# U.T. 5: CREACION DE APLICACIONES WEB. SPRING BOOT



https://spring.io/guides

## 1. Spring framework

Spring es un *framework* de Java para el desarrollo de aplicaciones y servicios web. En nuestro caso, lo que queremos construir es una pequeña aplicación web con una Base de Datos y que podamos, si así lo queremos, proporcionar algo de lógica en el lado servidor cuando sea necesario. Para eso utilizaremos *Spring Boot* que es una parte de este *framework* que facilita bastante el trabajo para casos como el que a nosotros nos interesa.

Para eso, lo primero que haremos será utilizar el Spring Initializr para preparar el proyecto inicial sobre el que luego diseñaremos nuestra pequeña aplicación web.

Una vez tengamos creado el proyecto inicial, podemos empezar a trabajar en él. En este caso se trata de crear una aplicación web para una tienda de comercio electrónico para la que haremos, a modo de ejemplo,

la parte donde se lista el catálogo de productos, así como las operaciones CRUD que se consideren oportunas.

Spring Boot es una de las tecnologías principales dentro del mundo de Spring. ¿Qué es y cómo funciona Spring Boot? Para entender el concepto primero debemos reflexionar sobre cómo construimos aplicaciones con Spring Framework.

Fundamentalmente, para desarrollar una aplicación existen tres pasos a realizar:

1. Crear un proyecto Maven/Gradle y descargar las dependencias necesarias.
2. Desarrollar la aplicación, y
3. Desplegar la aplicación en un servidor.

Si nos paramos a pensar un poco, únicamente el paso segundo es una tarea de desarrollo. Los otros pasos están más orientados a infraestructura. Ahí es donde *SpringBoot* nace con la intención de simplificar los pasos 1 y 3, para que nos podamos centrar en el desarrollo de nuestra aplicación.

Antes de comenzar, las dependencias requeridas serán *Lombok* (projectLombok) para las anotaciones propias de Spring, el controlador de PostgreSQL, *Spring Web* (spring-boot-starter-web), *Spring Data JPA* y *Thymeleaf*, generador de HTML y otros formatos (*spring-boot-starter-thymeleaf*).

Para IntellijIDEA será conveniente instalar el plugin *Spring Boot Helper* desde el menú *File --> Settings*.

# 2. CONFIGURACIÓN DEL SERVIDOR

Lo primero de todo será editar el fichero de configuración *application.properties* (en src/main/resources) del proyecto para personalizarlo a nuestro caso:

```
# Configuración para el acceso a la Base de Datos
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.globally_quoted_identifiers=true

# Puerto donde escucha el servidor una vez se inicie
server.port=8080

# Datos de conexion con la base de datos MySQL
spring.datasource.url=jdbc:mysql://localhost:3306/myshoponline
spring.datasource.username=myshopuser
spring.datasource.password=mypassword
spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver
```

Observa que este fichero contendrá parámetros que habitualmente incluíamos, para Hibernate, en el fichero *hibernate.cfg.xml*.

# 3. DEFINIR LA BASE DE DATOS

Hay que tener en cuenta que *Spring* utiliza por debajo el *framework* de *Hibernate* para trabajar con la Base de Datos. Eso nos va a permitir trabajar con nuestras clases Java directamente sobre la Base de Datos, ya que será *Hibernate* quien realizará el mapeo entre el objeto Java (y sus atributos) y la tabla relacional (y sus columnas) a la hora de realizar consultas, inserciones, modificaciones o borrados. E incluso a la hora de crear las tablas, puesto que bastará con definir nuestro modelo de clases con las anotaciones apropiadas para que Spring pueda crearlas en base a éstas (y porque tenemos la opción *spring.jpa.hibernate.ddl-auto=update* en el fichero de configuración en tiempo de desarrollo).

Los posibles valores de dicha propiedad son

- none: para indicar que no queremos que genere la base de datos
- update: si queremos que la genere de nuevo en cada arranque

- validate: valor en producción para no modificar la base de datos
- create: si queremos que la cree pero que no la genere de nuevo si ya existe

Cuando ya no queramos que Spring genere automáticamente la base de datos en cada arranque (por ejemplo, en producción), podremos cambiar esa opción a valor *none* o *validate*.

Así, simplemente tenemos que crear la clase con los atributos y métodos que queramos, y añadir las anotaciones que orientarán a *Hibernate* para saber a qué tabla corresponden los objetos de la clase y a qué columnas sus atributos.

Para más simplicidad en el código usaremos la librería Lombok para la generación automática de *getters*, *setters* y constructores.

```
import lombok.*;

import javax.persistence.*;
import java.time.LocalDateTime;
/**
 * Producto de la tienda online
 *
 */
@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity(name = "productos")
public class Producto {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column
    private String name;
    @Column
    private String description;
    @Column
    private String category;
    @Column
    private float price;
    @Column(name = "creation_date")
    private LocalDateTime creationDate;
}
```

Todas las anotaciones Java del ejemplo anterior son clases que pertenecen al paquete *jakarta.persistence*. El significado de cada una de las anotaciones queda a interpretación del lector, por ser sus nombres descriptivos.

