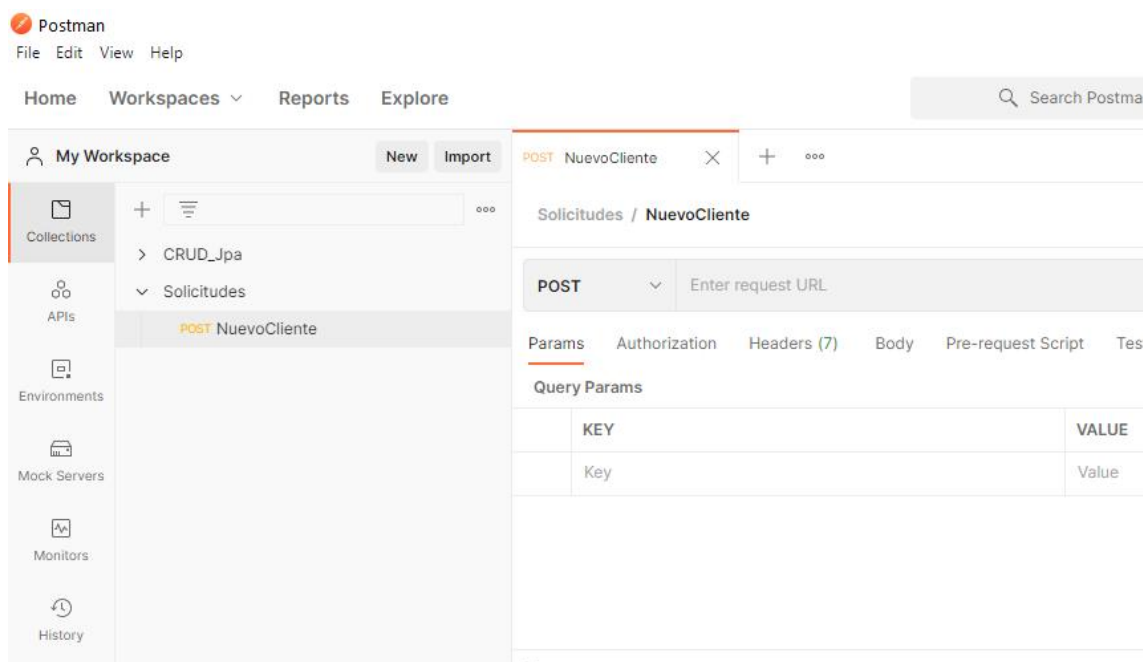
    }
}
```

Como se mencionó en otras ocasiones, el método POST no envía valores mediante la URL en cuestión, sino mediante el cuerpo o la cabecera de los mensajes HTTP.

Como los navegadores utilizan por defecto el método GET, para poder probar los endpoints que utilizan el método POST, es necesario simular las solicitudes que se realizan mediante este verbo con algún software adicional. Uno de los más utilizados por excelencia es Postman.

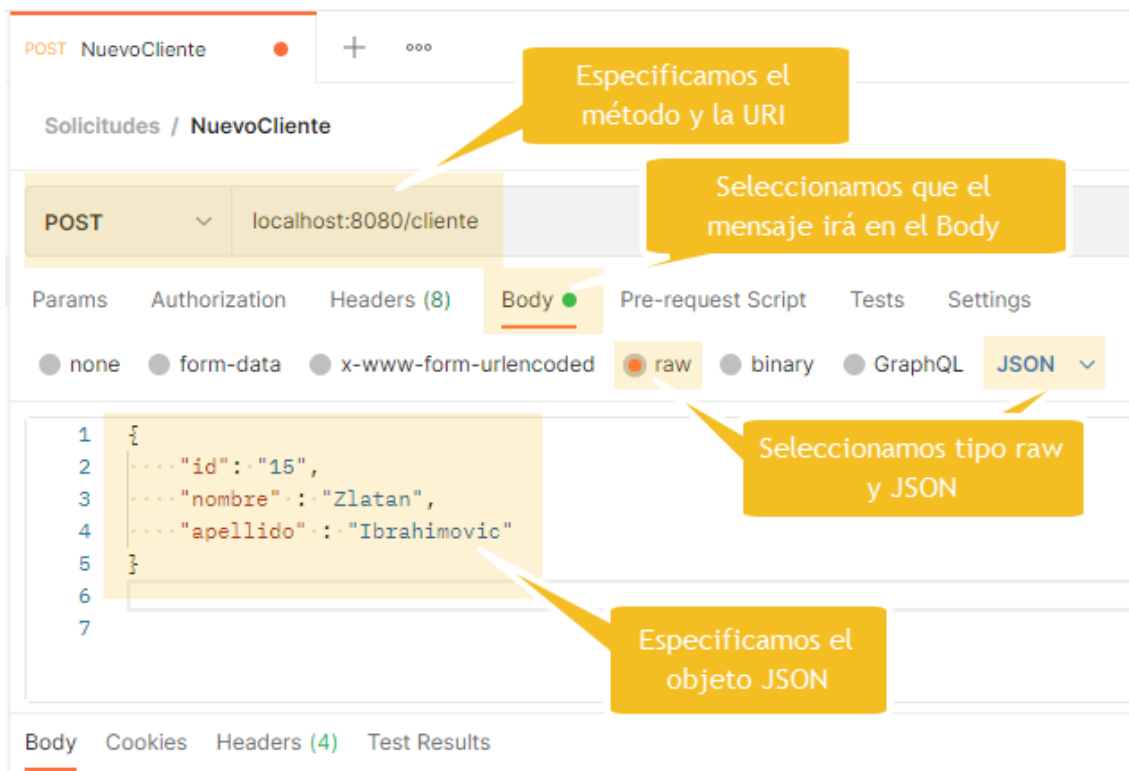
¿Qué es Postman?

Postman es un software que permite simular el envío de solicitudes HTTP REST mediante diferentes métodos sin la necesidad de desarrollar una aplicación cliente o de contar con un front-end en particular. Para utilizarlo únicamente es necesario descargarlo de su página oficial <https://www.postman.com/downloads/>, instalarlo y empezar con sus configuraciones iniciales. Un ejemplo de cómo se ve la aplicación puede verse a continuación en la ilustración. A partir de ahora, utilizaremos Postman para probar y simular las diferentes requests o solicitudes HTTP hacia nuestras APIs:

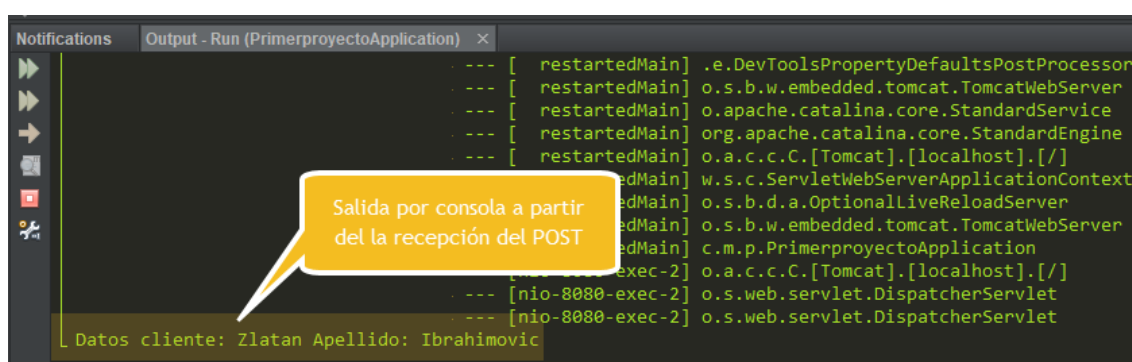


Pruebas con POSTMAN

Partiendo del ejemplo desarrollado en la Ilustración anterior, pasaremos a simular en POSTMAN una solicitud HTTP mediante POST, donde incluiremos en el body del mensaje, un objeto JSON que luego Spring se encargará de convertir a un objeto Java mediante `@RequestBody` :



Una vez que tengamos todo configurado, podemos hacer clic en Send y esto enviará la solicitud POST a nuestra aplicación. En la consola de salida del servidor, que nos proporciona el IDE, podemos visualizar si efectivamente el `System.out.println` que colocamos, recibió y mostró correctamente los valores que recibimos. Un ejemplo de esto puede verse a continuación:



Respuestas con @ResponseBody

@ResponseBody

Así como el método POST permite la utilización de una annotation `@RequestBody` para recibir un objeto JSON en una solicitud, el método GET también posee una

annotation `@ResponseBody` para conformar un objeto para la devolución de un mensaje.

Supongamos que tenemos una lista de clientes, y que mediante una solicitud GET deseamos devolver estos clientes que se encuentran en la lista, para ello crearemos un método `traerClientes()` y lo mapearemos con la annotation `@ResponseBody`, esto puede verse en la siguiente imagen:

```
package com.miaplicacion.primerproyecto.Controller;
import com.miaplicacion.primerproyecto.model.Cliente;
import java.util.ArrayList;
import java.util.List;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HolaController {

    @GetMapping ("/cliente/traer")
    @ResponseBody
    public List<Cliente> traerClientes() {

        List <Cliente> listaClientes = new ArrayList<Cliente>();
        listaClientes.add(new Cliente(1L,"Zlatan", "Ibrahimovic"));
        listaClientes.add(new Cliente(2L,"Cristiano", "Ronaldo"));
        listaClientes.add(new Cliente(3L,"Lionel", "Messi"));

        return listaClientes;
    }
}
```

Creamos una lista de prueba y agregamos registros

Devolvemos la lista de clientes mediante el ResponseBody

Si simulamos la solicitud GET mediante POSTMAN, recibimos como resultado los 3 objetos java creados en el endpoint, en el body de la response recibida, tal como vemos a continuación:

The screenshot shows a REST client interface. At the top, the method is 'GET' and the URL is 'localhost:8080/cliente/traer/'. Below this, there are tabs for 'Params', 'Authorization', 'Headers (6)', 'Body', 'Pre-request Script', 'Tests', and 'Settings'. The 'Body' tab is selected, and the content type is set to 'none'. Below the tabs, there is a section for 'This request de' (partially visible). At the bottom, there are tabs for 'Body', 'Cookies', 'Headers (5)', and 'Test Results'. The 'Body' tab is selected, and the content is displayed in 'JSON' format. The JSON response is as follows:

```

1  [
2    {
3      "id": 1,
4      "nombre": "Zlatan",
5      "apellido": "Ibrahimovic"
6    },
7    {
8      "id": 2,
9      "nombre": "Cristiano",
10     "apellido": "Ronaldo"
11   },
12   {
13     "id": 3,
14     "nombre": "Lionel",
15     "apellido": "Messi"
16   }
17 ]

```

ResponseEntity

Anteriormente vimos el funcionamiento de `@ResponseBody`, el cual permite enviar datos o mensajes JSON dentro del cuerpo de un mensaje HTTP, sin embargo, no es la única forma de devolver mensajes desde el controlador, como otra opción tenemos a `ResponseEntity`.

`Response Entity` administra todo el paquete completo de una respuesta HTTP, es decir, puede manipular el cuerpo, la cabecera o incluso los códigos de estado, haciendo que la respuesta brindada sea totalmente personalizada. Un ejemplo de una respuesta con `ResponseEntity` con un Status Code puede verse en la siguiente ilustración. Por otro lado, como resultado obtenemos lo que podemos observar en la segunda ilustración.

Ejemplo de uso de `ResponseEntity`:

```
package com.miaplicacion.primerproyecto.Controller;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HolaController {

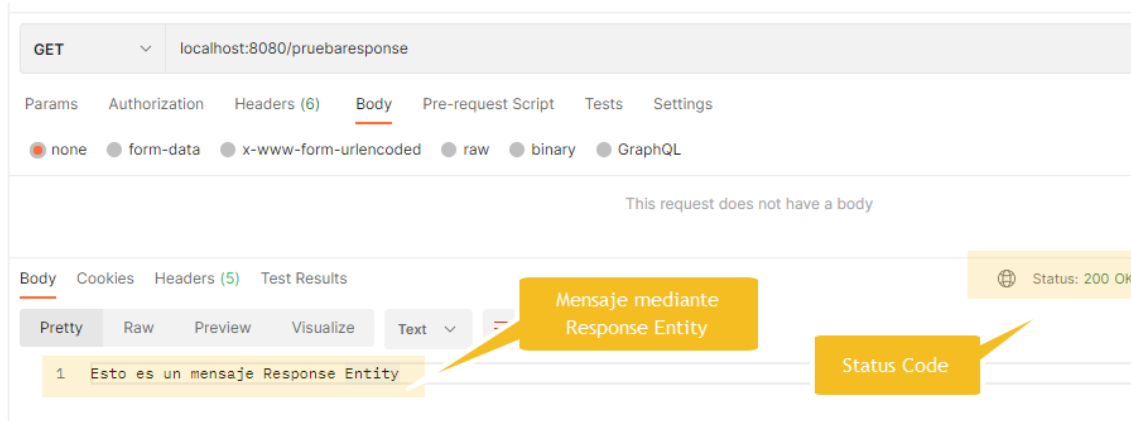
    @GetMapping ("/pruebaresponse")
    ResponseEntity<String> traerRespuesta() {

        return new ResponseEntity<>("Esto es un mensaje Response Entity", HttpStatus.OK);

    }

}
```

Prueba de Response Entity desde Postman



Patrón DTO en Spring

Implementación del patrón DTO

Supongamos un caso de una inmobiliaria que contiene una clase Propiedad (que muestra los datos de una determinada propiedad en alquiler) y una clase Inquilino (que muestra los datos de un potencial inquilino). En el modelo de datos, ambas clases estarán por separado, pero si aplicamos el patrón DTO, podemos crear una clase que incorpore los datos que necesitamos de cada una de ellas para devolverlo en una response. Veamos un ejemplo paso a paso:

PASO 1

Creamos un proyecto SpringBoot con inicializr. Luego dentro de él, creamos las clases Propiedad e Inquilino en un paquete llamado model:

```
package com.inmobiliaria.alquileres.model;

public class Propiedad {

    private Long id_propiedad;
    private String tipo_propiedad;
    private String direccion;
    private Double metros_cuadrados;
    private Double valor_alquiler;

}

package com.inmobiliaria.alquileres.model;

public class Inquilino {

    private Long id_inquilino;
    private String dni;
    private String nombre;
    private String apellido;
    private String profesion;

}
```

PASO 2

Creamos un nuevo paquete llamado DTO y dentro crearemos una clase llamada PropiedadDTO, donde incorporaremos datos de las propiedades y de los inquilinos:

```
package com.inmobiliaria.alquileres.dto;
import java.io.Serializable;

public class PropiedadDTO implements Serializable{

    private Long id_propiedad;
    private String tipo;
    private String direccion;
    private Double valor_alquiler;
    private String nombre_inquilino;
    private String apellido_inquilino;

}
```

PASO 3

Creamos nuestro paquete controller y dentro armamos nuestra clase controladora. Luego creamos un nuevo endpoint, en donde tendremos un objeto propiedad y otro inquilino que unificaremos en un objeto propiedaddto para devolverlo mediante @ResponseBody. Un ejemplo de cómo implementar esto:

```
@RestController
public class AlquileresController {

    @GetMapping ("/propiedad/{id}")
    @ResponseBody
    public PropiedadDTO devolverPropiedad(@PathVariable Long id) {

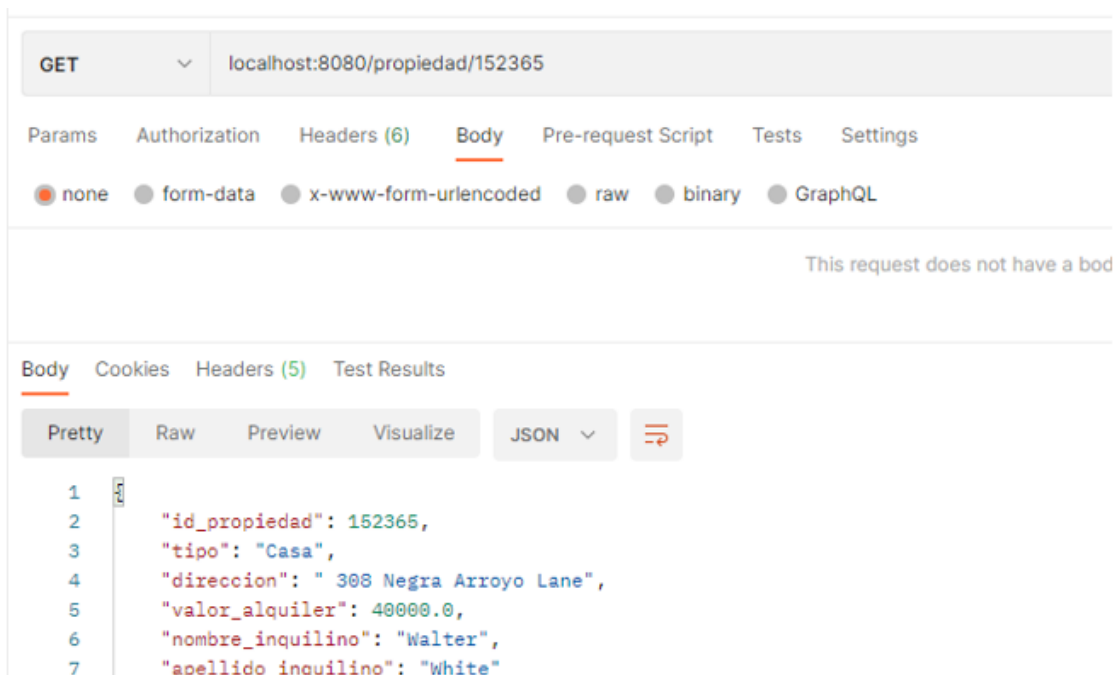
        //supongamos que obtenemos una propiedad por su id y su inquilino
        //desde una base de datos
        Inquilino inqui = new Inquilino (1L, "12345678", "Walter", "White", "Profesor");
        Propiedad prop = new Propiedad (152365L, "Casa", " 308 Negra Arroyo Lane", 200.0, 40000.0);

