

# Departamento de Informática

## Acceso a Datos: Herramientas ORM

Juan Gualberto Gutiérrez Marín

Junio 2023

## Índice

<b>1</b>	<b>Herramientas ORM</b>	<b>3</b>
1.1	Spring Data JPA . . . . .	4
1.2	Spring MVC . . . . .	4
1.3	Para ampliar... . . . .	5
<b>2</b>	<b>Inyección de Dependencias con Spring</b>	<b>7</b>
2.1	Prerrequisitos . . . . .	7
2.2	Puntos clave de Spring . . . . .	8
2.3	Creación del proyecto tipo . . . . .	9
<b>3</b>	<b>Ejemplo de DI</b>	<b>12</b>
3.1	Cómo ejecutar una aplicación Spring Boot desde CLI . . . . .	13
3.2	Añadiendo Starters a Spring Initializr . . . . .	13
3.2.1	Devtools . . . . .	13
3.2.2	Spring JPA . . . . .	13
3.2.3	Mysql Driver . . . . .	13
3.2.4	Lombok . . . . .	13
3.3	Un poco de magia Spring . . . . .	14
3.4	WEBJARS . . . . .	15
<b>4</b>	<b>Análisis y Planificación del proyecto</b>	<b>16</b>
4.1	Análisis del problema . . . . .	16
4.2	Planificación . . . . .	18
4.2.1	Primer Sprint . . . . .	18
4.2.2	Segundo Sprint . . . . .	19
4.2.3	Tercer Sprint . . . . .	19
4.2.4	Cuarto y quinto Sprint . . . . .	19
<b>5</b>	<b>Creación de los contenedores para la práctica</b>	<b>20</b>
5.1	Mysql y Adminer . . . . .	20
5.2	Roundcube/Dovecot/Postfix . . . . .	21
<b>6</b>	<b>Las clases entidad</b>	<b>24</b>
<b>7</b>	<b>Repositorios Spring</b>	<b>27</b>
7.1	Ejemplos de repositorios . . . . .	28
7.1.1	Repositorios con nuestras propias consultas . . . . .	29

<b>8 Controladores</b>	<b>31</b>
8.1 Ejemplo de controlador sencillo . . . . .	31
8.2 Ejemplo de controlador un poco más elaborado . . . . .	33
<b>9 Vistas</b>	<b>37</b>
9.1 Fragmentos . . . . .	38
9.2 Ejemplo de vista para listar . . . . .	41
9.3 Ejemplo sencillo para crear . . . . .	43
9.4 Ejemplo sencillo de modificar . . . . .	45
9.5 Ejemplo algo más completo de listar . . . . .	46
9.6 Maestro-detalle para crear incidencia . . . . .	48
<b>10 Spring Security</b>	<b>52</b>
<b>11 Login Social/OAuth</b>	<b>54</b>

## 1 Herramientas ORM

Una herramienta ORM (Object-Relational Mapping) es una tecnología o biblioteca que permite mapear objetos de una aplicación orientada a objetos a tablas en una base de datos relacional. Proporciona una capa de abstracción entre la base de datos y el código de la aplicación, permitiendo que los desarrolladores trabajen con objetos y clases en lugar de tener que escribir consultas SQL directamente.

El objetivo principal de una herramienta ORM es simplificar y agilizar el desarrollo de aplicaciones al eliminar la necesidad de escribir consultas SQL manualmente y manejar la interacción con la base de datos de manera transparente. Al utilizar una herramienta ORM, los desarrolladores pueden modelar las entidades de su aplicación como clases en lenguajes de programación orientados a objetos como Java, C#, Python, etc., y luego utilizar métodos y consultas específicas del ORM para interactuar con la base de datos.

Algunas de las funciones y características comunes proporcionadas por las herramientas ORM incluyen:

1. **Mapeo objeto-relacional:** Las herramientas ORM mapean automáticamente las propiedades de las clases a las columnas de la base de datos y viceversa. Esto permite que los objetos se almacenen, se recuperen y se actualicen en la base de datos de manera transparente sin necesidad de escribir consultas SQL manualmente.
2. **Generación de consultas SQL:** Las herramientas ORM generan automáticamente las consultas SQL necesarias para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en la base de datos a partir de las operaciones realizadas en las entidades de la aplicación.
3. **Administración de transacciones:** Las herramientas ORM proporcionan mecanismos para administrar transacciones en la base de datos. Esto garantiza la integridad y consistencia de los datos al permitir que las operaciones se realicen en forma atómica (todo o nada) y se deshagan automáticamente si se produce un error.
4. **Caché de objetos:** Las herramientas ORM a menudo incluyen un mecanismo de caché de objetos para mejorar el rendimiento y reducir las consultas a la base de datos. Esto permite almacenar en memoria los objetos recuperados de la base de datos para un acceso más rápido en futuras operaciones.
5. **Consultas avanzadas y optimizaciones:** Las herramientas ORM suelen ofrecer funciones avanzadas para realizar consultas complejas, consultas personalizadas y optimizaciones de rendimiento, como la carga ansiosa (eager loading) y la recuperación diferida (lazy loading) de datos.

Algunas de las herramientas ORM populares son Hibernate (para Java), Entity Framework (para .NET), Django ORM (para Python), Sequelize (para JavaScript/Node.js), entre otras. Estas herramientas ofrecen

una abstracción poderosa y simplificada para trabajar con bases de datos relacionales, mejorando la productividad del desarrollo y facilitando el mantenimiento del código.

## 1.1 Spring Data JPA

Spring en sí mismo no es una herramienta ORM, pero ofrece integraciones y soporte para varias herramientas ORM populares como Hibernate, JPA (Java Persistence API) y MyBatis.

Spring Data JPA es uno de los módulos de Spring que proporciona una capa de abstracción adicional sobre JPA, simplificando aún más el desarrollo de aplicaciones ORM en Java. Spring Data JPA combina la potencia de JPA con las características y funcionalidades adicionales de Spring, como la inyección de dependencias, la administración de transacciones y el manejo de excepciones.

Al utilizar Spring Data JPA, puedes aprovechar las anotaciones de mapeo de entidades, las consultas JPA, la administración de transacciones y otras funcionalidades proporcionadas por JPA para interactuar con la base de datos. Spring Data JPA también ofrece características adicionales, como la generación automática de consultas, consultas personalizadas basadas en convenciones de nomenclatura y soporte para paginación y clasificación de resultados.

Además de Spring Data JPA, Spring Framework en general proporciona soporte para la configuración y administración de transacciones, lo que facilita la integración con diversas herramientas ORM. Spring también ofrece capacidades de caché a través del módulo Spring Cache, que se puede utilizar en combinación con una herramienta ORM para mejorar el rendimiento de las consultas y la recuperación de datos.

## 1.2 Spring MVC

Spring MVC (Model-View-Controller) es un framework de desarrollo web basado en el patrón de diseño MVC. Proporciona una estructura para el desarrollo de aplicaciones web en Java, donde el flujo de ejecución se divide en tres componentes principales: modelo, vista y controlador.

El patrón de diseño MVC separa la lógica de la aplicación en tres componentes distintos:

- **Modelo (Model):** Representa los datos y la lógica de negocio de la aplicación. El modelo encapsula la información y proporciona métodos para acceder, actualizar y manipular los datos. También puede contener la lógica para validar los datos y realizar operaciones relacionadas con la lógica del dominio.
- **Vista (View):** Es la representación visual de los datos del modelo. La vista es responsable de la presentación y el formato de los datos que se muestran al usuario. Puede ser una página HTML, una plantilla, una interfaz de usuario o cualquier otro medio para mostrar la información al cliente.

- **Controlador (Controller):** Actúa como intermediario entre el modelo y la vista. Recibe las solicitudes del cliente, interactúa con el modelo para procesar los datos y determina la vista adecuada para presentar la respuesta al usuario. El controlador maneja las solicitudes HTTP, invoca métodos del modelo y selecciona la vista apropiada para generar la respuesta.

Spring MVC se basa en este patrón y proporciona una implementación flexible y escalable para el desarrollo de aplicaciones web en Java. Algunas características y beneficios de Spring MVC incluyen:

- **Separación clara de responsabilidades:** La arquitectura basada en MVC permite una separación clara de las responsabilidades, lo que facilita el mantenimiento, la reutilización y la prueba de las diferentes capas de la aplicación.
- **Configuración flexible:** Spring MVC se configura mediante anotaciones, archivos XML o Java, lo que brinda flexibilidad en la configuración de las rutas, los controladores, las vistas y otros aspectos de la aplicación.
- **Integración con otros componentes de Spring:** Spring MVC se integra de manera natural con otros componentes del ecosistema de Spring, como Spring Boot, Spring Data, Spring Security, entre otros.
- **Soporte para pruebas unitarias:** Spring MVC proporciona herramientas y APIs para facilitar las pruebas unitarias de los controladores, lo que permite una prueba eficaz de la lógica de la aplicación y la interacción con el modelo y las vistas.

### 1.3 Para ampliar...

Si quieres aprender más sobre el proyecto Spring, o Spring Framework, puedes visitar su Web <https://spring.io/>.

Spring es un proyecto de código abierto que abarca una amplia gama de tecnologías y componentes para el desarrollo de aplicaciones empresariales en Java. Proporciona una plataforma completa y coherente para el desarrollo de aplicaciones, abordando diferentes aspectos como la creación de aplicaciones web, la administración de transacciones, la integración con bases de datos, la seguridad, la programación orientada a aspectos y mucho más.

Por desgracia en este curso no tenemos tiempo de ver completamente Spring, pero estos son algunos de los principales componentes y características:

- **Inversión de control (IoC):** Spring hace uso extensivo del patrón de diseño Inversión de Control (IoC), también conocido como Inyección de Dependencias (DI). Esto permite que los objetos sean creados y administrados por el contenedor de Spring, en lugar de que las clases creen y administren sus propias dependencias. Esto promueve una arquitectura más modular y facilita la prueba unitaria y la reutilización de componentes.

- **Spring MVC:** Es el módulo de Spring para el desarrollo de aplicaciones web basadas en el patrón Modelo-Vista-Controlador (MVC). Proporciona una estructura y conjunto de clases para construir fácilmente aplicaciones web, manejar solicitudes HTTP, administrar formularios, realizar validaciones, manejar sesiones, y mucho más.
- **Persistencia de datos:** Spring ofrece soporte para el acceso y la persistencia de datos mediante diferentes tecnologías y herramientas ORM como Hibernate, JPA, MyBatis y JDBC. Spring Data es un subproyecto de Spring que simplifica aún más el desarrollo de capas de persistencia mediante la generación automática de consultas, la gestión de transacciones y la integración con diversas bases de datos.
- **Seguridad:** Spring Security es otro módulo clave de Spring que proporciona funciones y herramientas para la implementación de la seguridad en aplicaciones web y de servicios. Ofrece autenticación y autorización, protección contra ataques, gestión de sesiones y más.
- **Integración:** Spring facilita la integración con otras tecnologías y sistemas mediante el soporte de numerosos protocolos y estándares, como SOAP, REST, JMS, RMI, entre otros. También ofrece integración con frameworks y bibliotecas populares, como Apache Kafka, RabbitMQ, Apache Solr, entre otros.
- **Programación orientada a aspectos (AOP):** Spring ofrece soporte para la programación orientada a aspectos, lo que permite modularizar aspectos transversales de una aplicación, como la seguridad, la auditoría y el manejo de transacciones, separándolos del código principal y promoviendo una mejor separación de preocupaciones.
- Spring tiene como objetivo proporcionar una plataforma sólida y flexible para el **desarrollo de aplicaciones empresariales** en Java, simplificando tareas comunes, promoviendo las mejores prácticas y fomentando la modularidad y la reutilización de componentes.

## 2 Inyección de Dependencias con Spring

¿Por qué usamos el framework Spring? Spring nos ayuda a automatizar muchas tareas de “calentar el plato” o *boilerplate*.

Spring nos permite desarrollar aplicaciones de manera más rápida, eficaz y corta, saltándonos tareas repetitivas y ahorrándonos líneas de código.

### 2.1 Prerrequisitos

Necesitamos tener instalada una JDK y Maven en el equipo y accesible en la variable PATH del sistema operativo. Dependiendo del sistema operativo, es posible instalarlos desde las diferentes herramientas (ej. apt en Debian, brew en MAC....) sin tener que buscar en Internet.