# 4. EL ACCESO A LA BASE DE DATOS

Ahora creamos la interfaz donde se definirán los métodos que permitirán acceder a la Base de Datos. En este caso nos basta con definir una interfaz que herede de *CrudRepository*. En su documentación (https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html) podemos ver que es una interfaz que incluye los métodos principales para las distintas operaciones CRUD como *count()*, diversos métodos de búsqueda *findXXX()*, de almacenado *save()* y de borrado *deleteXXX()*. Será el *framework* el que se encargue de su implementación. A nuestra interfaz la podemos mejorar con métodos propios, pero no será necesario.

Además *CrudRepository* también tenemos *JpaRepository* que es una extensión (extends) de la anterior, aunque, a nosotros, nos bastará con la primera.

```java
/**
 * Repositorio de Productos
 */
@Repository
public interface ProductoRepository extends CrudRepository<Producto, Long> {

}
```

# 5. IMPLEMENTACIÓN DE LA LÓGICA DE NEGOCIO: LOS SERVICES

Los *Services* serán la capa de nuestra aplicación web donde implementaremos toda la lógica de negocio.

Definiremos una interfaz con todos los métodos que necesitemos; por ejemplo:

```java
public interface ProductoService {
     ArrayList<Producto> findAll();
     Optional<Producto> findById(Long id);
     void saveProducto(Producto producto);
     Producto updateProducto(Long id, Producto producto);
     void deleteProductoById(Long productoId);
}
```

Que implementaremos en la clase *ProductServiceImpl* siguiente. Observa que se están implementando las distintas operaciones CRUD.

```java
@Service
public class ProductoServiceImpl implements ProductService {
    @Autowired
    private ProductoRepository productoRepository;
    @Override
    public ArrayList<Producto> findAll() {
        return (ArrayList<Producto>) productoRepository.findAll();
    }
    @Override
    public Optional<Producto> findById(Long id) {
        return productoRepository.findById(id);
    }
    @Override
    public void saveProduct(Producto producto) {
        productoRepository.save(producto);
    }
    @Override
    public Producto updateProducto(Long id, Producto producto)
    {
        Producto prodDB = productoRepository.findById(id).get();
            // Si el nombre de Producto no está a null ni como String vacío
        if (Objects.nonNull(producto.getName()) &&
             !"".equalsIgnoreCase(producto.getName()))
            prodDB.setName(producto.getName());
            // Si el precio de Producto no está a null
        if (Objects.nonNull(producto.getPrice()))
            prodDB.setPrice(producto.getPrice());
        productoRepository.save(prodDB);
        return prodDB;
```

```
    }
    // delete operation
    @Override
    public void deleteProductoById(Long productoId) {
        productoRepository.deleteById(productoId);
    }
}
```

# 6. IMPLEMENTACIÓN DEL CONTROLADOR

Las clases que definimos como controladoras son responsables de procesar las llamadas entrantes (request) que ingresan a nuestra aplicacion, validarlas y dar una respuesta (response). El controlador podrá ser implementado como un controlador simple o un controlador REST, indicados mediante sus correspondientes anotaciones. Veámoslas.

Anotación **@Controller**: esta anotación indica que una clase en particular cumple la función de un controlador. La anotación *Controller* generalmente se usa en combinación con métodos de controlador anotados basados en la anotación *@RequestMapping*. Sólo se puede aplicar a clases, que en ese caso queda marcadas como controlador de solicitudes web y, principalmente, con aplicaciones Spring MVC. Esta anotación actúa como un estereotipo de la clase anotada, indicando su rol. El despachador escanea dichas clases anotadas en busca de métodos asignados y detecta anotaciones *@RequestMapping*.

Un "Rest Controller", por su parte, es un subtipo de controlador que recibe peticiones con un formato específico que cumple con formatos de solicitud RESTful principalmente en JSON , aunque también se pueden usar otros formatos como HTML, XML, o simplemente texto. La anotación **@RestController** también se usa a nivel de clase y permite que la clase maneje las solicitudes realizadas por el cliente para servicios RESTful. Esta anotación equivale al uso conjunto de *@Controller* y *@ResponseBody*.