        PropiedadDTO propiDTO = new PropiedadDTO();
        //Ahora unificamos los datos del inquilino y de la
        //propiedad en un solo objeto
        propiDTO.setId_propiedad(prop.getId_propiedad());
        propiDTO.setTipo(prop.getTipo_propiedad());
        propiDTO.setDireccion(prop.getDireccion());
        propiDTO.setValor_alquiler(prop.getValor_alquiler());
        propiDTO.setNombre_inquilino(inqui.getNombre());
        propiDTO.setApellido_inquilino(inqui.getApellido());

        return propiDTO;
    }
}
```

PASO 4

Probamos con Postman realizar una solicitud y ver si obtenemos como respuesta el objeto DTO que acabamos de crear:



Arquitectura Multicapa

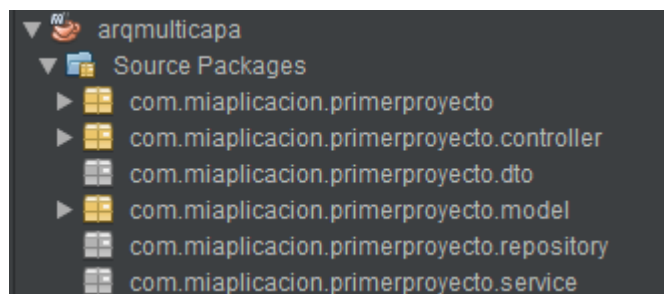
Toda aplicación que sigue buenas prácticas en cuanto al desarrollo de software cumple con algún tipo de modelo o arquitectura de capas, es decir, una separación entre cada una de las partes con la que interactúa la misma y una forma de comunicación entre ellas. Existen diversos modelos o arquitecturas multicapa que pueden ser implementados o utilizados según el proyecto sobre el cual se esté trabajando, sin embargo, hay algunas estandarizaciones a seguir que se adaptan a la mayoría de los desarrollos realizados en Java. A continuación, se especifica en mayor detalle cada una de las capas de una de las arquitecturas multicapas estándar utilizada para desarrollo de aplicaciones con Spring Boot:

- **Controller:** Es la capa encargada de atender las solicitudes http entrantes, derivarlas a la capa que corresponda, esperar por una respuesta, generarla y transmitirla nuevamente al cliente. Generalmente la capa “Controller” trabaja estrechamente con la capa de “Service”, donde a partir de una request llama a las funciones que necesite de la capa service para generar una response. Cabe destacar que, dependiendo del proyecto, no se considera una “buena práctica” el pasaje de clases DTO entre Controller y Service, es necesario recordar que los DTO están pensados para la generación de las respuestas del Controller y no para la comunicación entre capas.
- **Repository o DAO (Data Access Object):** Es la capa encargada de la persistencia de los datos, es decir, del resultado de la interacción de modelado entre las clases desarrolladas y las tablas de una base de datos. Permite el acceso a los datos mediante diferentes tecnologías como por ejemplo JDBC o algún ORM como por ejemplo JPA con Hibernate. Cada una de las clases que se encuentren dentro de esta capa deben estar mapeadas/etiquetadas mediante la annotation @Repository.
- **Model (o Entity):** La capa “model” trabaja estrechamente en conjunto con la clase Repository. Cada una de las clases modela un objeto de la vida real y es marcado con la annotation @Entity en caso de que se transforme en una entidad (tabla) en la base de datos. Cada instancia que se haga a una clase entity, en caso que sea persistida, representará una fila en una tabla de la base de datos.
- **DTO (Data Transfer Object):** Esta capa se encarga de contener todas las clases DTO que hayan sido especificadas en un proyecto. Los DTO buscan desacoplar la forma de presentación de los datos (a futuro en el frontend) con respecto a los objetos propiamente dichos de la capa Model.
- **Service:** La capa de “Service”, mejor conocida como lógica de negocio, es la capa donde se especifican todas las funciones u operaciones que sean necesarias y que puedan ofrecer, como dice su nombre, un servicio a la capa controller. La capa service, por ejemplo, puede encargarse de las autenticaciones o de las políticas de autorización que puede tener la aplicación con respecto al acceso a determinadas funciones.

Arquitectura Multicapas en un proyecto

Implementar la arquitectura multicapas en un proyecto desarrollado con Spring Boot es bastante sencillo. Básicamente, cada capa se representa mediante un

paquete dentro de la aplicación desarrollada en donde irán todas las clases que correspondan a esa capa en cuestión. Un ejemplo de modelo de capas implementado en Netbeans:



Si bien cada una de las capas se representa con paquete distinto, todas se encuentran dentro del paquete principal de la aplicación que estemos desarrollando.

Por otro lado, es importante que las capas puedan comunicarse entre si, para ello si bien es posible hacer esto mediante la creación de instancias de objetos, es una muy buena práctica realizarla mediante interfaces. Esta forma de implementación permite que se puedan cambiar las distintas implementaciones sin la necesidad de modificar el RestController (donde residen nuestros endpoints) o la capa que esté utilizando la dependencia a la que hagamos referencia. Este concepto se conoce como inversión de control, y lo desarrollaremos en mayor detalle a continuación.

Inyección de Dependencias e Inversión de Control

Inversión de control (Inversion of control)

Un aspecto clave y muy característico de Spring como framework es la utilización de la inversión de control, o mejor conocida como IoC, por sus siglas en inglés: Inversion Of Control.

La inversión de control es un concepto en el cual, en lugar de que el programador a través de la aplicación lleve el control del flujo de la misma, sea el framework quien lo haga. Tal como lo dice su nombre, los roles de control se invierten.



En la ilustración anterior podemos ver un flujo normal de trabajo, donde la app, a través de las líneas de código que especifique el programador, se encarga de llamar a las diferentes funcionalidades de un framework. Sin embargo, en la

inversión de control, este proceso es diferente, en donde el framework se encarga de llamar a la aplicación. Un ejemplo de esto podemos verlo a continuación:



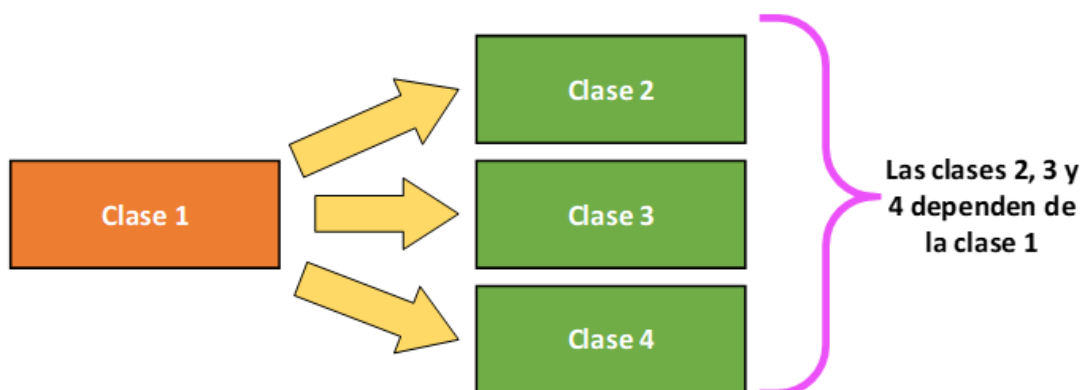
Para poder aplicar este concepto, el framework tiene que entender qué es lo que hace la aplicación o de qué manera funciona. Para ello, el framework le requiere a la aplicación una serie de datos que puede estar expresada de diferentes maneras dependiendo del lenguaje de programación, en Java, por ejemplo, lo expresamos mediante las Annotations.

Uno de los principales “problemas” que trata la IoC es la creación o instanciación innecesaria de objetos mediante la sentencia `new`; mediante la IoC, el framework se asegura de crear los objetos (generalmente respetando el patrón de diseño Singleton que establece la utilización de una sola instancia por clase) y los pone a disposición de la aplicación mediante otro concepto muy importante conocido como Inyección de Dependencias

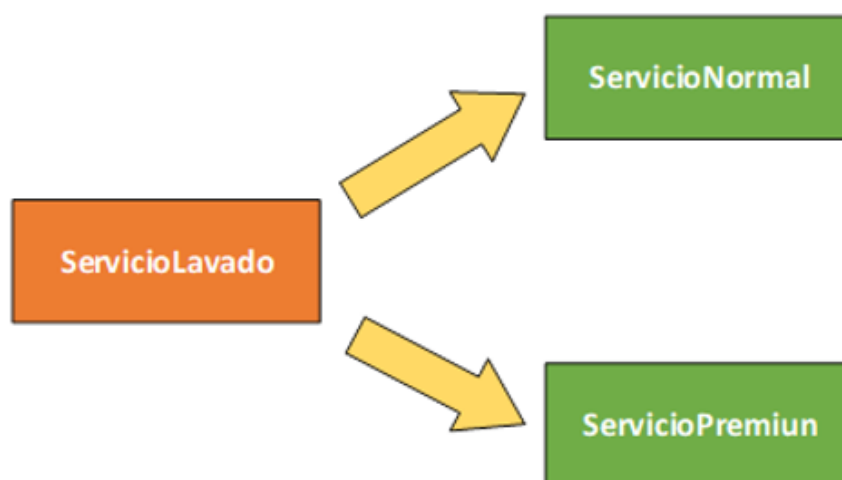
Inyección de Dependencias

La inyección de dependencias, o mejor conocida como DI (dependency injection por sus siglas en inglés), es un patrón de diseño que está orientado al manejo de los objetos de una aplicación. Su principal objetivo es el de mantener las capas de una aplicación lo más desacopladas posible entre sí. Para poder lograr esto, la inyección de dependencias permite que cada una de las partes del programa que se esté desarrollando sea independiente y que no se comuniquen entre sí mediante instancias, sino mediante interfaces.

Se entiende entonces que la inyección de dependencias busca desacoplar lo máximo posible la relación entre clases o capas, pero... ¿Qué es una dependencia? Una dependencia es una relación que puede existir entre una o varias clases, donde generalmente una (o varias) dependen de otra principal:



¿Cómo se representa esto a nivel código? Supongamos que tenemos el modelado de un lavadero de autos, donde existe una clase llamada ServicioLavado de la cual dependen otras dos clases, ServicioNormal y ServicioPremiun, tal como puede verse gráficamente en las ilustraciones a nivel código:



```

public class ServicioLavado {

    ServicioNormal serviNorm; // genero dependencia
    ServicioPremiun serviPrem; //genero dependencia

    public ServicioLavado() {
        serviNorm = new ServicioNormal();
        serviPrem = new ServicioPremiun();
    }
}
    
```


Como se puede ver en ambas imágenes, tanto `ServicioNormal` como `ServicioPremium` dependen de `ServicioLavado` y es él quien tiene la responsabilidad de inicializar a ambos servicios en su constructor, sin embargo, si aplicáramos inyección de dependencias, podríamos delegar esta responsabilidad que tiene `ServicioLavado` a otra clase, como por ejemplo, la clase `main` que tengamos en el proyecto. Ahora bien, ¿Cómo se aplica la inyección de dependencias? Existen diferentes maneras de hacerlo, sin embargo, hay 3 que son las más comunes:

1. Mediante un constructor
2. Mediante un setter
3. Mediante la Annotation `@Autowired`

Inyección mediante un constructor

En la inyección de dependencias mediante un constructor, es el propio constructor de una clase el encargado de inyectar la dependencia. En el ejemplo de la ilustración anterior se podía observar cómo el constructor creaba las instancias de ambos tipos de servicios dentro de él; en la inyección de dependencias mediante un constructor, este únicamente recibirá como parámetros los objetos ya creados y los asignará según corresponda. Un ejemplo de esto puede verse a continuación:

```
public class ServicioLavado {  
  
    ServicioNormal serviNorm; // genero dependencia  
    ServicioPremium serviPrem; //genero dependencia  
  
    public ServicioLavado(ServicioNormal serviNorm, ServicioPremium serviPrem) {  
        this.serviNorm = serviNorm;  
        this.serviPrem = serviPrem;  
    }  
}
```

Inyección mediante un Setter

Los métodos `getter` y `setter` nos permiten obtener o setear valores a los atributos de los objetos que sean creados, de igual manera los métodos `set` nos permiten inyectar dependencias, donde a partir de la recepción de un objeto como un parámetro este se asigna:

```
public class ServicioLavado {  
  
    ServicioNormal serviNorm; // genero dependencia  
    ServicioPremium serviPrem; //genero dependencia  
  
    public void setServiNorm(ServicioNormal serviNorm) {  
        this.serviNorm = serviNorm;  
    }  
  
    public void setServiPrem(ServicioPremium serviPrem) {  
        this.serviPrem = serviPrem;  
    }  
  
}
```

@Autowired

Así como Java permite realizar inyección de dependencias de forma genérica mediante setters o constructores, Spring como framework ofrece una annotation para hacerlo. Ésta es conocida como @Autowired.

Cuando se trabaja con una arquitectura multicapas donde es posible que exista una dependencia entre los objetos de las mismas, @Autowired es de gran ayuda. A continuación vamos a ver un ejemplo de su uso en un modelo de capas implementado con Spring Boot:

Supongamos que tenemos un sistema creado con Spring Boot para un blog, en donde tenemos una clase entidad llamada Posteo y una clase PosteoRepository con un solo método que nos devuelve la lista de posteos existentes. Por otro lado tenemos un controller que recibe las peticiones mediante el path /posteos :

```
package com.inyeccion.ejemploAutowired.model;  
  
import lombok.Getter;  
import lombok.Setter;  
  
//lombok nos permite resumir los getters y setters  
//mediante annotations  
@Getter @Setter  
public class Posteo {  
  
    private Long id;  
    private String titulo;  
    private String autor;  
  
    public Posteo(Long id, String titulo, String autor) {  
        this.id = id;  
        this.titulo = titulo;  
        this.autor = autor;  
    }  
  
}
```

Creamos un constructor para crear desde repository un par de objetos para simular una lista traída desde por ejemplo una base de datos.

```
@Repository
public class PosteoRepository {

    public List<Posteo> traerTodos () {

        List<Posteo> listaPosteos = new ArrayList<Posteo>();
        listaPosteos.add(new Posteo (1L, "¿Cómo formatear una PC?", "Luisina de Paula"));
        listaPosteos.add(new Posteo (2L, "¿Cómo mantener la seguridad?", "Gabriel Guismín"));

        return listaPosteos;
    }
}
```

Creamos dos objetos para simular una lista que viniese, por ejemplo, de una base de datos

```
@RestController
public class PosteoController {

    //El autowired inyecta la dependencia
    //sin necesidad de crear un nuevo objeto
    @Autowired
    PosteoRepository repository;

    @GetMapping ("/posteos")
    public List<Posteo> traerTodos () {

        return repository.traerTodos();
    }
}
```

De esta manera, una vez que levantemos nuestra aplicación y se haga la solicitud en cuestión, recibiremos como respuesta los posteos que teníamos en el repository, tal como puede verse en la siguiente ilustración. De esta forma, el controlador inyectó una dependencia al repository mediante @Autowired:

GET localhost:8080/posteos

Params Authorization Headers (6) Body Pre-request Script T

Query Params

KEY	VALUE

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```

1  [
2    {
3      "id": 1,
4      "titulo": "¿Cómo formatear una PC?",
5      "autor": "Luisina de Paula"
6    },
7    {
8      "id": 2,
9      "titulo": "¿Cómo mantener la seguridad?",
10     "autor": "Gabriel Guismin"
11   }
12 ]

```

Bases de Datos con SpringBoot, JPA e Hibernate

¿Por qué esta tecnología?

Uno de los aspectos más importantes a la hora de desarrollar aplicaciones web es sin dudas encontrar la forma de almacenar y manejar los datos que se manejen en ellas. Para ello, existen diferentes tecnologías, algunas son de bajo nivel (que nos hace programar más en detalle) y las de más alto nivel (que nos permite abstraernos de los detalles).

JDBC es un estándar de bajo nivel (y más antiguo) para la interacción con bases de datos. JPA es un estándar de nivel superior para el mismo propósito.