Vamos a trabajar con VisualStudio Code y necesitamos tener instalados los plugins siguientes:

#### Java Extension Pack

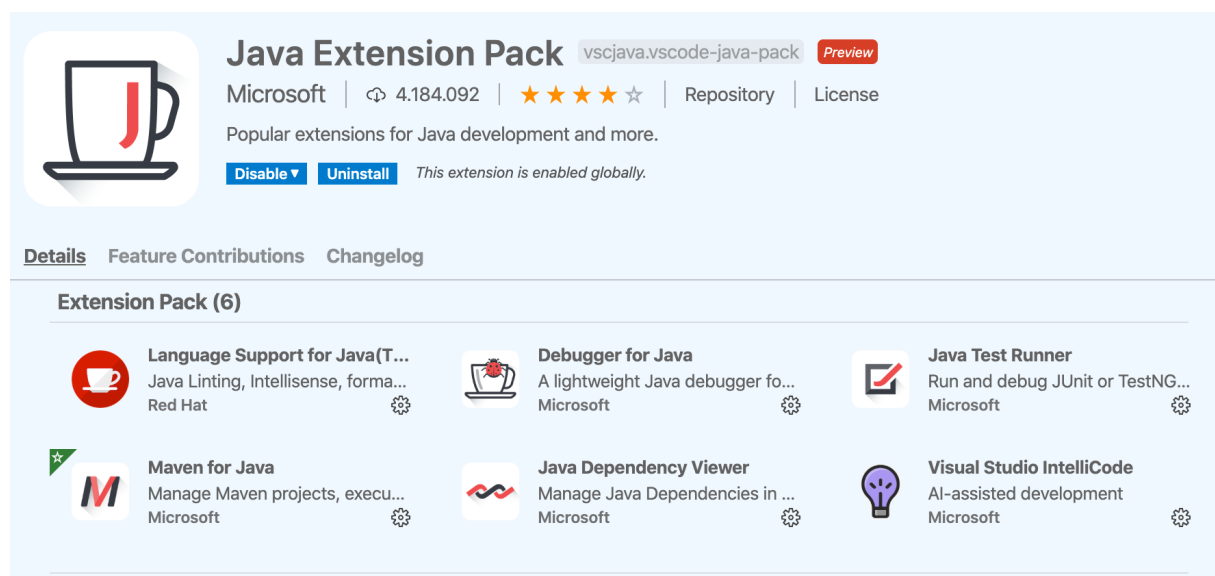
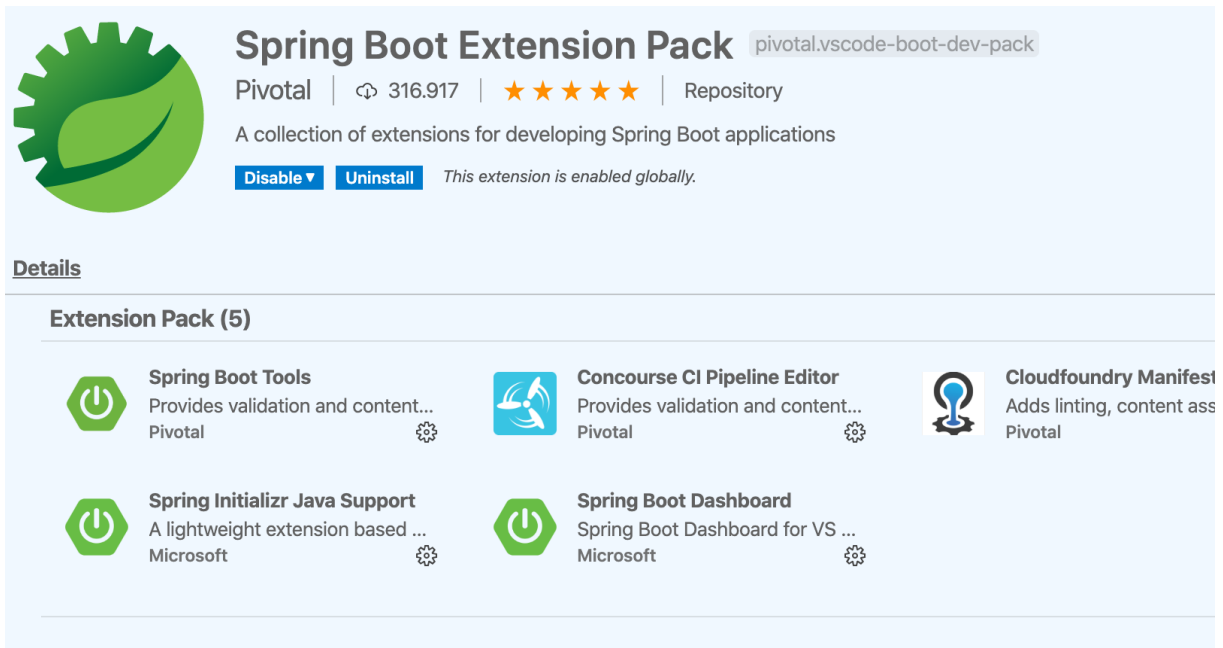


Figura 1: Java Extension Pack

#### Spring boot Extension Pack





The image shows the Visual Studio Code interface for the 'Spring Boot Extension Pack'. At the top left is a green gear icon with a leaf inside. To its right, the title 'Spring Boot Extension Pack' is displayed in a large, bold font, followed by the repository name 'pivotal.vscode-boot-dev-pack' in a smaller font. Below the title, the publisher 'Pivotal' is listed, along with a download count of 316,917 and a five-star rating. A description states it is 'A collection of extensions for developing Spring Boot applications'. There are buttons for 'Disable' and 'Uninstall', and a note that the extension is enabled globally. Below this, a section titled 'Details' contains a sub-section 'Extension Pack (5)' which lists five individual extensions: 'Spring Boot Tools' (Pivotal), 'Concourse CI Pipeline Editor' (Pivotal), 'Cloudfoundry Manifest' (Pivotal), 'Spring Initializr Java Support' (Microsoft), and 'Spring Boot Dashboard' (Microsoft). Each extension entry includes its icon, name, description, publisher, and a settings gear icon.

**Spring Boot Extension Pack** pivotal.vscode-boot-dev-pack

Pivotal | 316.917 | ★★★★★ | Repository

A collection of extensions for developing Spring Boot applications

[Disable](#) [Uninstall](#) This extension is enabled globally.

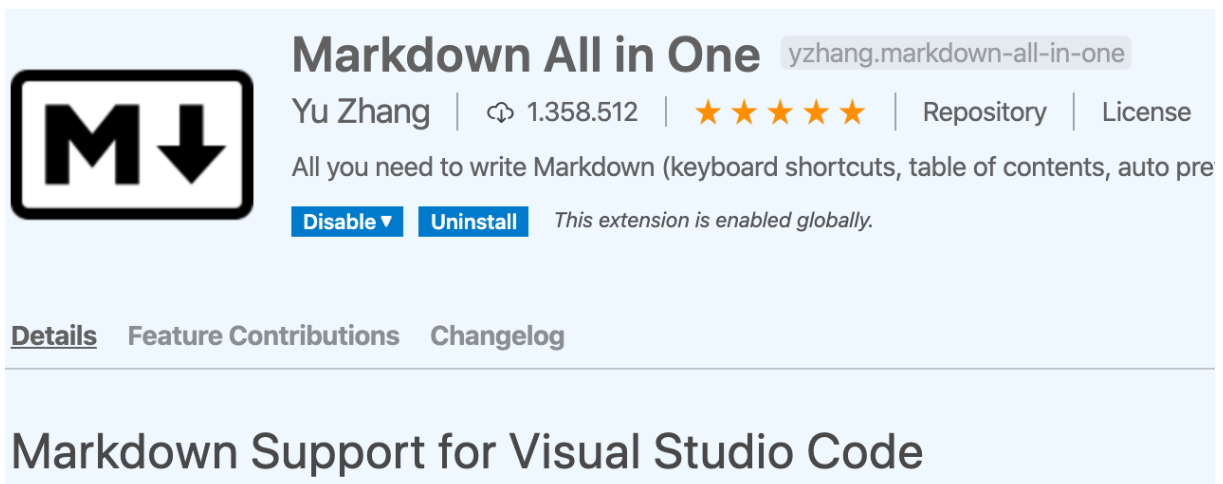
**Details**

**Extension Pack (5)**

- Spring Boot Tools** Provides validation and content... Pivotal
- Concourse CI Pipeline Editor** Provides validation and content... Pivotal
- Cloudfoundry Manifest** Adds linting, content ass Pivotal
- Spring Initializr Java Support** A lightweight extension based ... Microsoft
- Spring Boot Dashboard** Spring Boot Dashboard for VS ... Microsoft

Figura 2: Springboot Extension PACK

## Markdown All in One



The image shows the Visual Studio Code interface for the 'Markdown All in One' extension. On the left is a large icon with a black 'M' and a downward arrow. To the right, the title 'Markdown All in One' is displayed in a large, bold font, followed by the repository name 'yzhang.markdown-all-in-one'. Below the title, the publisher 'Yu Zhang' is listed, along with a download count of 1,358,512 and a five-star rating. A description states 'All you need to write Markdown (keyboard shortcuts, table of contents, auto pre'. There are buttons for 'Disable' and 'Uninstall', and a note that the extension is enabled globally. Below this, a section titled 'Details' contains links for 'Feature Contributions' and 'Changelog'. At the bottom, the text 'Markdown Support for Visual Studio Code' is displayed in a large, bold font.

**Markdown All in One** yzhang.markdown-all-in-one

Yu Zhang | 1.358.512 | ★★★★★ | Repository | License

All you need to write Markdown (keyboard shortcuts, table of contents, auto pre

[Disable](#) [Uninstall](#) This extension is enabled globally.

**Details** [Feature Contributions](#) [Changelog](#)

**Markdown Support for Visual Studio Code**

Figura 3: Markdownallinone

## 2.2 Puntos clave de Spring

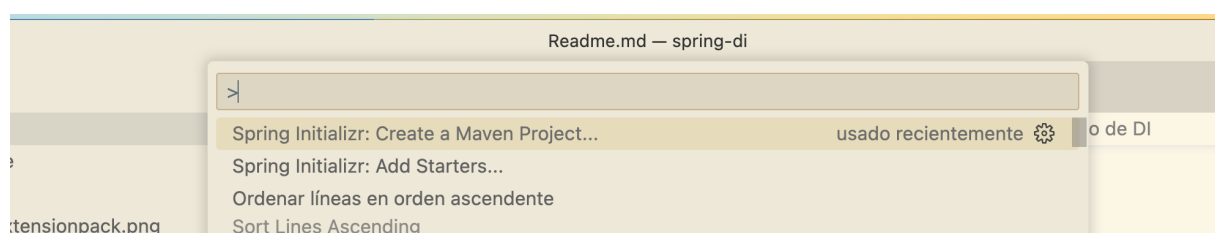
- **Inversion de Control (IoC):** básicamente de lo que se trata es de invertir la forma en que se controla la aplicación, lo qué antes dependía del programador, una secuencia de comandos

desde alguno de nuestros métodos, ahora depende completamente del framework, con la idea de crear aplicaciones más complejas y con funcionamientos más automáticos.

- **Inyección de dependencia (DI):** el manejo de las propiedades de un objeto son inyectadas a través de un constructor, un setter, un servicio, etc.