REST (Representational State Transfer) es un estilo de arquitectura a la hora de realizar una comunicación entre cliente y servidor en nuestras aplicaciones. Las características de estos servicios web son:

- permiten listar, crear, leer, actualizar y borrar información
- para las operaciones anteriores necesitan una URL y un método HTTP para accederlas
- usualmente regresan la información en formato JSON; cada método de respuesta devuelve un objeto de dominio en lugar de una vista.
- retornan códigos de respuesta HTML, por ejemplo 200, 201, 404, etc

Este tipo *RestController* permite manejar todas las solicitudes GET, POST, DELETE y PUT. Es la que vamos a utilizar. El siguiente artículo trata sobre los diferentes métodos HTTP en servicios REST: https://www.dariawan.com/tutorials/rest/http-methods-spring-restful-services/.

Vamos ya, por último, a crear la clase que hará de `Controller` de la aplicación. En ella introduciremos los métodos con las operaciones que queremos que nuestros usuarios puedan realizar, programaremos la lógica que necesitemos y accederemos a los datos a través del `ProductRepository` que hemos creado en el paso anterior.

Para el uso de *@RestController* el código quedaría:

```
/**
 * Controlador para la web
 */
@RestController
public class WebController {
    @Autowired
    private ProductoService productoService;

    // Operación de guardar
```

```
    @PostMapping("/alta")
    public void saveProducto(@RequestBody Producto producto) {
        productoService.saveProducto(producto);
    }

    // Read operation
    @GetMapping("/productos")
    public List<Producto> findAll() {
        return productoService.findAll();
    }

    // Update operation
    @PutMapping("/productos/{id}")
    public Producto updateProducto(@PathVariable("id") Long id,
              @RequestBody Producto producto) {
        return productoService.updateProducto(id, producto);
    }

    // Delete operation
    @DeleteMapping("/productos/{id}")
    public String deleteProductoById(@PathVariable("id") Long Id) {
        productoService.deleteProductoById(Id);
        return "Deleted Successfully";
    }
}
```

Para entender el anterior fragmento de código conviene tener en cuenta lo siguiente:

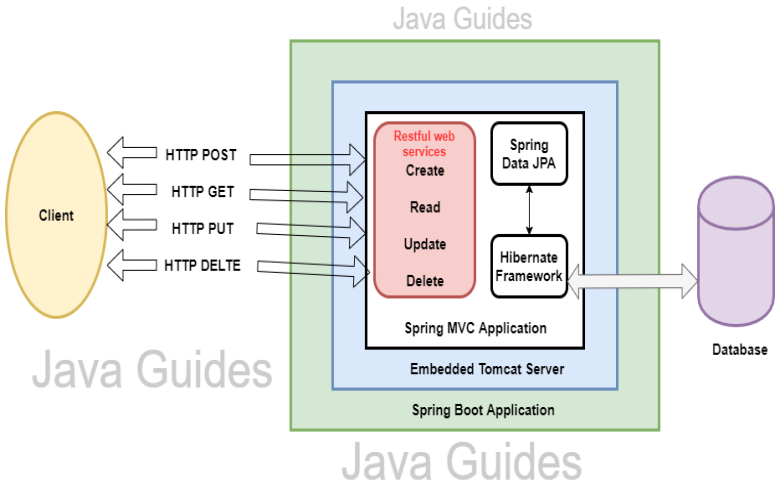
- cada método anotado define un punto de acceso que podrá ser invocado por otra aplicación o cliente
- las anotaciones *@GetMapping, @PostMapping, @PutMapping, @PatchMapping, @DeleteMapping* definen el método (GET, POST, PUT, PATCH, DELETE) y la URL de cada punto
- Si el acceso debe utilizar parámetros en la ruta (Path Params) vendrán definidos en la URL y también en el método como *@PathVariable*
- Si el acceso ha de utilizar parámetros en la consulta (Query Params) vendrán definidos solamente en el método como *@RequestParam* (haciendo uso del formato ?variable=valor al final de una URL).
- si el punto de acceso ha de utilizar parámetros en el cuerpo de la petición (Body Params) vendrán definidos solamente en el método como *@RequestBody*

La anotación *@GetMapping* garantiza que las solicitudes GET de HTTP a, en este caso, "/productos" se asignen al método correspondiente (en este caso *findAll()*). Igual sucede con *@PostMapping* y POST, así como *@PutMapping* y *DeleteMapping*. También existe la anotación *@RequestMapping* de la que las anteriormente citadas derivan y puede servir como sinónimo. Por ejemplo, *@RequestMapping(method=GET)* sería equivalente a *@GetMapping*.