JPA es un ORM (Object Relational Mapping – Mapeo Objeto Relacional) que nos permite utilizar un modelo de objetos en nuestras aplicaciones que puede hacernos la vida mucho más fácil cuando tenemos que trabajar con bases de datos, permitiéndonos relacionar las tablas con las clases de la aplicación.

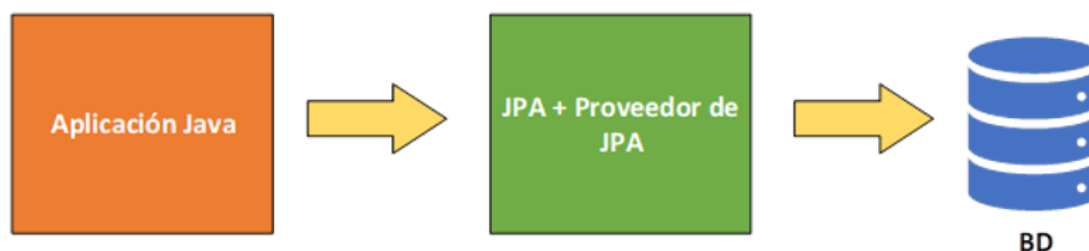
A continuación, veremos algunas de las herramientas ORM más utilizadas en Java en conjunto con Spring Boot:

¿Qué es JPA?

JPA, o mejor conocida por sus siglas “Java Persistence API”, es una colección de clases y métodos que tienen como objetivo lograr la persistencia de datos entre una aplicación desarrollada en Java y una base de datos. Cabe destacar que JPA no es un framework, sino una API que brinda sus servicios y herramientas para poder ser implementados.

JPA busca, como todo ORM, traducir el modelado de las clases Java a un modelado relacional en una base de datos, posibilitando al programador elegir qué clases u objetos persistir, de qué manera y con qué nombre o tipo de dato serán representados.

JPA se vale de una serie de mapeos que se deben realizar sobre cada uno de los elementos de una clase, los mismos, se representan mediante annotations. De esta manera, JPA hace de una especie de traductor entre las clases Java de la aplicación con la que se esté trabajando y la base de datos en sí. Un ejemplo de esto puede verse en el siguiente diagrama:



Podemos observar en la imagen que JPA no es una tecnología que se ofrezca por sí misma, sino que cuenta de un “Proveedor de JPA”. Estos proveedores de JPA brindan las herramientas y annotations necesarias para poder realizar correctamente las operaciones entre las aplicaciones y las bases de datos. Existen diferentes proveedores, entre los cuales se encuentran: EclipseLink e Hibernate, siendo los más conocidos, sin embargo, por su alta flexibilidad y eficacia, Hibernate es uno de los favoritos en la comunidad informática. A continuación, lo veremos en mayor detalle:

¿Qué es Hibernate?

Hibernate es un servicio ORM de persistencia y consultas a bases de datos implementado para Java. No se trata de una herramienta o servicio aparte sino de una implementación o proveedor de JPA (no son conceptos separados, sino que trabajan a la par).

El principal objetivo de Hibernate es el de mapear las clases del modelo de datos de una aplicación y así convertirlos o asociarlos a bases de datos, para ello, como se mencionó anteriormente, se utilizan annotations. Algunas de las más conocidas se explican a continuación:

- **@Entity:** Se utiliza para mapear todas las clases que se convertirán en entidades (tablas) en la futura base de datos.
- **@Table:** Se utiliza en conjunto con la annotation @Entity. Su principal función es la de mapear con una tabla de una base de datos en particular (en caso de que ya existiese) o establecer el nombre que queremos que la entidad tome como tabla en la base de datos. Su uso es opcional, en caso de que no la utilicemos, JPA tomará automáticamente como nombre de la tabla al nombre de la clase.
- **@Id:** Se utiliza para mapear las id de cada clase, las cuales se reflejarán en las bases de datos como primary keys (claves primarias).
- **@GeneratedValue:** Se utiliza en conjunto con @Id y permite establecer el tipo de secuencia o generación que va a tener una determinada id. Entre las principales estrategias de generación automática de secuencias que posee GeneratedValue se encuentran:
 - o **Auto:** Es la opción por defecto. Pensada principalmente para ids numéricas. Esta estrategia deja a criterio de Hibernate la forma de generación que considere mejor para la id con la que se esté trabajando.
 - o **Identity:** Generalmente son auto-incrementales (aumentan su valor de forma automática según un incremento que se establezca). Se utilizan principalmente para asignar a claves primarias ya existentes en una base de datos por las cuales Hibernate debe guiarse para continuar la secuencia.
 - o **Sequence:** Es uno de los tipos de generación de valor más utilizados. Permite generar secuencias numéricas. De forma automática hace el incremento de 1 en 1, pero puede ser personalizada según sea necesario.
 - o **Table:** Se utiliza para casos en donde es necesario asignar claves primarias para las entidades de una base de datos mediante los datos que se encuentren contenidos en una tabla, guardando en ésta el último valor utilizado como referencia.
- **@Column:** Se utiliza para mapear cada uno de los atributos de una clase con las columnas de una tabla. No es una annotation obligatoria, en caso de que no se la utilice, JPA toma de forma automática como nombre de columna al nombre del atributo de la clase en cuestión. Es una annotation principalmente pensada para el mapeo de atributos sobre columnas de tablas en bases de datos ya existentes.
- **@OneToOne, @OneToMany, @ManyToOne y @ManyToMany:** Son annotations utilizadas principalmente para el mapeo de relaciones entre clases, los cuales se traducirán a nivel de base de datos como relaciones entre tablas (uno a uno, uno a muchos, muchos a uno o muchos a muchos).
- **@JoinColumn:** Es utilizada para manifestar los distintos tipos de Join que sean necesarios entre campos de tablas

ABML con SpringBoot + JPA + Hibernate

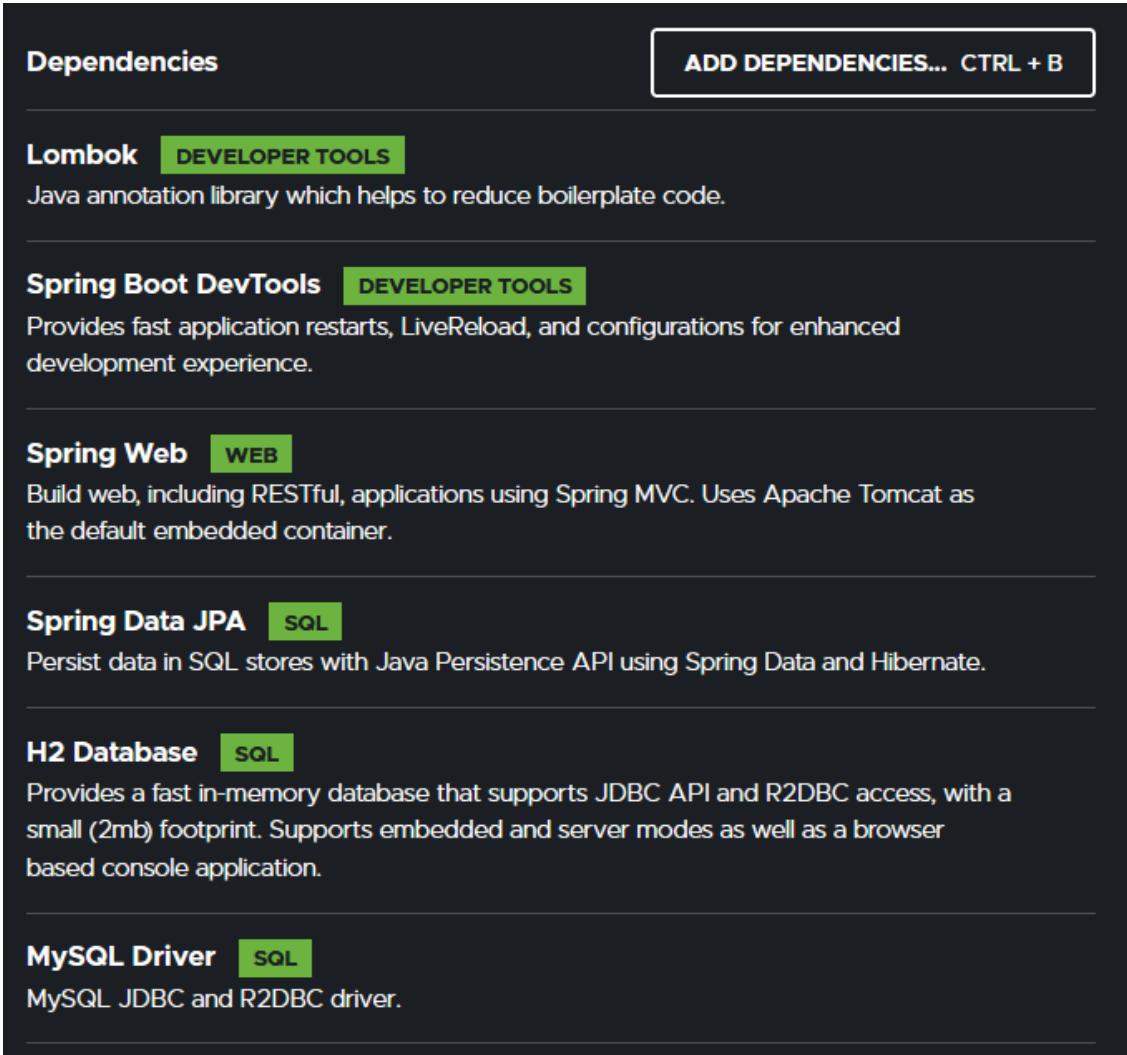
Conociendo ya los conceptos de JPA y su principal proveedor Hibernate, es posible implementar ahora una aplicación sencilla que represente un ABML (Alta, Baja, Modificación y Lectura).