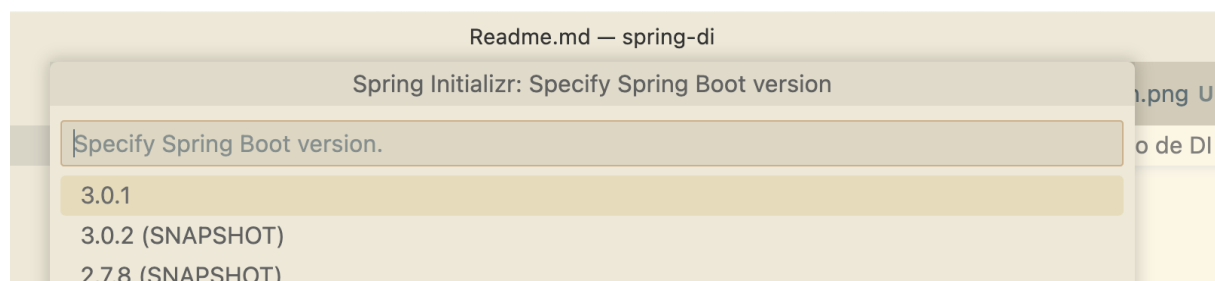
## 2.3 Creación del proyecto tipo

Para la creación del proyecto nos vamos a la paleta de comandos y creamos un proyecto Spring con Maven:



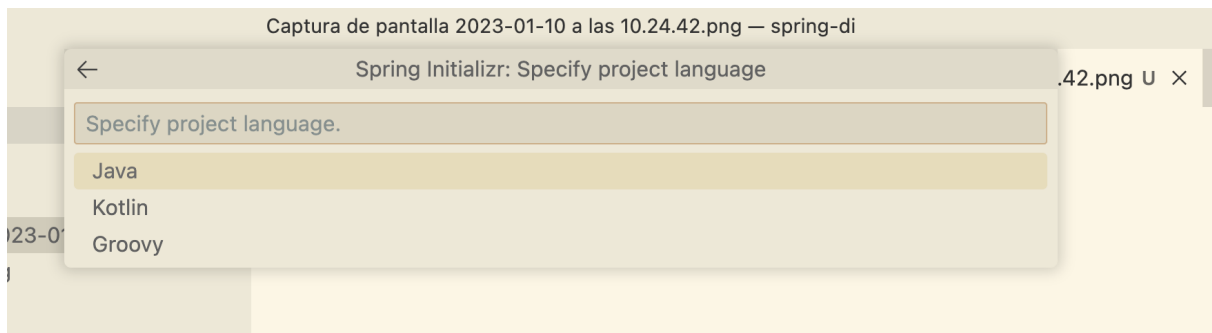
**Figura 4:** Creación de proyecto Spring con Maven

Al pulsar **enter** podemos seleccionar la versión de Spring boot que queremos usar, seleccionamos la última en nuestro caso.



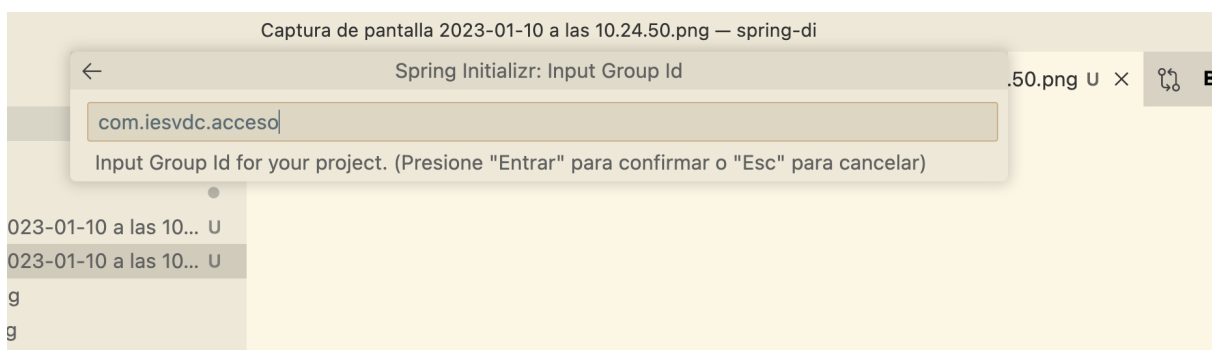
**Figura 5:** Selección Spring Boot

Seguidamente seleccionamos el lenguaje de programación que queremos usar, en nuestro caso nos decantamos por Java:



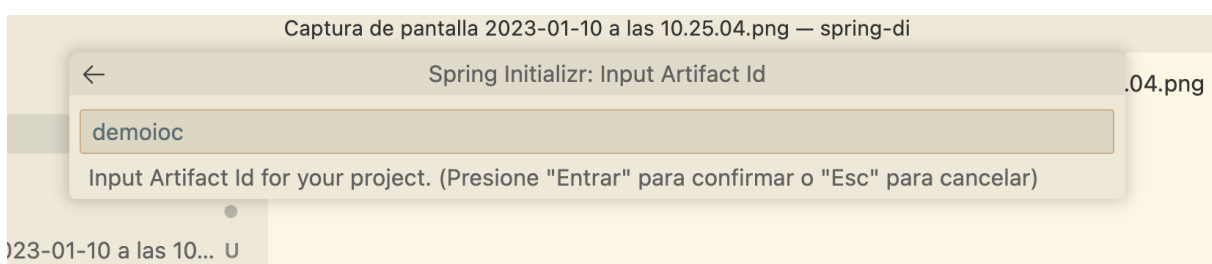
**Figura 6:** Selección del lenguaje de programación

Ya podemos indicar el grupo (paquete) donde va a estar nuestra aplicación:



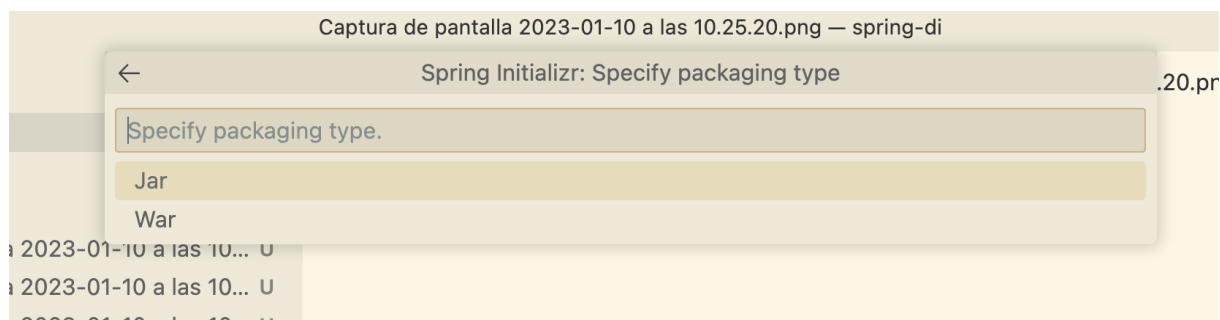
**Figura 7:** Indicamos el paquete

Tras el paquete, hay que introducir el nombre de nuestro artefacto (aplicación):



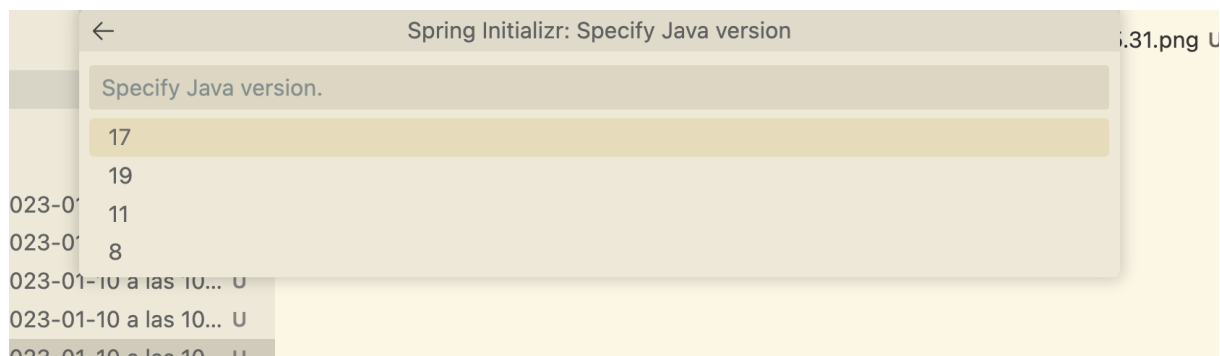
**Figura 8:** Indicamos el artefacto

Luego el tipo de empaquetado, como es una aplicación Spring Boot usaremos **JAR**, pues no necesitamos un servidor de aplicaciones, lleva embebido un Tomcat:



**Figura 9:** Empaquetado

A continuación seleccionamos la versión de Java, donde seleccionaremos 17 por ser la última LTS liberada a día de hoy:



**Figura 10:** Selección de la versión de Java

En este proyecto *tonto* no necesitamos añadir ninguna dependencia a nuestro proyecto Maven, así que simplemente pulsamos enter en la selección de las mismas:



**Figura 11:** Selección de dependencias Maven

### 3 Ejemplo de DI

En la carpeta **resources** Spring busca por recursos para utilizar en el proyecto (por ejemplo, configuración de los DataSources, seguridad, etc.).

En dicha carpeta crearemos un fichero llamado `beans.xml` con la siguiente estructura:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://www.springframework.org/schema/beans
5         http://www.springframework.org/schema/beans/spring-beans.xsd">
6     <!-- aquí van nuestras beans -->
7 </beans>
```

Esta será la estructura genérica que siempre tendrá nuestro archivo de beans. Ahora dentro, ya podemos crear elementos a los que Spring hará su magia y los convertirá en objetos en tiempo de ejecución, por ejemplo, si dentro del archivo creamos el bean:

```
1 <bean id="mi_moto" class="com.iesvdc.acceso.beans.Moto">
2     <property name="marca" value="Harley Davidson"></property>
3     <property name="modelo" value="SportSter 1200"></property>
4 </bean>
```

Desde la aplicación Spring Boot podremos leerlo así:

```
1 public class App
2 {
3     public static void main( String[] args )
4     {
5         ApplicationContext ac =
6             new ClassPathXmlApplicationContext("com/iesvdc/acceso/xml/
7             beans.xml");
8         Moto miMoto = (Moto) ac.getBean("mi_moto");
9         System.out.println(miMoto.toString());
10    }
```

### 3.1 Cómo ejecutar una aplicación Spring Boot desde CLI

Para invocar la aplicación desde Maven bastará con escribir desde la raíz del proyecto la siguiente orden:

```
1 mvn spring-boot:run
```

### 3.2 Añadiendo Starters a Spring Initializr

#### 3.2.1 Devtools

Spring reinicia el programa en ejecución cada vez que hay un cambio en el disco (cuando pulsamos CTRL+S). Cuidado si tienes activado el autoguardado en tu IDE porque puede dar problemas.

#### 3.2.2 Spring JPA

Para las anotaciones de las clases entidad (modelo).

#### 3.2.3 Mysql Driver

Necesario para conectar a MySQL.

#### 3.2.4 Lombok

Un clásico en los proyectos con clases modelo (los llamados POJO o Plain Old Java Objects).

Lombok automatiza la tarea de añadir todos los constructores, getters, setters, etc. a nuestras clases modelo.

### 3.3 Un poco de magia Spring

La Inversión de Control (IoC) es un principio de ingeniería del software por el cual se transfiere el control de objetos o parte de un programa a un contenedor o framework

Spring es un framework que permite inversión de control, es decir, parte de su gran poder reside en hacer tareas por sí misma que de otra manera tendría que implementar el programador.

Por ejemplo, para montar una API REST que haga el marshalling de un objeto a JSON para que sea consumido en un endpoint concreto, con Java tendríamos que usar Jersey, JAXB, un proxy JSON y un servidor de aplicaciones donde correrlo.

Con Spring montar una API de ejemplo se hace en apenas unas líneas de código. Veamos esta versión extendida del anterior programa:

```
1  import org.springframework.boot.SpringApplication;
2  import org.springframework.boot.autoconfigure.SpringBootApplication;
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.support.
    ClassPathXmlApplicationContext;
5  import org.springframework.web.bind.annotation.RequestMapping;
6  import org.springframework.web.bind.annotation.GetMapping;
7  import org.springframework.web.bind.annotation.RestController;
8  // clase base, el POJO
9  import com.iesvdc.acceso.beans.Moto;
10
11  @RestController
12  @RequestMapping("api")
13  @SpringBootApplication
14  public class App {
15      public static void main(String[] args) {
16          SpringApplication.run(App.class, args);
17      }
18
19
20      @GetMapping(value = "moto")
21      public Moto findMoto() {
22          ApplicationContext ac = new ClassPathXmlApplicationContext("com
23                                  /iesvdc/acceso/xml/beans.xml");
24          Moto miMoto = (Moto) ac.getBean("mi_moto");
25          return miMoto;
26      }
27  }
```

En una terminal podemos escribir:

```
mvn spring-boot:run
```

Mientras que en otra se puede comprobar con:

```
curl http://localhost:8080/api/moto
```

Acabamos de ver cómo sólo con añadir la anotación **@RestController** nuestra aplicación ya contesta a peticiones Web y cómo hemos hecho el marshalling/unmarshalling del objeto moto (lo hemos leído de un XML y lo mandamos en JSON), todo en tres líneas de código.

### 3.4 WEBJARS

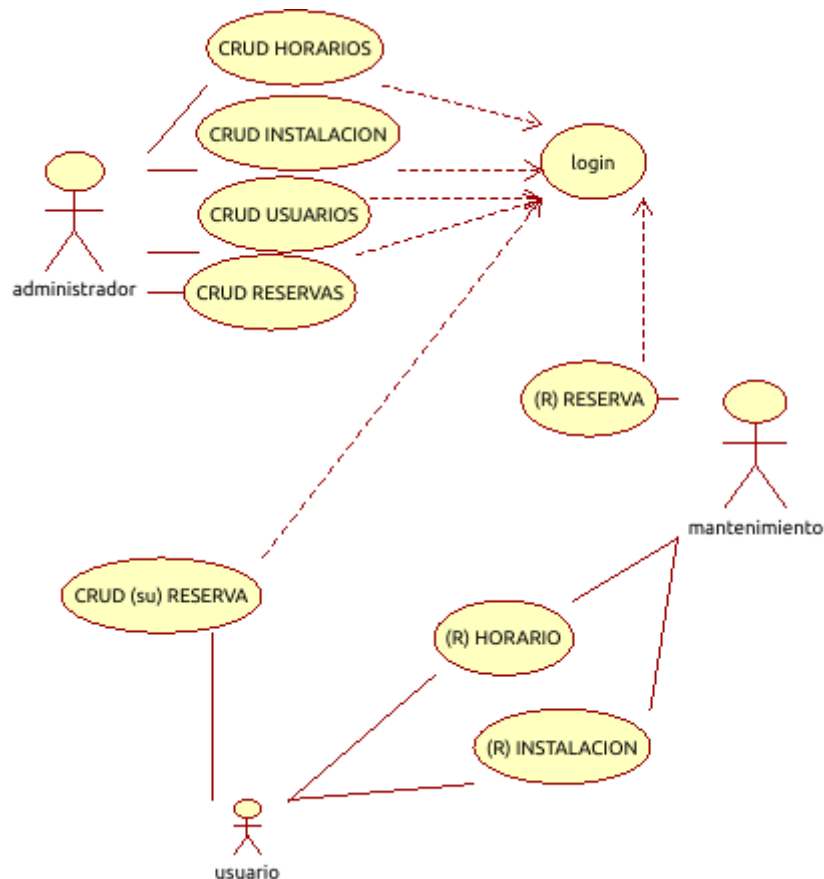
Para ver cómo importar, la ruta que tenemos que poner en nuestro HTML de los WebJars, ponemos: <https://www.webjars.org/all>.



## 4 Análisis y Planificación del proyecto

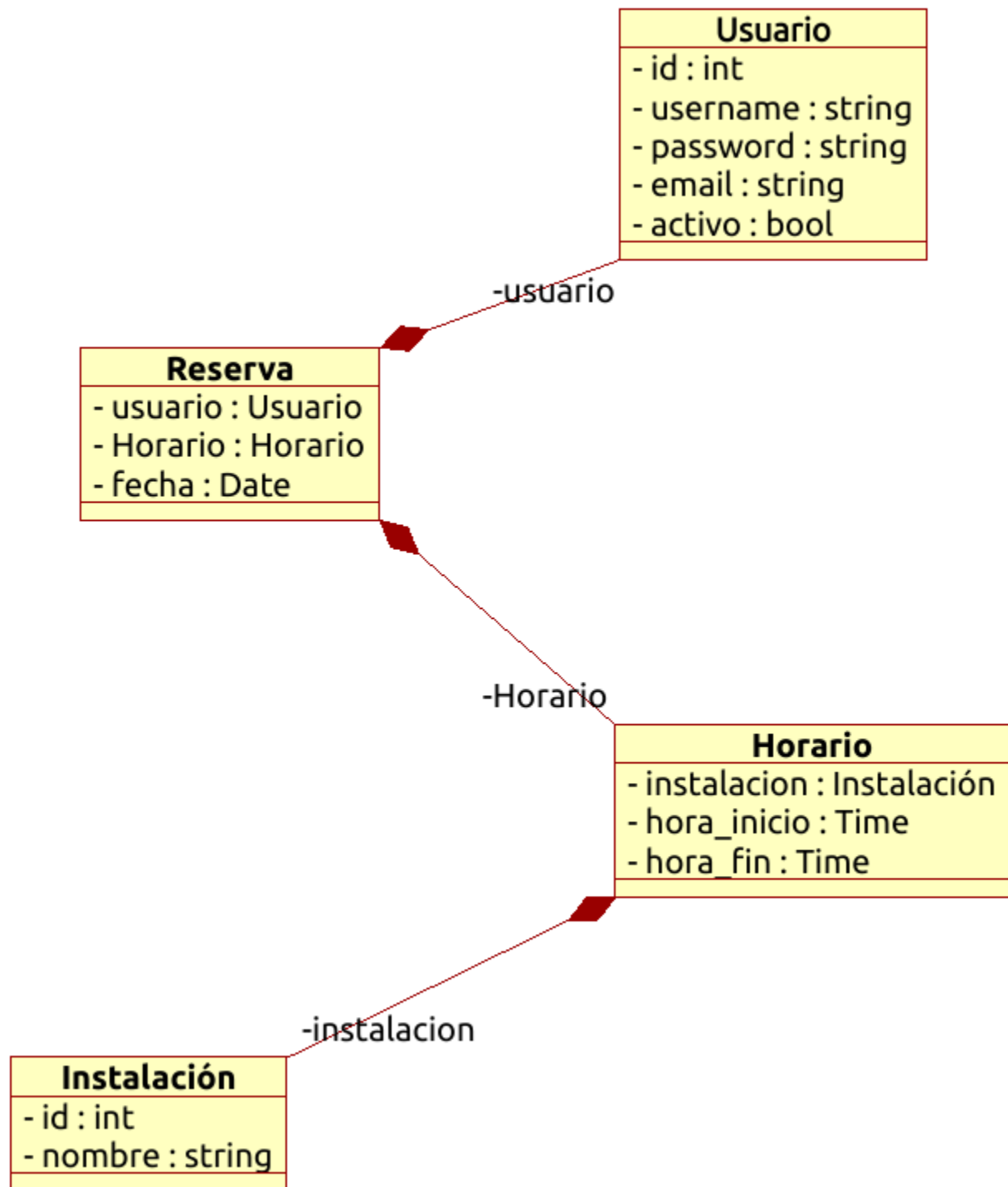
### 4.1 Análisis del problema

Primero hacemos un diagrama de alto nivel de casos de uso del sistema (en este caso cuidado que es de otro ejemplo):

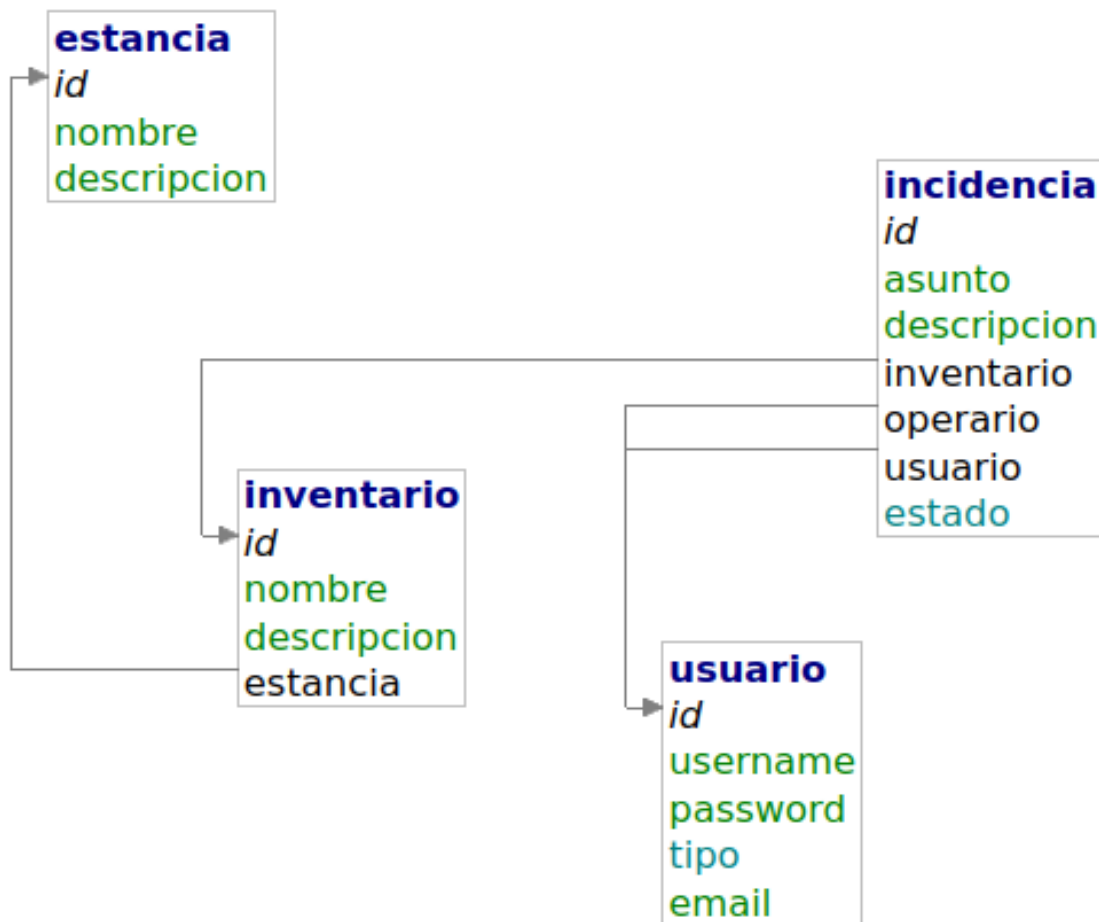


**Figura 12:** diagrama de casos de uso

Posteriormente, a raíz de este diagrama de casos de uso, podemos crear nuestro primer diagrama de clases (repetimos, ten ).

**Figura 13:** diagrama de clases

Estas clases, la herramienta ORM, en nuestro caso Spring, darán lugar de manera *automática* a las tablas:



**Figura 14:** diagrama E-R

## 4.2 Planificación

### 4.2.1 Primer Sprint

- Análisis:
  - Diseño de clases
  - Diagrama de caso de uso
- Creación de los modelos
- Creación de los repositorios

- Creación de los servicios

#### **4.2.2 Segundo Sprint**

- Análisis:
  - Diagrama de casos de uso
- Creación Web con Thymeleaf de listado de instalaciones
- Creación Web con Thymeleaf de listado de horarios

#### **4.2.3 Tercer Spring**

- Análisis:
  - Diagrama de casos de uso
  - Listado de roles y rutas
- Spring Security con sesiones

#### **4.2.4 Cuarto y quinto Spring**

- Nuestra IA predice que el product owner nos cambiará en dos meses todo a REST con autenticación JWT y login social (GitHub) + registro por email.
- Hay que refactorizar y preparar un frontend en React.

## 5 Creación de los contenedores para la práctica

Docker es una plataforma de contenedores que permite empaquetar y distribuir aplicaciones junto con sus dependencias en contenedores ligeros y portátiles. Estos contenedores son unidades de software autónomas que encapsulan todo lo necesario para ejecutar una aplicación, incluyendo el código, las bibliotecas, las herramientas y las configuraciones. Docker proporciona una forma fácil y eficiente de crear, distribuir y ejecutar aplicaciones en diferentes entornos.

Recuerda que debemos poner cada servicio en un contenedor diferente en Docker se basa en los principios de modularidad, escalabilidad y aislamiento de aplicaciones, además de facilitar su gestión:

- **Modularidad:** Al colocar cada servicio en un contenedor separado, se puede seguir el principio de diseño de software de “separación de intereses”. Cada contenedor contiene un componente o servicio específico de la aplicación, lo que facilita la gestión y el mantenimiento del sistema en general. Además, al utilizar contenedores independientes, es posible actualizar o cambiar un servicio sin afectar a otros componentes de la aplicación, lo que brinda flexibilidad y facilita la evolución de la aplicación con el tiempo.
- **Escalabilidad:** Al dividir la aplicación en servicios individuales en contenedores separados, es posible escalar vertical u horizontalmente los recursos de manera independiente según las necesidades de cada servicio. Esto permite asignar más recursos a los servicios que requieren mayor capacidad y optimizar el rendimiento general de la aplicación.
- **Aislamiento:** Docker proporciona aislamiento entre los contenedores, lo que significa que cada contenedor tiene su propio entorno aislado. Esto garantiza que los servicios no interfieran entre sí, evitando posibles conflictos o dependencias entre ellos. Además, si un contenedor falla, los demás continúan funcionando sin problemas, lo que mejora la tolerancia a fallos y la disponibilidad del sistema.
- **Facilidad de gestión:** Al tener servicios individuales en contenedores separados, la gestión de cada servicio se simplifica. Cada contenedor se puede configurar, monitorear y escalar de forma independiente. Además, es más sencillo realizar pruebas y depurar problemas en un servicio específico sin afectar al resto de la aplicación.

### 5.1 Mysql y Adminer

A partir de documentación de la imagen oficial [https://hub.docker.com/\\_/mysql](https://hub.docker.com/_/mysql), creamos un fichero docker-compose para poder levantar estos dos servicios:

- **MySQL:** El servicio de la archiconocida base de datos relacional (a día de hoy la segunda en el ranking <https://db-engines.com/en/ranking>). Por defecto el servidor está en el puerto 3306, luego nuestros programas deberán conectarse a ese puerto para *hablar* con la base de datos.

- **Adminer:** Un servicio Web escrito en PHP para conectarse y gestionar bases de datos relacionales. Aunque realmente no es necesario, nos resultará muy útil a la hora de interactuar con la base de datos vía Web sin necesidad de usar software adicional o instalar MySQL Workbench. Por defecto está en el puerto 8080, luego si abrimos nuestro navegador y escribimos `http://HOST:8080` donde HOST es el nombre del equipo donde está instalado, podemos acceder vía Web a este entorno que a su vez se conecta a la base de datos (recuerda que MySQL estaba en el puerto 3306).

Como es posible que tengamos alguno de esos puertos estándar ya en uso, nosotros vamos a *natear* a otros puertos en el *docker-compose*. ¿Qué es eso de *natear*? Pues hacer NAT, recuerda que los contenedores están inicialmente aislados dentro de tu equipo, en una red virtual interna, luego para exponerlos a nuestra red local LAN, podemos “conectarlos” a un puerto de nuestra máquina física. Así por ejemplo vamos a exponer el puerto 3306 del MySQL en el puerto 33306 de nuestra máquina real. Lo mismo con el puerto 8080 del Adminer en el puerto 8181 de la máquina real.

Vamos a crear también una red virtual llamada *skynet* para que todos los contenedores *hablen* entre sí.