Una diferencia clave entre un controlador MVC tradicional y el controlador de servicio web RESTful es la forma en la que se crea el cuerpo de la respuesta HTTP. En lugar de confiar en una tecnología de visualización para realizar la representación del lado del servidor de los datos de respuesta en HTML, este tipo de controlador rellena y retorna un objeto. Los datos del objeto se escribirán directamente en la respuesta HTTP como JSON.

https://www.arquitecturajava.com/rest-http-return-codes-y-sus-curiosidades/

# 7. EJECUCIÓN DE LA APLICACIÓN WEB




Para poder ejecutar la aplicación crearíamos *main* con la anotación *@SpringBootApplication* y el siguiente contenido.

```
@SpringBootApplication
public class App {
        public static void main( String[] args )
        {
                SpringApplication.run(App.class, args);
        }
}
```

Una vez terminado todo, para lanzar el servidor tenemos tres opciones:

- desde el propio IDE, ejecutando "mvn spring-boot:run", requerirá añadir el *plugin* correspondiente en el *pom.xml*.
- utilizando el jar que podemos generar con el comando "mvn package" y ejecutarlo con el comando "java -jar". El ".jar" generado lo podremos encontrar en la carpeta "target"
- utilizar la opción "Run" de nuestro IDE.

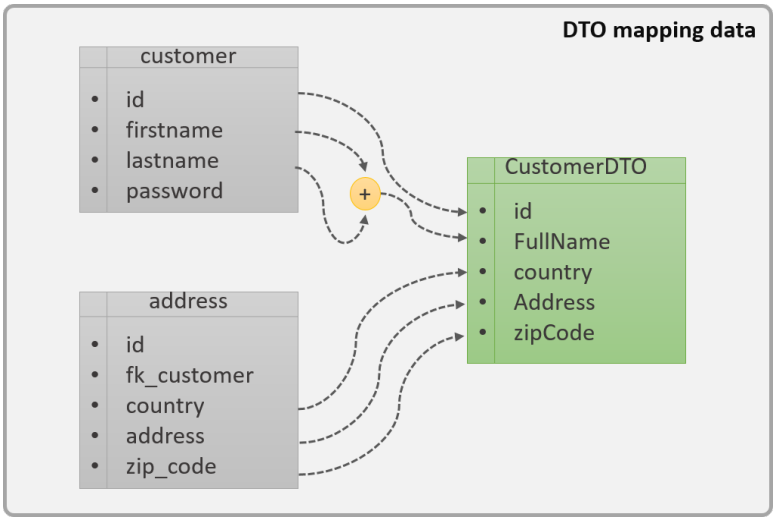
Finalmente interactuaríamos con la aplicación desde una herramienta gráfica como *Postman* o, alternativamente desde el terminal, con *curl*.

```
curl -X GET localhost:8080/productos

curl -X POST localhost:8080/alta -d '{"descripcion":"sierra","precio":80}' -H "Content-Type: application/json"

curl -X PUT localhost:8080/productos/1 -d '{"descripcion":"alicates","precio":28}' -H "Content-Type: application/json"

curl -X DELETE localhost:8080/productos/1
```

# 8. APIS REST CON RELACIONES

Para crear un proyecto de este tipo podemos hacer uso de Bootify (https://bootify.io). Es un servicio que simplifica el desarrollo en Spring Boot al generar muchos modelos de código, lo cual permite centrarnos en el trabajo lógico. Abrimos el sitio web de Bootify y hacemos clic en el botón "Start Project" en la esquina superior derecha. Con Bootify especificamos nuestras preferencias y, automáticamente, importamos las dependencias, similar a Spring Initializr. Pero además, con la opción "Entities" del proyecto, nos permite definir las entidades, con sus atributos y tipos, y las relaciones entre ellas. A partir de ellas, generará el modelo correspondiente y sus clases para el patrón de software DTO.

El patrón DTO permite crear un objeto plano (POJO) con un conjunto de atributos que puedan ser enviados o recuperados del servidor en una sola llamada, de forma que un DTO puede contener información de múltiples orígenes o entidades y concentrarlas en una única clase simple. El repositorio trabaja con las entidades y el controlador con las clases DTOs, mientras que el servicio realiza la transformación entre ellas.




Bootify puede incluso generar el servicio y el código a nivel de control para operaciones comunes de CRUD. Sólo tenemos que hacerle algunos ajustes al código generado. Para ello debemos seleccionar Maven como el tipo de compilación, la versión de Java, habilitar el uso de Lombok, seleccionar la base de datos, habilitar la creación de entidades, etc. Un ejemplo:

Versión de Java: 21

Habilita "Enable Lombok"

Base de datos: Postgresql, H2 ... (la que corresponda)

Habilita "add dateCreated/lastUpdated to entities"

Packages: "Technical"

Habilita "OpenAPI/Swagger UI"

Y, por último, faltaría añadir "org.springframework.boot:spring-boot-devtools" en "further dependencies".

# 9. PLANTILLAS HTML PARA RENDERIZAR EL CONTENIDO

Las plantillas HTML son los ficheros HTML que definen las diferentes secciones de la web en las que hay que renderizar contenido al usuario. Pueden contener solamente código HTML (en ficheros como el índice o index.html) o también etiquetas del motor de plantillas Thymeleaf para acceder a objetos Java que le han sido pasados como parámetro. De esta manera podemos incluir estos datos en el contenido que se mostrará al usuario (en el ejemplo, el fichero *catalogo.html* que veremos a continuación).