Como resumen en primer lugar trabajaremos con las siguientes herramientas:

1. Base de datos My SQL (administrada con PHPMy admin)
2. Netbeans 12
3. Spring Boot + Initializr
4. JPA + Hibernate

PASO 1

Ir a inicializar y crear un nuevo proyecto Spring Boot teniendo en cuenta las dependencias especificadas en la Ilustración. Luego, abrirlo en Netbeans y dejar descargar las correspondientes dependencias mediante Maven:



The screenshot shows the 'Dependencies' section of the Spring Initializr web application. At the top right is a button labeled 'ADD DEPENDENCIES... CTRL + B'. Below this, several dependency categories are listed, each with a category tag and a description:

- Lombok** (DEVELOPER TOOLS): Java annotation library which helps to reduce boilerplate code.
- Spring Boot DevTools** (DEVELOPER TOOLS): Provides fast application restarts, LiveReload, and configurations for enhanced development experience.
- Spring Web** (WEB): Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
- Spring Data JPA** (SQL): Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
- H2 Database** (SQL): Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.
- MySQL Driver** (SQL): MySQL JDBC and R2DBC driver.

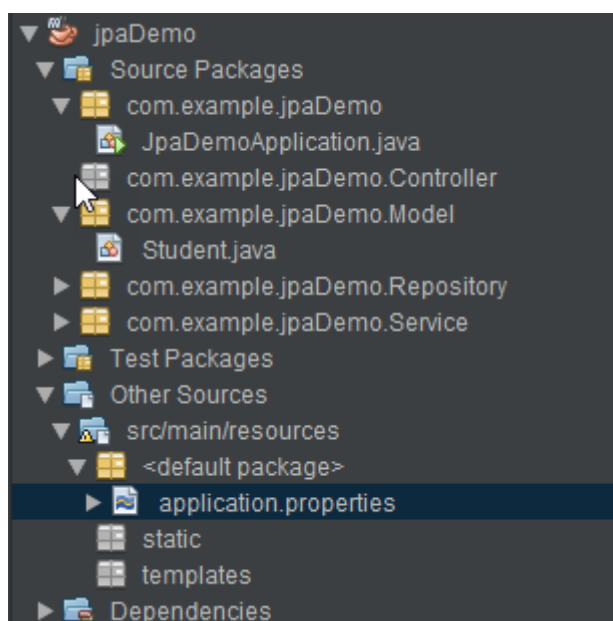
PASO 2

Crear una base de datos MySQL llamada prueba_jpa y asignar un usuario y contraseña con todos los privilegios, para ello, utilizar el servidor y gestor de preferencia. Asegurarse que la base de datos se encuentre totalmente vacía (sin ninguna tabla creada). En este ejemplo se utiliza:

1. Xampp Server
2. Nombre base de datos: prueba_jpa
3. Usuario: admin y Contraseña: admin

PASO 3

Ir al archivo **application.properties** y configurar los parámetros tal y como se muestra en la ilustración. Tener en cuenta de reemplazar los valores según se hayan especificado en la base de datos creada:



```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost:3306/prueba_jpa?useSSL=false&serverTimezone=UTC
spring.datasource.username=admin
spring.datasource.password=admin
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
```

El archivo **application.properties** permite establecer diversas configuraciones sobre una aplicación realizada con Spring Boot, en este caso está siendo utilizado para establecer los parámetros de la base de datos. Tal como se puede visualizar en la ilustración anterior, los parámetros que se configuran son los siguientes:

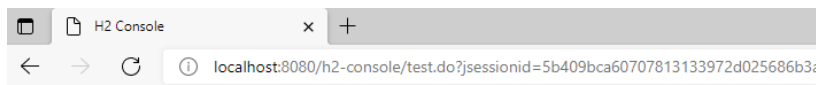
- **spring.jpa.hibernate.ddl-auto:** establece la estrategia que utilizarán JPA y Spring para el manejo de tablas. Al colocar el valor en Update se establece que la estrategia será de actualización de tablas.
- **spring.datasource.url:** establece la dirección donde se encuentra alojada la base de datos. Es de vital importancia en este apartado colocar correctamente el nombre de la base de datos (en este ejemplo prueba_jpa) y el puerto donde se encuentra levantada la misma (generalmente 3306 en bases de datos mysql). Por otro lado la especificación `?useSSL=false&serverTimezone=UTC` establecen que para la conexión no se utilizará [SSL](#) y que se tendrá en cuenta una zona horaria estándar para el servidor (esta configuración es de gran importancia para posibles errores en cuanto a diferencias de fecha y hora entre el servidor de base de datos y la aplicación).
- **spring.datasource.username:** se utiliza para parametrizar el nombre de usuario de la base de datos (configurado anteriormente en la misma).
- **spring.datasource.password:** se utiliza para parametrizar la contraseña asociada al nombre de usuario especificado anteriormente.
- **spring.jpa.database-platform:** en este apartado se debe especificar el dialecto que se utilizará para comunicar la aplicación con la base de datos MySQL creada. Es importante tener en cuenta la versión del driver de MySQL que se esté utilizando en este momento, para ello, podemos verificar la misma en el archivo pom.xml:

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope>
</dependency>
```

En caso de que en el archivo pom.xml no se especifique ninguna versión en particular, podemos tomar la más reciente. En este ejemplo utilizaremos la versión 8, mediante "org.hibernate.dialect.MySQL8Dialect".

PASO 4

- Ejecutar la aplicación. **Nota:** Antes de esto, asegurarse de que el servidor de la base de datos esté activo y ejecutándose.
- Una vez ejecutada, intentar acceder a la url: <http://localhost:8080/h2-console/>
- Si todo funciona correctamente será posible visualizar una pantalla como la que puede observarse en la siguiente ilustración.
- H2, es una base de datos "lógica" y "embebida" (de alguna manera) que ofrece Spring para poder trabajar sobre ella sin la necesidad de tener un servidor de base de datos aparte. No será utilizada en este momento porque ya se cuenta con una base de datos MySQL creada, pero se hace referencia por ser una herramienta de gran ayuda.



English Preferences Tools Help

Login

Saved Settings: Generic H2 (Embedded) ▼

Setting Name: Generic H2 (Embedded) Save Remove

Driver Class: com.mysql.jdbc.Driver

JDBC URL: jdbc:mysql://localhost:3306/prueba_jpa

User Name: admin

Password:

Connect Test Connection

Con la consola H2 podemos verificar la correcta conexión con nuestra base de datos mediante test connection (No olvidar colocar correctamente los datos correspondientes)

Test successful

PASO 5

Una vez probada la correcta conexión con la base de datos, procederemos a crear una clase **Persona** mediante la cual realizaremos un ABML. Para ello crearemos las siguientes clases/interfaces:

1. En la capa **model**: clase **Persona**.
2. En la capa **repository**: interface **PersonaRepository**.
3. En la capa **service**: clase **PersonaService**.

Como resultado de este paso, es posible ejecutar la aplicación y la misma, mediante JPA deberá crear dos tablas:

1. La tabla **persona**
2. La tabla **hibernate_sequence** (para el manejo de secuencias)

```
@Getter @Setter
@Entity
public class Persona {

    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE)
    private Long id;
    private String nombre;
    private String apellido;
    private int edad;
}
```

```
import com.jpa.pruebahibernate.model.Persona;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

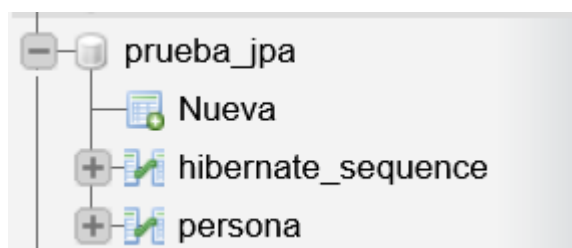
@Repository //mapeamos como repositorio
//la interface extiende de JpaRepository (que maneja repositorios JPA)
//en los parámetros <> deben ir: <clase a persistir, tipo de dato de su ID>
public interface PersonaRepository extends JpaRepository <Persona, Long>{

}
```

```
@Service
public class PersonaService {

    @Autowired
    private PersonaRepository persoRepository;

}
```



PASO 6

A partir de la correcta creación de la clase de secuencias de hibernate y de la clase Persona, es posible llevar a cabo los métodos correspondientes para lograr un CRUD (ABML en inglés). Para ello, es necesario configurar los servicios que tendrá disponible la aplicación, para lo cual se deberá crear una interface

IPersonaService la cual contendrá los métodos necesarios para el ABML; a su vez, la clase PersonaService implementará dicha interfaz para poder especificar cada uno de los métodos en mayor detalle.