```
1 version: '3.1'
2 services:
3   db:
4     image: mysql
5     command: --default-authentication-plugin=mysql_native_password
6     restart: "no"
7     environment:
8       MYSQL_ROOT_PASSWORD: zx76wbz7FG89k
9     networks:
10      - skynet
11     ports:
12      - 33306:3306
13
14   adminer:
15     image: adminer
16     restart: "no"
17     networks:
18      - skynet
19     ports:
20      - 8181:8080
21 networks:
22   skynet:
```

## 5.2 Roundcube/Dovecot/Postfix

Para nuestras pruebas de registro de nuevos usuarios vamos a usar este cómodo contenedor <https://registry.hub.docker.com/r/maroooo/postfix-roundcube> para no sacar de la red interna de

los contenedores correos electrónicos que puedan disparar todas las alarmas del departamento de Ciberseguridad de la empresa.

Si bien este contenedor en concreto no cumple con los requisitos anteriores y está algo obsoleto, como su **propósito** es exclusivamente para **testing** no queremos perder más tiempo ni recursos en hacerlo correctamente pues en producción estará a cargo del departamento de sistemas o IT y no del departamento de desarrollo.

En la documentación oficial vemos que podemos correr el contenedor así:

```
1 mkdir -p ./data
2 docker run -e "ADMIN_USERNAME=root"
3           -e "ADMIN_PASSWD=password"
4           -e "DOMAIN_NAME=example.com"
5           -e "USERS=user1:pass1,user2:pass2"
6           -d -v /data/mysql:/var/lib/mysql -v /data/vmail:/var/vmail
7           -v /data/log:/var/log
8           -p 25:25 -p 80:80 -p 110:110 -p 143:143 -p 465:465 -p
           993:993 -p 995:995
           marooou/postfix-roundcube
```

De ahí sacamos los puertos de necesitamos exponer si queremos acceder al contenedor. Dicho esto, modificamos el docker-compose anterior así:

```
1 version: '3.1'
2 services:
3   dbincidentes:
4     image: mysql
5     command: --default-authentication-plugin=mysql_native_password
6     restart: "no"
7     environment:
8       MYSQL_ROOT_PASSWORD: zx76wbz7FG89k
9     networks:
10      - skynet
11     ports:
12      - 33306:3306
13
14   adminerincidentes:
15     image: adminer
16     restart: "no"
17     networks:
18      - skynet
19     ports:
20      - 8181:8080
21
22   correoincidentes:
23     image: marooou/postfix-roundcube
24     environment:
25       ADMIN_USERNAME: root
26       ADMIN_PASSWD: zx76wbz7FG89k
```

```
27     DOMAIN_NAME: iesvdc.lan
28     USERS:
29         - user1:pass1
30         - user2:pass2
31     ports:
32         - 25:25
33         - 8282:80
34         - 110:110
35         - 143:143
36         - 465:465
37         - 993:993
38         - 995:995
39     networks:
40         - skynet
41
42 networks:
43     skynet:
```



## 6 Las clases entidad

En Spring Java llamamos POJO o Plain Old Java Objects a aquellas clases Java sencillas que no necesitan heredar o implementar interfaces del framework Spring. En este caso también lo haremos para denominar a las clases entidad de nuestra aplicación.

Veamos la clase usuario:

1. La anotación `@Entity` indica que esta clase es una entidad persistente y se mapeará a una tabla en la base de datos.
2. La anotación `@Data` es una anotación de Lombok que genera automáticamente los métodos `equals()`, `hashCode()`, `toString()`, getters y setters para todos los campos de la clase.
3. La anotación `@NoArgsConstructor` es otra anotación de Lombok que genera un constructor sin argumentos para la clase.
4. La anotación `@Id` indica que el campo `id` es el identificador único de la entidad.
5. La anotación `@GeneratedValue` **especifica la estrategia de generación de valores para el campo `id`. En este caso, se utiliza la estrategia `GenerationType.IDENTITY`**, que indica que el valor del campo se generará automáticamente por la base de datos.
6. Los campos `username`, `password`, `tipo` y `email` representan las propiedades de un usuario.

```
1 import jakarta.persistence.Entity;
2 import jakarta.persistence.GeneratedValue;
3 import jakarta.persistence.GenerationType;
4 import jakarta.persistence.Id;
5 import lombok.Data;
6 import lombok.NoArgsConstructor;
7
8 @Entity
9 @Data
10 @NoArgsConstructor
11 public class Usuario {
12     @Id
13     @GeneratedValue(strategy = GenerationType.IDENTITY)
14     Integer id;
15     String username;
16     String password;
17     String tipo;
18     String email;
19 }
```

Esta clase `Usuario` es una entidad persistente que representa a un usuario en el sistema. Contiene campos como `username`, `password`, `tipo` y `email`, donde `tipo` es una relación con la entidad `Rol` (en este caso un *Enum*). Esta clase se mapeará a una tabla en la base de datos y se utilizará para almacenar y recuperar información de usuarios.

Lo mismo hacemos con la clase estancia:

```
1 @Entity
2 @Data
3 @NoArgsConstructor
4 public class Estancia {
5     @Id
6     @GeneratedValue(strategy = GenerationType.IDENTITY)
7     Integer id;
8     String nombre;
9     int planta;
10    String descripcion;
11 }
```

Es importante que te fijes en las anotaciones necesarias para que tanto JPA como Spring hagan introspección y puedan crear las tablas necesarias en la base de datos, así por ejemplo tenemos las anotaciones:

- **@Entity**: Define una clase entidad, es decir, será también una tabla en la base de datos.
- **@Data**: Es una anotación de Lombok que me genera *automáticamente* todos los *getters*, *setters*, *toString*, etc..
- **@NoArgsConstructor**: En Lombok, define el constructor vacío, necesario para hacer el marshalling/unmarshalling de objetos, sin él no se puede hacer. Es necesario para la biblioteca JAXB.
- **@Id**: Define cual será la llave primaria en la tabla.
- **@GeneratedValue(strategy = GenerationType.IDENTITY)**: Lo usamos para indicar que se trata de un *autoincrement* en MySQL.

Ahora en inventario incluimos una anotación nueva:

- **@ManyToOne**: Una estancia contiene muchos ítems de inventario.

Su anotación antagónica sería **@OneToMany** pero ojo, su significado es muy diferente porque lo tendíamos que usar en una lista en *estancia*, por ejemplo:

- **@OneToMany** List listaInventario;

Que sería la lista de ítems que hay en inventario para una estancia determinada.

```
1 @Entity
2 @Data
3 @NoArgsConstructor
4 public class Inventario {
5     @Id
6     @GeneratedValue(strategy = GenerationType.IDENTITY)
7     Integer id;
8     String nombre;
9     String descripcion;
```

```
10     @ManyToOne
11     Estancia estancia;
12 }
```

Igualmente con *incidencia*:

```
1  @Entity
2  @Data
3  @NoArgsConstructor
4  public class Incidencia {
5      @Id
6      @GeneratedValue(strategy = GenerationType.IDENTITY)
7      Integer id;
8      String asunto;
9      String descripcion;
10     @ManyToOne
11     Usuario usuario;
12     @ManyToOne
13     Usuario operario;
14     String estado;
15 }
```

Los campos usuario y operario están anotados con `@**ManyToOne**`, lo que indica que son relaciones muchos a uno con la entidad Usuario. Esto significa que una incidencia tiene un usuario y un operario asociados.

## 7 Repositorios Spring

Los repositorios son una abstracción que se utiliza para acceder y manipular datos en una base de datos. Los repositorios facilitan la implementación del patrón de diseño de Repositorio en una aplicación.

En Spring, los repositorios se definen como interfaces que extienden de una de las interfaces proporcionadas por el módulo Spring Data, como `JpaRepository`, `CrudRepository` o `PagingAndSortingRepository`. Estas interfaces proporcionan métodos predefinidos para realizar operaciones comunes de persistencia, como guardar, eliminar, buscar y filtrar registros en la base de datos. Básicamente proporcionan una capa de abstracción para acceder a los datos de una base de datos de manera sencilla y eficiente, evitando la necesidad de escribir código repetitivo y consultas SQL complejas. Esto mejora la productividad del desarrollador y facilita el mantenimiento de la capa de persistencia en una aplicación Spring.

Los repositorios de Spring funcionan de la siguiente manera:

1. **Definición de la interfaz del repositorio:** Se crea una interfaz que extiende de una de las interfaces de repositorio proporcionadas por Spring Data. Esta interfaz define métodos para realizar operaciones CRUD y consultas personalizadas.
2. **Anotaciones y configuración:** Se utilizan anotaciones de Spring, como `@Repository`, para marcar la interfaz del repositorio y permitir que Spring la detecte y cree una implementación en tiempo de ejecución. También se configura la conexión a la base de datos y otras propiedades relacionadas en el archivo de configuración de Spring.
3. **Inyección de dependencias:** Los repositorios se inyectan en otras capas de la aplicación, como servicios o controladores, mediante la anotación `@Autowired` o mediante la inyección de dependencias de Spring.
4. **Uso de los métodos del repositorio:** En otras capas de la aplicación, se utilizan los métodos definidos en la interfaz del repositorio para realizar operaciones de persistencia. Estos métodos abstraen las consultas y operaciones CRUD, lo que simplifica la interacción con la base de datos y evita la necesidad de escribir consultas SQL complejas manualmente.
5. **Personalización y consultas personalizadas:** Los repositorios de Spring Data permiten personalizar las consultas mediante la definición de métodos con nombres específicos, utilizando convenciones de nomenclatura. También se pueden definir consultas personalizadas utilizando anotaciones como `@Query`, que permite escribir consultas en lenguajes como JPQL, SQL o MongoDB Query Language.

Para que los repositorios funcionen, hemos de explicar a Spring dónde y cómo almacenar la información. En la carpeta **main/resources** tenemos un archivo de configuración llamado **application.properties** que se utiliza en un proyecto de Spring para configurar propiedades relacionadas con la base de datos y otras configuraciones. Aunque también es posible en vez de tenerlo como archivo

de propiedades, tenerlo como archivo en formato YAML, nosotros usaremos el formato nativo de propiedades Java:

```
1 spring.datasource.url=jdbc:mysql://localhost:33306/gestion_inventario
2 spring.datasource.username=root
3 spring.datasource.password=zx76wbz7FG89k
4 spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.
   MySQL8Dialect
5 spring.jpa.hibernate.ddl-auto=update
6 spring.session.store-type=jdbc
```

En este archivo estamos indicando:

- `spring.datasource.url`: Es la URL de conexión a la base de datos MySQL. En este caso, se está conectando a una base de datos llamada “gestion\_inventario” en el localhost en el puerto 33306.
- `spring.datasource.username`: Es el nombre de usuario utilizado para autenticarse en la base de datos MySQL. En este caso, se utiliza el nombre de usuario “root”.
- `spring.datasource.password`: Es la contraseña utilizada para autenticarse en la base de datos MySQL. En este caso, se utiliza la contraseña “zx76wbz7FG89k”.
- `spring.jpa.properties.hibernate.dialect`: Especifica el dialecto de Hibernate a utilizar. En este caso, se utiliza el dialecto “org.hibernate.dialect.MySQL8Dialect” para trabajar con MySQL 8.
- `spring.jpa.hibernate.ddl-auto`: Especifica cómo Hibernate manejará la creación y actualización de la estructura de la base de datos. En este caso, se configura en “update”, lo que significa que Hibernate actualizará automáticamente la estructura de la base de datos según las entidades definidas en el proyecto.
- `spring.session.store-type`: Especifica el tipo de almacenamiento de sesiones que se utilizará en la aplicación. En este caso, se configura en “jdbc” para almacenar las sesiones en la base de datos a través de JDBC.

En resumen, este archivo de configuración se utiliza para especificar la configuración de la base de datos, la configuración de Hibernate y otras configuraciones relacionadas con el proyecto de Spring. Estas propiedades se cargan automáticamente durante la ejecución del proyecto y se utilizan para establecer la conexión con la base de datos, configurar Hibernate y otros componentes del proyecto.

## 7.1 Ejemplos de repositorios

Como explicamos antes, un repositorio siempre extiende de la interfaz **JpaRepository** proporcionada por Spring Data JPA. A través de esta interfaz, el repositorio hereda una serie de métodos predefinidos

para realizar operaciones de persistencia en la entidad **Estancia** en este ejemplo.