Estas plantillas suelen estar ubicadas en la carpeta *templates* o *webapp* del proyecto. El controlador es quien accede a ellas y espera encontrarlas allí directamente. Por eso simplemente es necesario facilitarle el nombre de la misma.

La página de inicio, como para la mayoría de servidores web, suele estar definida por la plantilla `index.html` y presenta cierto contenido inicial en HTML al usuario:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Inicio</title>
</head>
<body>
```

```
    <h1>Bienvenidos a la tienda</h1>
    <p>Visita nuestro <a href="catalogo">catálogo</a> de productos</p>
</body>
</html>
```

La página donde se lista el contenido del catálogo de la tienda online, la plantilla `catalogo.html` contiene código HTML y también una serie de etiquetas del motor de plantillas *Thymeleaf* para acceder a la información que ha sido parámetrizada desde el controlador que veremos a continuación (`model.addAttribute("products", products)`).

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="UTF-8">
    <title>Catálogo</title>
</head>
<body>
    <h1>Nuestro catálogo de productos</h1>
    <ul th:each="producto : ${productos}">
        <li><p th:text="${producto.name}"></p></li>
    </ul>
</body>
</html>
```

## Controlador

El controlador sería algo como (observa el cambio de la anotación *@RestController* por *@Controller*):

```
/** Controlador para la web
 */
 @Controller
 public class WebController {
     @Autowired
     private ProductoService productoService;

     @RequestMapping(value = "/index")
     public String index(Model model) {
         return "index";
     }

     @RequestMapping(value = "/catalogo")
     public String catalogo(Model model) {
         Set<Product> productos = productoService.findAll();
         model.addAttribute("productos", productos);
         return "catalogo";
     }

 }
```

Por lo general los métodos del controlador admitirán como parámetro un objeto de la clase *Model* que es la plantilla que quedará asociada a él. De esa manera, podemos añadir atributos a dicha plantilla para que sea capaz de renderizar contenido de la parte Java. Lo haremos con el método *addAtribute()* de la clase *Model*. Cada método del controlador ha de retornar un objeto String que será el nombre de la plantilla que debe ser renderizada una vez se ejecute el método.

En este caso hemos creado dos métodos:

- String index(Model model)
- String catalogo(Model model)

Cada uno de los métodos tienen una URL de mapeo que marca el punto de entrada a la web para que se ejecute dicho método y se genere el código HTML correspondiente, que dependerá de la plantilla web devuelta (los métodos devuelven un String que es el nombre de la plantilla utilizada).

Así, según el código implementado, hay 2 secciones en la web accesibles por las siguientes URLs:

- http://localhost:8080: Página inicial que devuelve la plantilla *index.html*
- http://localhost:8080/catalogo: Página donde se lista el catálogo de productos de la web, con la plantilla *catalogo.html* que está parametrizada y permite mostrar información utilizando el motor de plantillas **Thymeleaf**.

Teniendo en cuenta esto, si desplegáramos la aplicación en un servidor, le asociáramos un dominio y modificáramos el puerto para que escuchara en el 80, podríamos acceder a las secciones a través de las siguientes URLs:

- http://www.mitienda.com
- http://www.mitienda.com/catalogo

Observa cómo para "/index" se retorna el nombre de la página. Sin ella tendríamos un error en la obtención de productos.

Recuerda que la diferencia entre las dos anotaciones para controlador es que *@RestController* es una anotación compuesta que en sí misma equivale a @Controller y @ResponseBody conjuntamente, y, por lo tanto, retorna directamente el cuerpo de la respuesta, frente a la resolución de la vista que se representa con una plantilla HTML en un controlador ordinario *@Controller*.

# 10. INVERSIÓN DE CONTROL E INYECCIÓN DE DEPENDENCIAS

Spring se suele definir como un *framework* para el desarrollo de aplicaciones y contenedor de inversión de control. Comprende diversos módulos que proveen un rango de servicios, pero el corazón de Spring Framework es su contenedor de inversión de control (IoC). Su función es instanciar, inicializar y conectar objetos de la aplicación, además de proveer una serie de características adicionales disponibles en Spring a través del tiempo de vida de los objetos.

¿Qué es la Inversión de Control (IoC)? La inversión de control es un principio de software, en el que el control de nuestros objetos es transferido a un contenedor o *framework*. Delegamos, por lo general, en el *framework* para que tome el flujo del programa. Este sistema tiene una serie de ventajas tales como:

- Desacopla el código.
- Mayor facilidad a la hora de testear debido a que se pueden probar partes aisladas.
- Tiene una gran modularidad.