```
public interface IPersonaService {  
  
    //método para traer todas las personas  
    public List<Persona> getPersonas ();  
  
    //método para dar de alta una persona  
    public void savePersona (Persona perso);  
  
    //método para borrar una persona  
    public void deletePersona (Long id);  
  
    //método para encontrar una persona  
    public Persona findPersona (Long id);  
  
}
```

```
@Service  
public class PersonaService implements IPersonaService{  
  
    @Autowired  
    private PersonaRepository persoRepository;  
  
    @Override  
    public List<Persona> getPersonas() {  
        List<Persona> listaPersonas = persoRepository.findAll();  
        return listaPersonas;  
    }  
  
    @Override  
    public void savePersona(Persona perso) {  
        persoRepository.save(perso);  
    }  
  
    @Override  
    public void deletePersona(Long id) {  
        persoRepository.deleteById(id);  
    }  
  
    @Override  
    public Persona findPersona(Long id) {  
        //acá si no encuentro la persona, devuelvo null por eso va el orElse  
        Persona perso = persoRepository.findById(id).orElse(null);  
        return perso;  
    }  
}
```

Con la implementación de cada uno de los métodos de la imagen anterior se tiene la lógica de negocio completa para llevar a cabo un ABML completo. Ahora, es necesario configurar el controller para recibir las solicitudes para llevar a cabo estas acciones.

PASO 7

Será necesario armar en el controller cada uno de los endpoints necesarios para recibir las requests para llevar a cabo las operaciones ABML. En la ilustración siguiente se puede visualizar un ejemplo completo de la clase controladora con cada uno de los endpoints creados según el método HTTP a utilizar:

```
@RestController
public class PersonaController {

    @Autowired
    private IPersonaService interPersona;

    @GetMapping ("/personas/traer")
    public List<Persona> getPersonas() {

        return interPersona.getPersonas();
    }

    @PostMapping ("/personas/crear")
    public String createStudent(@RequestBody Persona perso) {

        interPersona.savePersona(perso);
        //devuelve un string avisando si creó correctamente
        return "La persona fue creada correctamente";
    }

    @DeleteMapping ("/personas/borrar/{id}")
    public String deletePersona (@PathVariable Long id) {

        interPersona.deletePersona(id);
        //devuelve un string avisando si eliminó correctamente
        return "La persona fue eliminada correctamente";
    }

    @PutMapping ("personas/editar/{id}")
    public Persona editPersona (@PathVariable Long id,
                                @RequestParam ("nombre") String nuevoNombre,
                                @RequestParam ("apellido") String nuevoApellido,
                                @RequestParam ("edad") int nuevaEdad) {

        //busco la persona en cuestión
        Persona perso = interPersona.findPersona(id);

        //esto también puede ir en service
        //para desacoplar mejor aún el código en un nuevo método
        perso.setApellido(nuevoApellido);
        perso.setNombre(nuevoNombre);
        perso.setEdad(nuevaEdad);

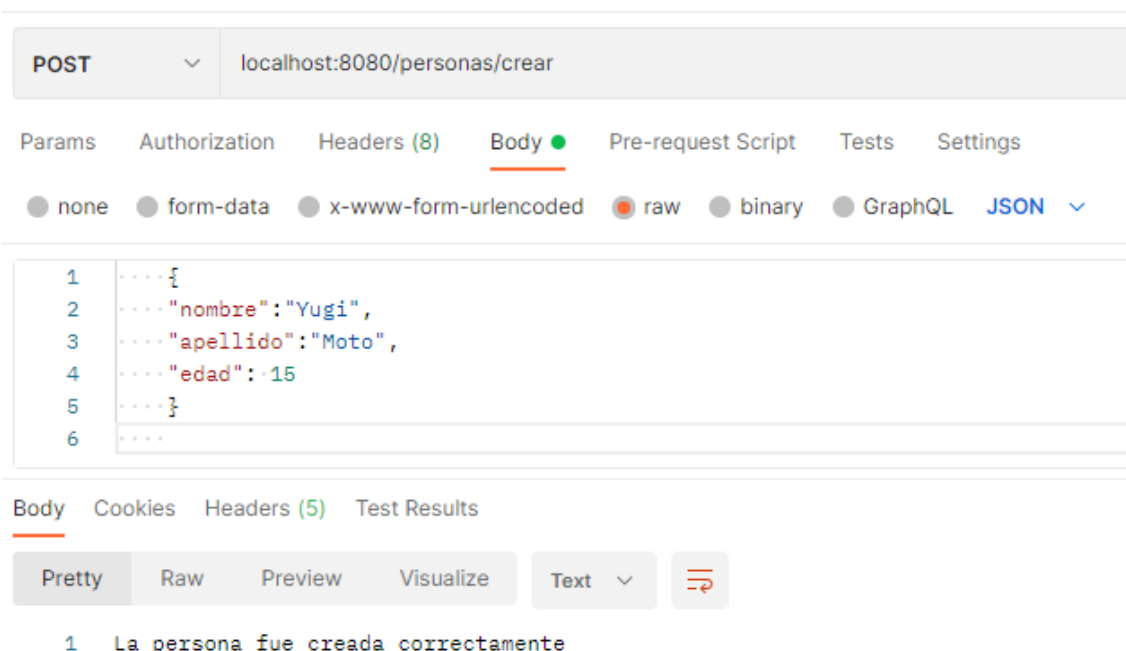
        interPersona.savePersona(perso);
        //retorna la nueva persona
        return perso;
    }
}
```

Una vez configurado todo, ejecutaremos la aplicación creada y probaremos cada uno de los endpoints mediante Postman.

Probando el ABML con Posman

Alta de una nueva persona

Daremos de alta una nueva persona, para ello en Postman seleccionaremos el método POST, incluiremos los datos de la nueva persona en el body mediante formato JSON y lo enviamos a nuestra aplicación. Si todo sale bien, la aplicación nos responderá el mensaje que parametrizamos. Al mismo tiempo, en nuestra base de datos tendremos un nuevo registro:



Procederemos a crear un par de personas más para poder contar con mayor cantidad de datos para probar las siguientes consultas. Los datos de las personas a agregar se pueden observar en la siguiente ilustración:

```

{
  "nombre": "Seto",
  "apellido": "Kaiba",
  "edad": 17
}

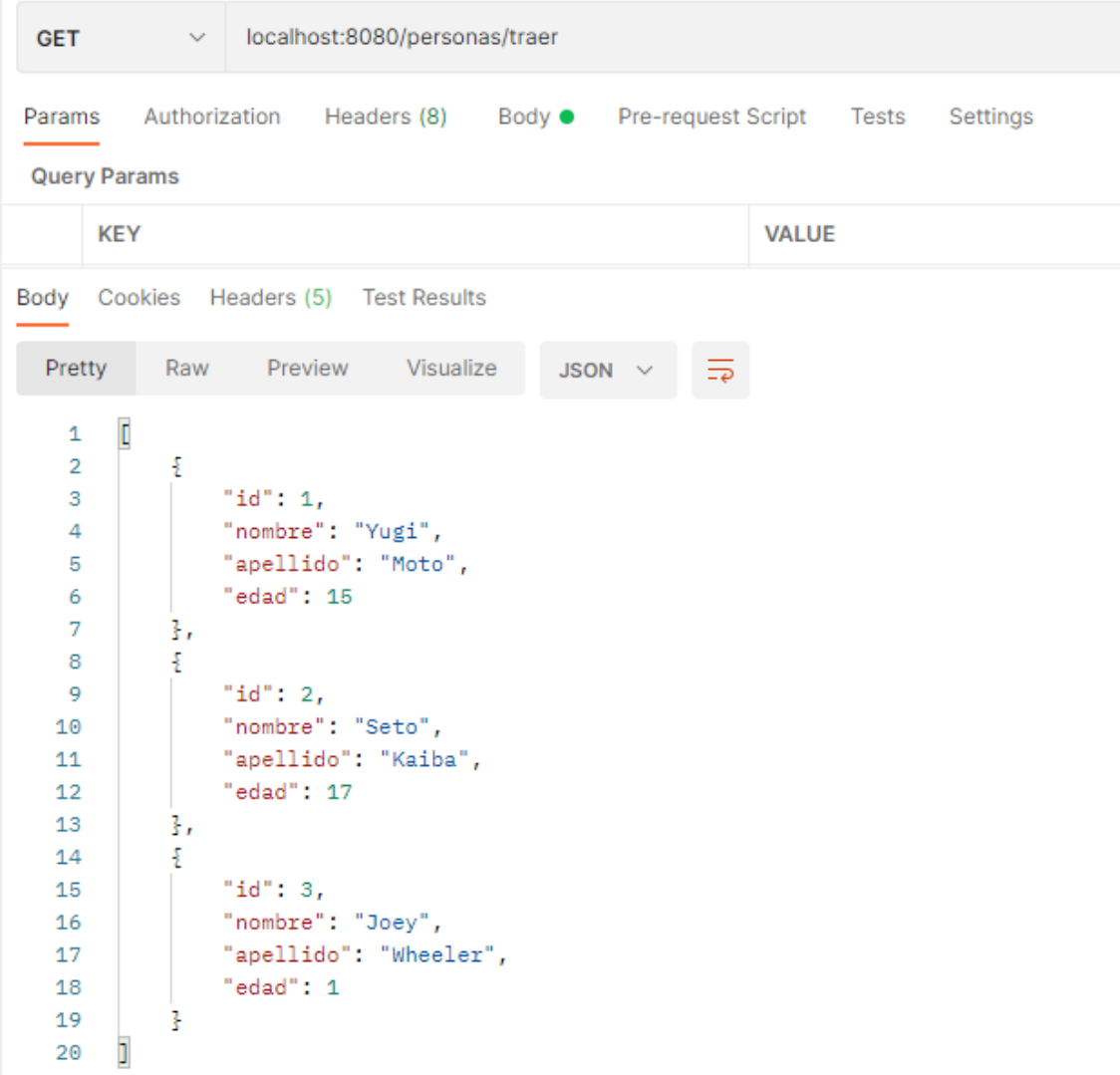
{
  "nombre": "Joey",
  "apellido": "Wheeler",
  "edad": 1
}

```

Lectura de la lista de personas

Para probar el endpoint que nos permite obtener la lista completa de personas, utilizaremos Postman y simularemos una solicitud GET. Si todo sale bien,

obtendremos como resultado un JSON con todas las personas cargadas en la base de datos:



GET localhost:8080/personas/traer

Params Authorization Headers (8) Body ● Pre-request Script Tests Settings

Query Params

KEY	VALUE
-----	-------

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```

1  [
2    {
3      "id": 1,
4      "nombre": "Yugi",
5      "apellido": "Moto",
6      "edad": 15
7    },
8    {
9      "id": 2,
10     "nombre": "Seto",
11     "apellido": "Kaiba",
12     "edad": 17
13   },
14   {
15     "id": 3,
16     "nombre": "Joey",
17     "apellido": "Wheeler",
18     "edad": 1
19   }
20 ]

```

Baja o Eliminar una persona

Para llevar a cabo la eliminación de una persona, realizaremos una request mediante postman y el método HTTP DELETE. Además de esto, indicaremos en la URL la id de la persona que queramos borrar. En este caso elegiremos la id 3. Veamos un ejemplo de la solicitud:

CRUD JPA PERSONAS / Eliminar

DELETE
localhost:8080/personas/borrar/3

Params
Authorization
Headers (6)
Body
Pre-request Script
Tests
Settings

Query Params

KEY	VALUE
Key	Value

Body
Cookies
Headers (5)
Test Results

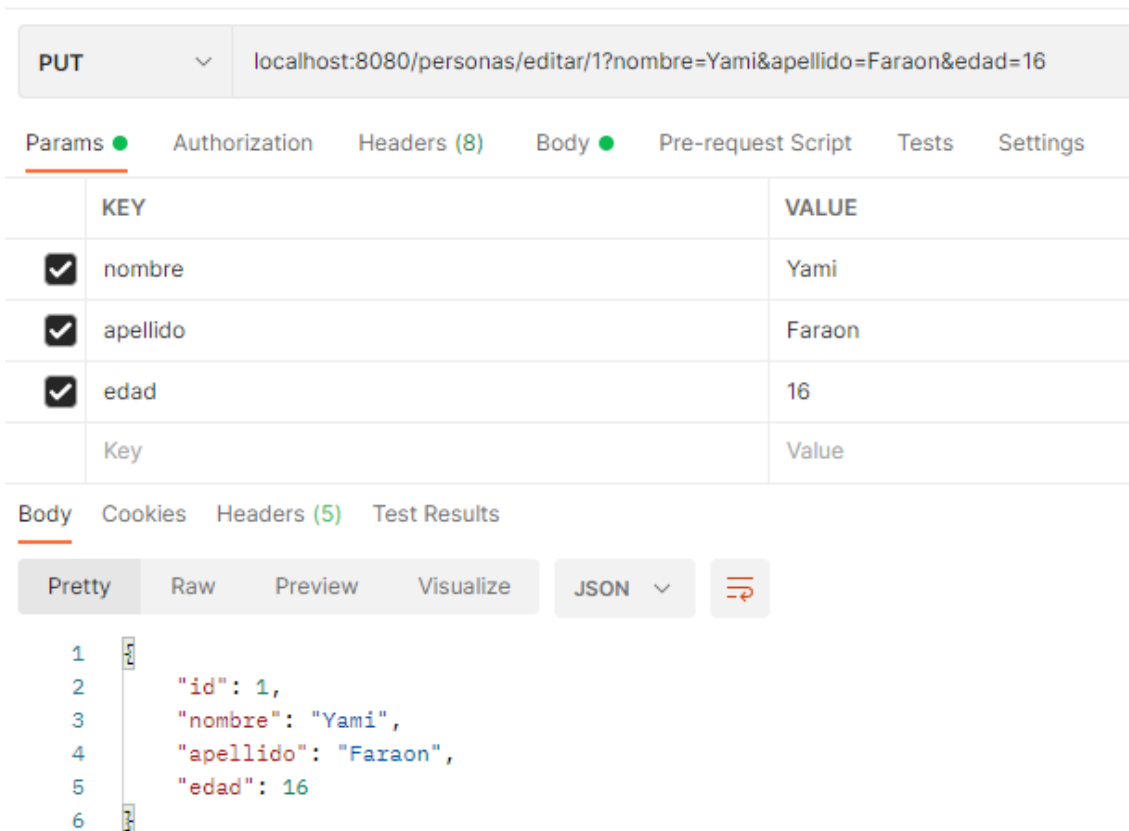
Pretty
Raw
Preview
Visualize
Text
⌵
⌵

```
1 La persona fue eliminada correctamente
```

Si volvemos a realizar una consulta GET para traer todas las personas, observaremos que la persona con id 3 ya no existe en la base de datos.

Modificación o Actualización de una persona

Supongamos que una de las personas tiene sus datos ingresados incorrectamente y queremos corregirlos. Para ello simularemos una solicitud UPDATE mediante POSTMAN, para solicitar la modificación a nuestra aplicación proporcionando la id, en este caso, la persona 1:




PUT ▼ localhost:8080/personas/editar/1?nombre=Yami&apellido=Faraon&edad=16

Params ● Authorization Headers (8) Body ● Pre-request Script Tests Settings

	KEY	VALUE
<input checked="" type="checkbox"/>	nombre	Yami
<input checked="" type="checkbox"/>	apellido	Faraon
<input checked="" type="checkbox"/>	edad	16
	Key	Value

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON ▼ 

```

1  {
2    "id": 1,
3    "nombre": "Yami",
4    "apellido": "Faraon",
5    "edad": 16
6  }
```

Al utilizar Postman, es necesario especificar los nuevos parámetros en la pestaña “params”, esto agregará los mismos de forma automática a la URL sin que tengamos que hacerlo manualmente. En la ilustración se ve cómo el resultado de la request es el objeto creado (tal cual como fue parametrizado en el endpoint).

Con todos estos pasos tendremos como resultado un ABML (CRUD) completo implementando Spring Boot + JPA + Hibernate.