```
1 package com.iesvdc.acceso.tema03.inventario.repos;  
2  
3 import org.springframework.data.jpa.repository.JpaRepository;  
4  
5 import com.iesvdc.acceso.tema03.inventario.model.Estancia;  
6  
7 public interface RepoEstancia extends JpaRepository<Estancia, Integer>  
8     {  
9     }
```

La interfaz está parametrizada con dos tipos: Estancia y Integer. El primer tipo (Estancia) indica la entidad con la que se va a trabajar, y el segundo tipo (Integer) especifica el tipo de datos del identificador de la entidad.

Al extender de JpaRepository, el repositorio hereda métodos como save, delete, findById, findAll, entre otros. Estos métodos permiten realizar operaciones de persistencia básicas, como guardar, eliminar, buscar y recuperar registros de la base de datos.

No es necesario proporcionar una implementación para el repositorio, ya que Spring Data JPA se encarga de crear una implementación en tiempo de ejecución basada en las convenciones de nombres y la configuración de la aplicación.

La anotación @Repository no es necesaria en este caso, ya que JpaRepository ya la incluye y permite que Spring detecte automáticamente el repositorio.

En resumen, este repositorio proporciona una interfaz para interactuar con la entidad Estancia y realizar operaciones de persistencia en la base de datos de manera sencilla y eficiente, gracias a los métodos heredados de JpaRepository.

### 7.1.1 Repositorios con nuestras propias consultas

Vamos a ver el repositorio **RepoInventario**, que extiende de la interfaz JpaRepository proporcionada por Spring Data JPA. A través de esta interfaz, el repositorio hereda una serie de métodos predefinidos para realizar operaciones de persistencia en la entidad Inventario.

```
1 package com.iesvdc.acceso.tema03.inventario.repos;  
2  
3 import java.util.List;  
4  
5 import org.springframework.data.jpa.repository.JpaRepository;  
6 import org.springframework.data.jpa.repository.Query;  
7  
8 import com.iesvdc.acceso.tema03.inventario.model.Estancia;
```

```
9 import com.iesvdc.acceso.tema03.inventario.model.Inventario;
10
11 public interface RepoInventario extends JpaRepository<Inventario,
12     Integer> {
13     @Query("SELECT i FROM Inventario i WHERE i.estancia = ?1")
14     List<Inventario> findByEstancia(Estancia Estancia);
15 }
```

Recuerda que la interfaz `RepoInventario` está parametrizada con dos tipos: `Inventario` y `Integer`. El primer tipo (`Inventario`) indica la entidad con la que se va a trabajar, y el segundo tipo (`Integer`) especifica el tipo de datos del identificador de la entidad.

Al extender de `JpaRepository`, el repositorio hereda métodos como `save`, `delete`, `findById`, `findAll`, entre otros. Estos métodos permiten realizar operaciones de persistencia básicas, como guardar, eliminar, buscar y recuperar registros de la base de datos.

En el repositorio, se ha definido un método personalizado llamado `findByEstancia`, que utiliza la anotación `@Query` de Spring Data JPA. Esta anotación permite definir consultas personalizadas utilizando el lenguaje JPQL (Java Persistence Query Language). En este caso, la consulta busca todos los objetos `Inventario` que estén asociados a una instancia de `Estancia` específica. El parámetro `?1` se refiere al primer parámetro del método, que es la instancia de `Estancia` que se utilizará como filtro.

La anotación `@Query` se utiliza cuando se requiere una consulta personalizada que no se puede lograr utilizando los métodos heredados de `JpaRepository`. Permite escribir consultas más complejas y específicas para acceder a los datos de la base de datos.

## 8 Controladores

Los controladores son componentes que se utilizan para manejar las solicitudes HTTP y generar las respuestas correspondientes. Actúan como intermediarios entre el cliente y el servidor, procesando las solicitudes entrantes y produciendo las respuestas adecuadas.

En el contexto de Spring MVC (Model-View-Controller), los controladores reciben las solicitudes HTTP, extraen los datos necesarios (consultan bases de datos, colecciones de documentos...), invocan la lógica de negocio apropiada y devuelven una respuesta al cliente. Se definen como clases anotadas con la anotación `@Controller` o `@RestController`, que les permite ser reconocidos y administrados por el contenedor de Spring.

Funcionalidades clave de los controladores en Spring:

1. **Gestión de rutas:** Los controladores definen métodos que se asocian a rutas o URLs específicas. Esto se logra mediante la anotación `@RequestMapping` en Spring MVC o mediante anotaciones más específicas como `@GetMapping`, `@PostMapping`, etc.
2. **Recepción de parámetros:** Los métodos de los controladores pueden recibir parámetros enviados en la solicitud HTTP, como parámetros de consulta, encabezados, datos de formulario o cuerpo de la solicitud. Estos parámetros se pueden vincular directamente a los parámetros del método del controlador utilizando anotaciones como `@RequestParam`, `@PathVariable`, `@RequestHeader`, etc.
3. **Lógica de negocio:** Los controladores son responsables de invocar la lógica de negocio adecuada para procesar la solicitud. Esto puede implicar la interacción con servicios, repositorios u otros componentes de la aplicación para realizar operaciones, procesar datos y preparar la respuesta.
4. **Generación de respuestas:** Los controladores devuelven objetos que representan la respuesta a enviar al cliente. Estos objetos pueden ser cadenas de texto, objetos **JSON**, **vistas** (templates) a renderizar, archivos, redirecciones, entre otros. La selección del tipo de respuesta se basa en la anotación del método del controlador y la configuración de Spring.
5. **Manejo de excepciones:** Los controladores también pueden manejar excepciones que se produzcan durante el procesamiento de la solicitud. Esto permite capturar errores, realizar acciones específicas (como devolver un código de estado HTTP adecuado o mostrar una página de error personalizada) y mantener un flujo controlado de la aplicación.

### 8.1 Ejemplo de controlador sencillo

Este es un ejemplo de un controlador en Spring que maneja las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) para la entidad “Estancia”, ten cuidado que lo que ves en los **return** no es un simple texto, sino el nombre del archivo de plantilla que buscará Spring para renderizar la vista (sin el “.html”



del final):

```
1 package com.iesvdc.acceso.tema03.inventario.controller;
2
3 import com.iesvdc.acceso.tema03.inventario.repos.RepoEstancia;
4 import com.iesvdc.acceso.tema03.inventario.model.Estancia;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Controller;
7 import org.springframework.ui.Model;
8 import org.springframework.web.bind.annotation.*;
9
10 @Controller
11 @RequestMapping("/estancias")
12 public class EstanciaController {
13
14     @Autowired
15     RepoEstancia repoEstancia;
16
17     @GetMapping("")
18     public String listarEstancias(Model model) {
19         model.addAttribute("estancias", repoEstancia.findAll());
20         return "estancias";
21     }
22
23     @GetMapping("/create")
24     public String mostrarFormularioCrear(Model model) {
25         model.addAttribute("estancia", new Estancia());
26         return "estanciacreate";
27     }
28
29     @PostMapping("/guardar")
30     public String guardarEstancia(@ModelAttribute("estancia") Estancia
31         estancia) {
32         repoEstancia.save(estancia);
33         return "redirect:/estancias";
34     }
35
36     @GetMapping("/editar/{id}")
37     public String mostrarFormularioEditar(@PathVariable("id") Integer
38         id, Model model) {
39         Estancia estancia = repoEstancia.findById(id).get();
40         model.addAttribute("estancia", estancia);
41         return "estanciaedit";
42     }
43
44     @GetMapping("/eliminar/{id}")
45     public String eliminarEstancia(@PathVariable("id") Integer id) {
46         repoEstancia.deleteById(id);
47         return "redirect:/estancias";
48     }
49 }
```

- La anotación `@Controller` indica que esta clase es un controlador en Spring.
- La anotación `@RequestMapping("/estancias")` especifica que las rutas URL relacionadas con este controlador deben comenzar con `"/estancias"`.
- La anotación `@Autowired` se utiliza para inyectar una instancia del repositorio `"RepoEstancia"` en el controlador.
- El método `listarEstancias()` maneja las solicitudes GET a `"/estancias"` y devuelve una vista llamada `"estancias"`. Agrega todos los objetos `"Estancia"` recuperados del repositorio al modelo, que estarán disponibles en la vista.
- El método `mostrarFormularioCrear()` maneja las solicitudes GET a `"/estancias/create"` y devuelve una vista llamada `"estanciacreate"`. Agrega un nuevo objeto `"Estancia"` vacío al modelo para su uso en el formulario de creación.
- El método `guardarEstancia()` maneja las solicitudes POST a `"/estancias/guardar"`. Toma el objeto `"Estancia"` enviado en el formulario y lo guarda en el repositorio utilizando el método `save()`. Luego redirige a la página principal de `"estancias"`.
- El método `mostrarFormularioEditar()` maneja las solicitudes GET a `"/estancias/editar/{id}"`. Recupera la estancia correspondiente al ID proporcionado del repositorio utilizando el método `findById()`. Agrega el objeto `"Estancia"` al modelo y devuelve una vista llamada `"estanciaedit"`.
- El método `eliminarEstancia()` maneja las solicitudes GET a `"/estancias/eliminar/{id}"`. Elimina la estancia correspondiente al ID proporcionado utilizando el método `deleteById()` del repositorio. Luego redirige a la página principal de `"estancias"`.

Las vistas están disponibles en la carpeta **resources/templates** (la misma carpeta *resources* donde configuramos la aplicación).

## 8.2 Ejemplo de controlador un poco más elaborado

A continuación vemos el controlador llamado **IncidenciaController** que maneja las solicitudes relacionadas con las incidencias en el sistema.