Los objetos creados y gestionados por el contenedor se denominan objetos gestionados o "beans". Estos objetos son del tipo POJO. Para realizar su tarea el contenedor necesita información indicando cómo instanciar y conectar entre sí los *beans*. A esta información se la llama metadatos de configuración.

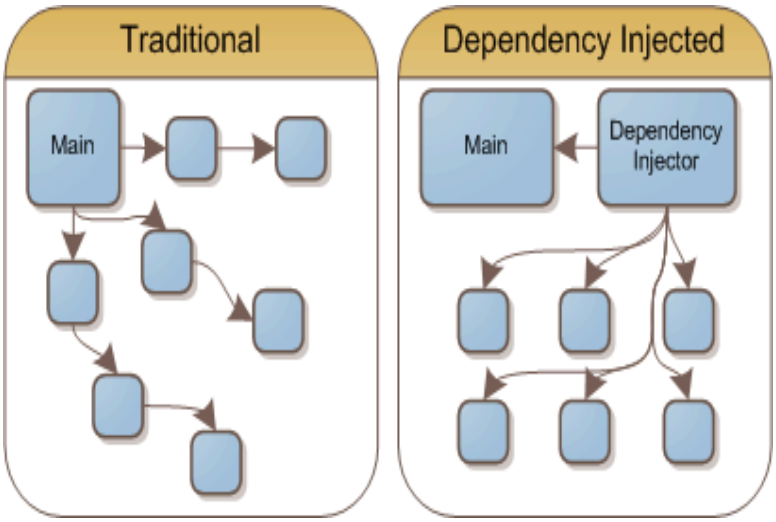
Hay distintas formas de proporcionar esta información, fundamentalmente dos: basándonos en XML, la primera, o en anotaciones, segunda opción. El contenedor es independiente del formato de los metadatos de configuración. El usuario puede usar el formato que desee e incluso mezclarlos en la misma aplicación.

**Inyección de Dependencias** (DI): es un patrón de software que implementa la inversión de control para resolver dependencias. Todas estas dependencias serán inyectadas por el contenedor que creará los *beans* necesarios, por Inversión de Control (IoC).

Podríamos definir y resumir este concepto como que la Inyección de Dependencias es el proceso de suministrar una dependencia externa a un componente de software. El contenedor IoC en Spring está

representado por la interfaz *ApplicationContext*, la cual es la responsable de configurar e instanciar todos los objetos (los beans) y manejar su ciclo de vida.

La inyección de dependencias es, quizás, la característica más destacable de Spring Framework, evitando que cada clase tenga que instanciar los objetos que necesite. En este caso, será Spring el que inyecte esos objetos, lo que quiere decir que es Spring el que creará los objetos, y cuando una clase necesite usarlos se le pasarán (como cuando le pasas un parámetro a un método).




Con la inyección de dependencias, en lugar de que sean las clases las encargadas de crear (instanciar) los objetos que van a usar (sus atributos), los objetos se inyectarán, mediante los métodos *setters* o mediante el constructor, en el momento en el que se cree la clase. Así cuando se quiera usar la clase en cuestión ya estará lista; sin usar DI, la clase necesitaría crear los objetos necesarios cada vez que se use.

Un ejemplo típico para ver su utilidad es el de una clase que necesita una conexión a base de datos. Sin DI, si varios usuarios necesitan usar esta clase se tendrán que crear múltiples conexiones a la base de datos con la consiguiente posible pérdida de rendimiento, pero usando la inyección de dependencia las dependencias de la clase (sus atributos) son instanciados una única vez, cuando se despliega la aplicación, y se comparten por todas las instancias de modo que una única conexión a la base de datos es compartida por múltiples peticiones. Este es el comportamiento por defecto, haciendo uso del patrón *Singleton*. La alternativa sería utilizar el patrón *Prototype* en el que se crea cada vez un objeto, diferentes objetos pero iguales entre sí. En definitiva, *Singleton* hace copias superficiales, mientras que *Prototype* hace copias en profundidad.

Y en lo referente al desacoplamiento, como no es necesario instanciar en una clase los objetos que necesita, si la clase que necesitamos cambia no es necesario modificar nada en el código de la clase que hacía uso de ella.

## 10.1. Inyección de dependencia mediante settters

Veamos un ejemplo en una aplicación de escritorio; en una aplicación web sería igual pero, como hay más archivos de configuración, es más fácil equivocarse. La única diferencia es que, para una aplicación de escritorio, hay que indicarle en *Main* dónde se encuentra el archivo en el que están declarados los beans, mientras que en una aplicación web se hace en el fichero *web.xml*.