```
1 package com.iesvdc.acceso.tema03.inventario.controller;
2
3 import com.iesvdc.acceso.tema03.inventario.model.Estancia;
4 import com.iesvdc.acceso.tema03.inventario.model.Incidencia;
5 import com.iesvdc.acceso.tema03.inventario.model.Inventario;
6 import com.iesvdc.acceso.tema03.inventario.model.Usuario;
7 import com.iesvdc.acceso.tema03.inventario.repos.RepoEstancia;
8 import com.iesvdc.acceso.tema03.inventario.repos.RepoIncidencia;
9 import com.iesvdc.acceso.tema03.inventario.repos.RepoInventario;
10 import com.iesvdc.acceso.tema03.inventario.repos.RepoUsuario;
11
12 import org.springframework.beans.factory.annotation.Autowired;
```

```
13 import org.springframework.stereotype.Controller;
14 import org.springframework.ui.Model;
15 import org.springframework.web.bind.annotation.*;
16
17 import java.util.ArrayList;
18 import java.util.Optional;
19 import java.util.List;
20
21 @Controller
22 @RequestMapping("/incidencias")
23 public class IncidenciaController {
24
25     @Autowired
26     private RepoIncidencia repoIncidencia;
27     @Autowired
28     private RepoUsuario repoUsuario;
29     @Autowired
30     private RepoEstancia repoEstancia;
31     @Autowired
32     private RepoInventario repoInventario;
33
34     @GetMapping("")
35     public String listarIncidencias(Model model) {
36         List<Incidencia> incidencias = repoIncidencia.findAll();
37         model.addAttribute("incidencias", incidencias);
38         return "incidencias";
39     }
40
41     @GetMapping("/create")
42     public String mostrarFormularioCrearIncidenciaPorEstancia(
43         @RequestParam(name="estanciaId") Optional<Integer> estanciaId,
44         Model model) {
45
46         List<Inventario> inventarios = new ArrayList<Inventario>();
47         Estancia estancia = null;
48
49         if (estanciaId.isPresent()) {
50             Optional<Estancia> oestancia = repoEstancia.findById(
51                 estanciaId.get());
52             if (oestancia.isPresent()) {
53                 inventarios = repoInventario.findByEstancia(oestancia.
54                     get());
55                 estancia = oestancia.get();
56             }
57             else {
58                 inventarios = repoInventario.findAll();
59             }
60         } else {
61             inventarios = repoInventario.findAll();
62         }
63     }
64 }
```

```
61     List<Usuario> usuarios = repoUsuario.findAll();
62     List<Usuario> operarios = repoUsuario.findAll();
63     List<Estancia> estancias = repoEstancia.findAll();
64
65     model.addAttribute("usuarios", usuarios);
66     model.addAttribute("operarios", operarios);
67     model.addAttribute("estancias", estancias);
68     model.addAttribute("inventarios", inventarios);
69     model.addAttribute("incidencia", new Incidencia());
70     model.addAttribute("estancia", estancia);
71
72     return "incidenciacreate";
73 }
74
75 @PostMapping("/guardar")
76 public String guardarIncidencia(@ModelAttribute("incidencia")
77     Incidencia incidencia) {
78     repoIncidencia.save(incidencia);
79     return "redirect:/incidencias";
80 }
81
82 @GetMapping("/editar/{id}")
83 public String mostrarFormularioEditarIncidencia(@PathVariable("id")
84     Integer id, Model model) {
85     Incidencia incidencia = repoIncidencia.findById(id).get();
86     List<Usuario> usuarios = repoUsuario.findAll();
87     List<Usuario> operarios = repoUsuario.findAll();
88     List<Estancia> estancias = repoEstancia.findAll();
89     model.addAttribute("usuarios", usuarios);
90     model.addAttribute("operarios", operarios);
91     model.addAttribute("estancias", estancias);
92     model.addAttribute("incidencia", incidencia);
93     return "incidenciaedit";
94 }
95
96 @GetMapping("/eliminar/{id}")
97 public String eliminarIncidencia(@PathVariable("id") Integer id) {
98     repoIncidencia.deleteById(id);
99     return "redirect:/incidencias";
100 }
```

Este controlador maneja las operaciones relacionadas con las incidencias, como listar, crear, editar y eliminar incidencias. Interactúa con los repositorios correspondientes para obtener y guardar los datos necesarios, y utiliza modelos y vistas para representar la información en la interfaz de usuario así:

- **listarIncidencias**: Este método maneja la solicitud GET a la ruta “/incidencias”. Obtiene todas las incidencias del repositorio `RepoIncidencia` y las agrega al modelo con el nombre

“incidencias”. Luego devuelve la vista “incidencias”.

- **mostrarFormularioCrearIncidenciaPorEstancia**: Este método maneja la solicitud GET a la ruta “/incidencias/create”. Toma un parámetro de consulta opcional llamado “estanciaId”. Si se proporciona un “estanciaId”, se busca la estancia correspondiente en el repositorio **RepoEstancia** y se obtienen los inventarios relacionados con esa estancia del repositorio **RepoInventario**. También se obtienen todos los usuarios, operarios y estancias del repositorio correspondiente. Luego, agrega todos estos datos al modelo y devuelve la vista “incidenciacreate”.
- **guardarIncidencia**: Este método maneja la solicitud POST a la ruta “/incidencias/guardar”. Recibe un objeto de tipo **Incidencia** mediante el parámetro **@ModelAttribute** y lo guarda en el repositorio **RepoIncidencia** utilizando el método **save()**. Luego redirige a la ruta “/incidencias”.
- **mostrarFormularioEditarIncidencia**: Este método maneja la solicitud GET a la ruta “/incidencias/editar/{id}”. Toma el parámetro de ruta “{id}” que representa el ID de la incidencia a editar. Busca la incidencia correspondiente en el repositorio **RepoIncidencia** y obtiene todos los usuarios, operarios y estancias del repositorio correspondiente. Luego agrega estos datos al modelo junto con la incidencia encontrada y devuelve la vista “incidenciaedit”.
- **eliminarIncidencia**: Este método maneja la solicitud GET a la ruta “/incidencias/eliminar/{id}”. Toma el parámetro de ruta “{id}” que representa el ID de la incidencia a eliminar. Utiliza el método **deleteById()** del repositorio **RepoIncidencia** para eliminar la incidencia correspondiente y luego redirige a la ruta “/incidencias”.

## 9 Vistas

Las vistas son componentes que se encargan de generar la representación visual de los datos para que puedan ser presentados al usuario. Las vistas son responsables de mostrar la información de una manera adecuada y estructurada, y permiten al usuario interactuar con la aplicación.

Thymeleaf es un motor de plantillas muy utilizado en aplicaciones Spring. Puedes acceder a su documentación en este enlace: <https://www.thymeleaf.org/doc/tutorials/3.1/thymeleafspring.html>.

Proporciona una forma sencilla y de integrar las plantillas HTML con el código Java en el lado del servidor gracias a:

1. **Sintaxis amigable:** Thymeleaf utiliza una sintaxis natural y fácil de leer que se asemeja a HTML. Esto facilita la creación y el mantenimiento de las plantillas, ya que no es necesario aprender una sintaxis nueva y compleja.
2. **Expresiones:** Thymeleaf permite utilizar expresiones en las plantillas para acceder a los datos y manipularlos. Estas expresiones son similares a las expresiones de lenguaje de plantillas (EL) utilizadas en otros motores de plantillas, lo que hace que sea fácil y familiar trabajar con ellas.
3. **Integración con Spring:** Thymeleaf está diseñado específicamente para trabajar con el framework de Spring. Se integra de manera transparente con otros componentes de Spring, como los controladores y los modelos, lo que simplifica el proceso de desarrollo de aplicaciones web.
4. **Procesamiento del lado del servidor:** Thymeleaf se ejecuta en el servidor, lo que significa que puede acceder a los datos y realizar operaciones antes de enviar la respuesta al cliente. Esto permite generar dinámicamente el contenido de las páginas en función de los datos y la lógica de negocio.
5. **Thymeleaf ofrece una amplia gama de características adicionales,** como la internacionalización, la validación de formularios, la manipulación de URL, la iteración de listas y la condicionalización de contenido. Estas características hacen que el desarrollo de aplicaciones web sea más eficiente y productivo.

En Spring, podemos situar nuestras plantillas (templates) en la carpeta **main/resources/templates**. Recuerda que el nombre del archivo debe ser lo mismo que retorna el controlador, quien es el encargado de buscar y/o procesar los datos para estas vistas.

Si queremos un ejemplo de CRUD completo, necesitaremos al menos tres vistas:

- Listar (de este listado, pulsando un botón podemos saltar a editar ese objeto o bien eliminarlo).
- Editar
- Crear

Para ayudarnos en esta tarea, como hay porciones del código que van a ser repetitivas, como las cabeceras, el pie de página, logotipos, menús..., usaremos fragmentos.

## 9.1 Fragmentos

En Thymeleaf, los **fragments** son secciones de una plantilla HTML que se pueden reutilizar en varias páginas. Permiten separar y organizar el código HTML en componentes más pequeños y modulares, lo que facilita el mantenimiento y la reutilización del código.

Un fragmento se define en una plantilla mediante la etiqueta `<th:block>` y se puede utilizar en otras plantillas utilizando la directiva `th:insert` o `th:replace`. Aquí hay un ejemplo de cómo se puede definir y utilizar un fragmento en Thymeleaf:

1. Definición de un fragmento en una plantilla:

```
1 <!-- plantilla fragmento.html -->
2 <th:block th:fragment="nombreDelFragmento">
3     <!-- contenido del fragmento -->
4     <p>Este es el contenido del fragmento</p>
5 </th:block>
```

2. Uso del fragmento en otra plantilla:

```
1 <!-- otra plantilla.html -->
2 <!DOCTYPE html>
3 <html xmlns:th="http://www.thymeleaf.org">
4 <head>
5     <title>Ejemplo de uso de fragmento</title>
6 </head>
7 <body>
8     <div>
9         <!-- inserta el fragmento -->
10        <div th:insert="fragmento.html :: nombreDelFragmento"></div>
11    </div>
12 </body>
13 </html>
```

En este ejemplo, el fragmento con el nombre “nombreDelFragmento” definido en el archivo `fragmento.html` se inserta en la plantilla `otra plantilla.html` utilizando la directiva `th:insert`. El contenido del fragmento se renderizará en el lugar donde se inserta.

Los fragments son especialmente útiles cuando se desea compartir código HTML común entre varias páginas, como encabezados, pies de página, menús de navegación, formularios, etc. Al utilizar fragments, se puede evitar la repetición de código y mantener una estructura modular y reutilizable en las plantillas Thymeleaf.

Nosotros vamos a crear el siguiente archivo con los menús y el pie de página (archivo **main/resources/templates/fragments/general.html**):

```
1 <!DOCTYPE html>
```

```
2 <html lang="es">
3
4 <head th:fragment="headerfiles">
5   <meta charset="utf-8">
6   <meta http-equiv="X-UA-Compatible" content="IE=edge">
7   <meta name="description" content="Gestión de Inventarios">
8   <meta name="viewport" content="width=device-width, initial-scale=1">
9   <link rel="stylesheet" href="/webjars/bootstrap/5.2.3/css/bootstrap.
    css">
10  <link rel="stylesheet" href="/webjars/webjars/font-awesome/6.4.0/css/
    all.css">
11 </head>
12
13 <body>
14   <div th:fragment="navigation">
15     <nav class="navbar navbar-expand-lg bg-light">
16       <div class="container-fluid">
17         <a class="navbar-brand" href="#">Inventario</a>
18         <button class="navbar-toggler" type="button" data-bs-toggle="
            collapse" data-bs-target="#navbarSupportedContent"
19           aria-controls="navbarSupportedContent" aria-expanded="false"
20           aria-label="Toggle navigation">
21           <span class="navbar-toggler-icon"></span>
22         </button>
23         <div class="collapse navbar-collapse" id="
            navbarSupportedContent">
24           <ul class="navbar-nav me-auto mb-2 mb-lg-0">
25             <li class="nav-item">
26               <a class="nav-link active" aria-current="page" href="/">
                Inicio</a>
27             </li>
28             <li class="nav-item dropdown">
29               <a class="nav-link dropdown-toggle" href="#" role="button"
                data-bs-toggle="dropdown"
30               aria-expanded="false">
31                 Incidencias
32               </a>
33               <ul class="dropdown-menu">
34                 <li><a class="dropdown-item" href="/incidencias">Gestió
                    n Incidencias</a></li>
35                 <li><a class="dropdown-item" href="/incidencias/create"
                    >Alta incidencia</a></li>
36               </ul>
37             </li>
38             <li class="nav-item dropdown">
39               <a class="nav-link dropdown-toggle" href="#" role="button"
                data-bs-toggle="dropdown"
40               aria-expanded="false">
41                 Estancias
42             </a>
```



```

43         <ul class="dropdown-menu">
44             <li><a class="dropdown-item" href="/estancias">Listado
45                 estancias</a></li>
46             <li><a class="dropdown-item" href="/estancias/create">
47                 Alta estancias</a></li>
48         </ul>
49     </li>
50     <li class="nav-item dropdown">
51         <a class="nav-link dropdown-toggle" href="#" role="button"
52             data-bs-toggle="dropdown"
53             aria-expanded="false">
54             Inventario
55         </a>
56         <ul class="dropdown-menu">
57             <li><a class="dropdown-item" href="/inventarios">Gesti3
58                3n inventario</a></li>
59             <li><a class="dropdown-item" href="/inventarios/create">
60                 Alta inventario</a></li>
61         </ul>
62     </li>
63     <li class="nav-item dropdown">
64         <a class="nav-link dropdown-toggle" href="#" role="button"
65             data-bs-toggle="dropdown"
66             aria-expanded="false">
67             Usuarios
68         </a>
69         <ul class="dropdown-menu">
70             <li><a class="dropdown-item" href="/users">Listado
71                 usuarios</a></li>
72             <li><a class="dropdown-item" href="/users/create">Alta
73                 usuarios</a></li>
74         </ul>
75     </li>
76 </div>
77 </div>
78 </div>
79 </nav>
80 </div>
81
82
83 <div th:fragment="footer" class="footer">
84     <p>Copyleft by IES VDC</p>
85     <script src="/webjars/jquery/3.6.4/jquery.js"></script>

```