En este primer ejemplo vamos a usar una clase Disco que entre sus atributos tendrá uno de la clase Autor y en Main mostraremos los datos del disco.

```
public class Disco {
    private String titulo;
    private Autor autor;
    private String genero;
    private String discografica;
    private double precio;
```

```
    // Getters y Setters, etc.
}

public class Autor {
    private String nombre;
    private String apellido;

    // Getters y Setters, etc.
}
```

La clase Disco es un simple POJO y, como vamos a inyectar las dependencias mediante *setters*, deberá disponer de ellas (lo mismo para Autor) y, para ello, en el programa debemos de cargar el fichero xml en el que estarán definidos los *beans* y luego obtenemos el que nos interesa (disco).

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {
    public static void main(String[] args) {
        BeanFactory factory = new
            ClassPathXmlApplicationContext("META-INF/spring/app-context.xml");
        Disco disco = (Disco) factory.getBean("disco");
        System.out.println("- " + disco.getTitulo());
        System.out.println("- " + disco.getAutor().getNombre() + " " +
             disco.getAutor().getApellido());
        System.out.println("- " + disco.getDiscografica());
        System.out.println("- " + disco.getGenero());
    }
}
```

Un detalle importante a destacar es que al método *getBean()* le paso disco en minúscula, mientras que la clase es Disco. No es un error, los *beans* se renombran cambiando la inicial de la clase en mayúscula a minúscula. Vamos con el *app-context.xml*, que será donde definamos los *beans* para poder inyectarlos donde lo necesitemos (este fichero en Intellij IDEA habrá de alojarse en el directorio "resources"):

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
  http://www.springframework.org/schema/context
     http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <bean id="disco" class="com.dam2.Disco">
        <property name="titulo" value="Power of three"/>
        <property name="autor" ref="autor"/>
        <property name="genero" value="jazz"/>
        <property name="precio" value="25"/>
    </bean>

    <bean id="autor" class="com.dam2.Autor" scope="prototype">
        <property name="nombre" value="Michel"/>
        <property name="apellido" value="Petrucciani"/>
```

```
    </bean>
</beans>
```

En primer lugar las clases se definen con la etiqueta <bean> y se deben de indicar el atributo *class* que indica el paquete con el nombre de la clase que será el *bean* y, si se quiere usar en otro lugar, el atributo *id* que nos servirá como identificador del *bean*. El atributo *scope* normalmente no lo utilizamos si nos basta con el patrón *Singleton*, que es el valor por defecto. También podríamos hacer uso de los atributos *init-method* y *destroy-method* para que ejecutara algún método cuando se crea el objeto y cuando se destruye.

```
<bean id = "disco" class = "com.dam2.Disco" init-method = "metodoInicial"
      destroy-method = "metodoFinal">
     ...
</bean>
```

Habría que definir, lógicamente, sendos métodos *metodoInicial()* y *metodoFinal()* en la clase POJO.

Dentro de cada *bean* se pueden indicar las propiedades que se quieren inyectar con la etiqueta *property*, con los atributo *name* para el nombre del atributo a inyectar y *value* con su valor. El atributo *ref* lo usaremos cuando queramos inyectar otro *bean*, es el caso de autor, y en el atributo *ref* debemos de poner el *id* del *bean* que queremos inyectarle, en este caso también es autor.

Y con esto, cuando ejecutamos la aplicación vemos que se muestran los valores introducidos en el código xml, sin tener que instanciar ningún objeto, ni de la clase Disco ni de la clase Autor.

## 10.2. Inyección de dependencia mediante el constructor

Además de inyectar mediante métodos *setters*, también se puede inyectar mediante constructor, aunque se suele considerar preferible la primera opción. Vamos a ver en este ejemplo cómo inyectar las propiedades del Autor mediante constructor para lo cual la clase Autor deberá de tener un constructor con los parámetros nombre y apellido y no hace falta que tenga métodos *setters*. No hace falta modificar otra cosa del resto del código Java.

Y el xml para poder inyectar mediante constructor en el *bean* autor es este:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
  http://www.springframework.org/schema/context
     http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <bean id="disco" class="com.dam2.Disco">
        <property name="titulo" value="Power of three"/>
        <property name="autor" ref="autor"/>
        <property name="genero" value="jazz"/>
        <property name="precio" value="25"/>
    </bean>

    <bean id="autor" class="com.dam2.Autor">
        <constructor-arg value="Michel"/>
        <constructor-arg value="Petrucciani"/>
    </bean>
</beans>
```

Los parámetros del disco los seguimos inyectando mediante *setters*, pero para el autor usaremos el constructor para lo que debemos de cambiar *property* por *constructor-arg* y solo le indicamos el *value*, pero

no hay que indicar el *name* ya que estos atributos se pasarán en orden al constructor (debemos de tener cuidado de ponerlos en el orden correcto).

## 10.3. Inyección de dependencia con anotaciones

En las dos formas anteriores era necesario indicar, en el archivo de configuración, qué *beans* podían ser inyectados en otros y sobre los que se querían inyectar. Mediante las anotaciones podemos hacer ambas cosas o solo una si se quiere; por ejemplo, podemos declarar los *beans* en el xml y usar anotaciones para inyectarlos en lugar de usar etiquetas del tipo <property name="autor" ref="autor">.

Una lista de las anotaciones principales sería:

@Component: sustituye la declaración del *bean* en el xml.

@Scope: se suele utilizar junto a la anterior anotación; sirve para indicar el ambito en el que se encontrará el *bean*. Por defecto sería *singleton*. En caso contrario utilizaríamos *@Scope("protoype")*.

@Autowired: sustituye la declaración de los atributos del *bean* en el xml para inyección de dependencias.

@Qualifier(«nombreBean»): para distinguir entre varias clases que implementan la misma interfaz, sirve para indicar qué clase es la que se debe inyectar. Irá junto a *@Autowired*.

@Required: indica si el atributo es obligatorio.

@Service, @Repository y @Controller: son estereotipos de @Component y se usan para indicar que la clase será un servicio (@Service), una clase de acceso a datos (@Repository) o un controlador (@Controller).

@PostConstruct: ejecuta el metodo con esta anotación despues de crear el objeto.

@PreDestroy: ejecuta el metodo con esta anotación antes de destruir el objeto. Estas 2 últimas anotaciones sólo se utilizan con *singleton*, no con *prototype*.

En este caso, nos va a bastar con las 2 anotaciones primeras:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <context:component-scan base-package="com.dam2"/>
</beans>
```

Observa cómo ya no definimos los *beans* en el *app-context.xml*, pero añadimos una única línea especificando el paquete base.

```
@Component
public class Disco {
    private String titulo = "Power of three";
    @Autowired
    private Autor autor;
    private String genero = "jazz";
    private String discografica = "ECM";
    private int precio = 25;
    // ...
}
```

```
@Component
public class Autor {
    private String nombre = "Michel";
    private String apellido = "Petrucciani";
}
```

# 11. CONTROLADOR MEJORADO

En este apartado vamos a ver cómo mejorar la respuesta de nuestro controlador, tanto en peticiones atendidas correctamente como en caso de error.

## 11.1. Usando ResponseEntity

*ResponseEntity* es una de las clases más habituales cuando uno utiliza Spring Framework con Servicios REST. Hasta ahora hemos realizado una aplicación básica, con servicios que dan respuestas básicas. Sin embargo, si queremos añadir flexibilidad y construir un servicio REST de mayor calidad será obligatorio conocer cómo funciona esta clase de Spring Framework.

Si recordamos la operación de alta de producto, en el controlador básico, era la siguiente:

```
// Operación de guardar
@PostMapping("/alta")
public void saveProduct(@RequestBody Producto producto) {
                productoService.saveProducto(producto);
}
```

Si hacemos una petición de inserción, por ejemplo (observa el parámetro -i):

```
curl -i -X POST localhost:8080/alta -d '{"descripcion":"sierra","precio":80}' -H "Content-Type: application/json"
```

Observaremos que el código de respuesta HTTP es el 200. La petición REST se realiza y obtendremos como resultado que todo ha ido bien. El 200 quiere decir que todo se ha ejecutado correctamente, pero no es suficiente ya que cuando uno realiza una petición POST debería recibir un status code de 201 si la petición se ha realizado de forma correcta . Un status code de 200 indica que todo ha ido correctamente pero un 201 es más especifico e indica que un nuevo recurso se ha añadido a la colección. Por tanto, el comportamiento de Spring por defecto, en este caso, no es el mejor de los posibles. ¿Cómo podemos cambiar este comportamiento?. Haciendo uso de la clase *ResponseEntity* que nos permite concretar más las respuestas.

```
// Operación de guardar
@PostMapping("/alta")
public ResponseEntity<Producto> saveProducto(@RequestBody Producto producto) {
    productoService.saveProducto(producto);
    return new ResponseEntity<Producto>(producto,null,HttpStatus.CREATED);
  }
```

Ahora cuando nosotros realicemos una petición POST este servicio nos devolverá un nuevo resultado. En este caso hemos decidido que, como respuesta, se devuelva el registro insertado así como un nuevo código HTTP de 201 que, como hemos dicho, indica que un recurso nuevo ha sido creado. Podemos ver cómo la respuesta, en este caso, incluye tanto el status code de 201 como el objeto JSON con la información que habíamos enviado.

## 11.2. Usando Excepciones propias

*Esta documentación, creada por Ricardo Cantó Abad, se distribuye bajo los términos de la licencia Creative Commons Atribución-NoComercial-CompartirIgual 3.0 Unported (CC BY-NC-SA 3.0) (http://creativecommons.org/licenses/by-nc-sa/3.0/es/deed.ca).*