```
86     <script src="/webjars/popper.js/2.9.3/umd/popper.min.js"></script>
87     <script src="/webjars/bootstrap/5.2.3/js/bootstrap.js"></script>
88   </div>
89 </body>
90
91 </html>
```

En este archivo tenemos:

1. `<head th:fragment="headerfiles">`: Esta sección define las etiquetas `<head>` del documento HTML. Incluye metadatos, como la codificación de caracteres, la descripción de la página y las referencias a archivos CSS y fuentes externas.
2. `<body>`: Esta sección representa el cuerpo principal de la página HTML.
3. `<div th:fragment="navigation">`: Esta sección define la barra de navegación de la página. Utiliza la sintaxis de fragmento de Thymeleaf para reutilizar el código de navegación en varias páginas. La barra de navegación tiene enlaces a diferentes secciones de la aplicación, como incidencias, estancias, inventario y usuarios.
4. `<div th:fragment="footer" class="footer">`: Esta sección define el pie de página de la página HTML. También utiliza la sintaxis de fragmento de Thymeleaf para reutilizar el código del pie de página en varias páginas. El pie de página muestra el mensaje “Copyleft by IES VDC” y carga los archivos JavaScript necesarios.

En resumen, proporciona una barra de navegación con enlaces a diferentes secciones de la aplicación y un pie de página con información de derechos de autor. Thymeleaf se utiliza para la generación dinámica de contenido y la reutilización de fragmentos de código en diferentes páginas.

## 9.2 Ejemplo de vista para listar

```
1 <!DOCTYPE html>
2 <html lang="es">
3
4 <head>
5   <title>Gestión Inventarios: Listado de Estancias</title>
6   <th:block th:include="fragments/general.html :: headerfiles"></th:
   block>
7 </head>
8
9
10 <body>
11   <div class="container-fluid">
12     <div th:replace="fragments/general.html :: navigation"> </div>
13
14     <h3>Listado de Estancias</h3>
```

```

15     <table class="table table-striped">
16         <thead>
17             <tr>
18                 <th>ID</th>
19                 <th>Nombre</th>
20                 <th>Planta</th>
21                 <th>Descripción</th>
22                 <th>Acciones</th>
23             </tr>
24         </thead>
25         <tbody>
26             <tr th:each="estancia : ${estancias}">
27                 <td th:text="${estancia.id}"></td>
28                 <td th:text="${estancia.nombre}"></td>
29                 <td th:text="${estancia.planta}"></td>
30                 <td th:text="${estancia.descripcion}"></td>
31                 <td>
32                     <a class="btn btn-primary" th:href="@{/estancias/editar/{id}(id=${estancia.id})}">
33                         Editar</a>
34                     <a class="btn btn-danger" th:href="@{/estancias/eliminar/{id}(id=${estancia.id})}">Eliminar</a>
35                 </td>
36             </tr>
37         </tbody>
38     </table>
39     <div th:replace="fragments/general.html :: footer"></div>
40 </div>
41 </body>
42
43 </html>

```

Vamos a centrarnos en la tabla que se genera (como usamos Bootstrap la tabla *tiene* que ser así para que se le dé formato adecuadamente), este código HTML se utiliza para generar una tabla que muestra una lista de estancias. Cada fila de la tabla representa una estancia, con columnas para el ID, el nombre, la planta, la descripción y acciones como editar y eliminar. Thymeleaf se utiliza para iterar sobre la lista de estancias y mostrar dinámicamente los datos en la tabla.

1. `<table class="table table-striped">`: Esta línea define una tabla HTML con la clase CSS “table” y “table-striped” para aplicar estilos de visualización a la tabla.
2. `<thead>`: Esta sección define el encabezado de la tabla.
3. `<tr>`: Esta etiqueta define una fila en el encabezado de la tabla.
4. `<th>`: Estas etiquetas definen las celdas de encabezado en la fila. En este caso, se definen los encabezados “ID”, “Nombre”, “Planta”, “Descripción” y “Acciones”.
5. `<tbody>`: Esta sección define el cuerpo de la tabla.

6. `<tr th:each="estancia : ${estancias}">`: Esta etiqueta define una fila en el cuerpo de la tabla y utiliza la sintaxis de iteración de Thymeleaf para iterar sobre la lista de estancias proporcionada en el modelo (variable `estancias`).
7. `<td th:text="${estancia.id}"></td>`: Estas etiquetas definen las celdas de datos en la fila y utilizan la sintaxis de expresión de Thymeleaf (`th:text`) para mostrar los valores de las propiedades de la estancia. En este caso, se muestra el ID, el nombre, la planta y la descripción de cada estancia.
8. `<td>`: Esta celda de datos contiene los botones de “Editar” y “Eliminar” para cada estancia. Los botones se enlazan a las URL correspondientes utilizando la sintaxis de enlace de Thymeleaf (`th:href`).

### 9.3 Ejemplo sencillo para crear

Este formulario directamente para el objeto “estancia” para crearlo:

```

1 <!DOCTYPE html>
2 <html lang="es">
3
4 <head>
5     <title>Gestión Inventarios: Crear Estancia</title>
6     <th:block th:include="fragments/general.html :: headerfiles"></th:
      block>
7 </head>
8
9
10 <body>
11     <div class="container-fluid">
12         <div th:replace="fragments/general.html :: navigation"> </div>
13
14         <h2>Crear Estancia</h2>
15         <form th:action="@{/estancias/guardar}" th:object="${estancia}"
              method="post">
16             <div class="form-group">
17                 <label for="nombre">Nombre:</label>
18                 <input type="text" id="nombre" th:field="*{nombre}"
                      class="form-control">
19             </div>
20             <div class="form-group">
21                 <label for="planta">Planta:</label>
22                 <input type="number" id="planta" th:field="*{planta}"
                      class="form-control">
23             </div>
24             <div class="form-group">
25                 <label for="descripcion">Descripción:</label>
26                 <textarea id="descripcion" th:field="*{descripcion}"
                      class="form-control"></textarea>

```

```
27         </div>
28         <button type="submit" class="btn btn-primary">Guardar</
          button>
29     </form>
30
31     <div th:replace="fragments/general.html :: footer"></div>
32 </div>
33 </body>
34
35 </html>
```

Aquí podemos ver un ejemplo del formulario:

Inventario

## Crear Estancia

Nombre:

Planta:

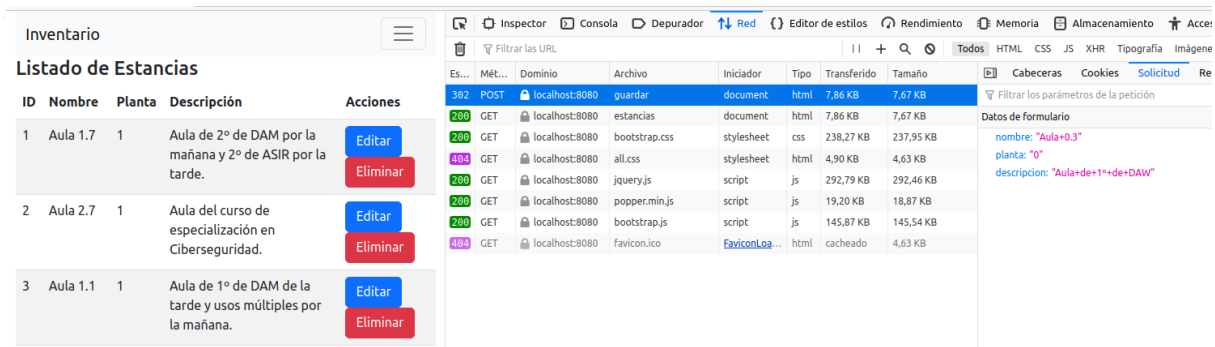
Descripción:

Guardar

Copyleft by IES VDC

**Figura 15:** Crear Estancia, Formulario

Cuando pulsamos en “GUARDAR” hacemos un método POST que envía el objeto que recibe el controlador en la ruta **/estancias/guardar** y lo almacena en la base de datos:



**Figura 16:** POST del formulario

## 9.4 Ejemplo sencillo de modificar

El código es prácticamente el mismo pero con una diferencia: ahora el controlador pasa a la vista los datos que se quieren modificar y por eso se renderizan:

```

1  <!DOCTYPE html>
2  <html lang="es">
3
4  <head>
5      <title>Gestión Inventarios: Editar Estancia</title>
6      <th:block th:include="fragments/general.html :: headerfiles"></th:
7  </head>
8
9
10 <body>
11     <div class="container-fluid">
12         <div th:replace="fragments/general.html :: navigation"> </div>
13
14         <h3>Editar Estancias</h3>
15         <form th:action="@{/estancias/guardar}" th:object="${estancia}"
16             method="post">
17             <input type="hidden" th:field="*{id}">
18             <div class="form-group">
19                 <label for="nombre">Nombre:</label>
20                 <input type="text" id="nombre" th:field="*{nombre}"
21                     class="form-control">
22             </div>
23             <div class="form-group">
24                 <label for="planta">Planta:</label>
25                 <input type="number" id="planta" th:field="*{planta}"
26                     class="form-control">
27             </div>
28             <div class="form-group">
29                 <label for="descripcion">Descripción:</label>

```

```

27         <textarea id="descripcion" th:field="*{descripcion}"
28             class="form-control"></textarea>
29         <button type="submit" class="btn btn-primary">Guardar</
30             button>
31     </form>
32     <div th:replace="fragments/general.html :: footer"></div>
33 </div>
34 </body>
35
36 </html>

```

## 9.5 Ejemplo algo más completo de listar

En este ejemplo, accedemos a atributos de los objetos que manejamos, de una manera muy similar a como lo haríamos con Java o JSON. Se trata de un listado de incidencias:

```

1  <!DOCTYPE html>
2  <html lang="es">
3
4      <head>
5          <title>Gestión Inventarios: Listado de Incidencias</title>
6          <th:block th:include = "fragments/general.html :: headerfiles">
7              </th:block>
8      </head>
9
10     <body>
11         <div class="container-fluid">
12             <div th:replace = "fragments/general.html :: navigation"> <
13                 /div>
14
15             <h3>Listar Incidencias</h3>
16             <table class="table">
17                 <thead>
18                     <tr>
19                         <th>ID</th>
20                         <th>Inventariable</th>
21                         <th>Asunto</th>
22                         <th>Descripción</th>
23                         <th>Usuario</th>
24                         <th>Operario</th>
25                         <th>Estado</th>
26                         <th>Acciones</th>
27                     </tr>
28                 </thead>
29                 <tbody>

```

```

30         <tr th:each="incidencia : ${incidencias}">
31             <td th:text="${incidencia.id}"></td>
32             <td th:text="${incidencia.inventario.nombre}"><
33                 <td th:text="${incidencia.asunto}"></td>
34                 <td th:text="${incidencia.descripcion}"></td>
35                 <td th:text="${incidencia.usuario.username}"></
36                 <td th:text="${incidencia.operario.username}"><
37                 <td th:text="${incidencia.estado}"></td>
38                 <td>
39                     <a th:href="@{/incidencias/editar/{id}(id=${
40                         ${incidencia.id})}" class="btn btn-
41                         primary">Editar</a>
42                     <a th:href="@{/incidencias/eliminar/{id}(id
43                         =${incidencia.id})}" class="btn btn-
44                         danger">Eliminar</a>
45                 </td>
46             </tr>
47         </tbody>
48     </table>
49     <a href="/incidencias/create" class="btn btn-success">Crear
50         nueva incidencia</a>
51     <div th:replace = "fragments/general.html :: footer"></div>
52 </div>
53 </body>
54 </html>

```

La vista proporcionada muestra una lista de incidencias en un formato tabular, y además tenemos:

- En la sección `<head>`, se establece el título de la página como “Gestión Inventarios: Listado de incidencia”. Además, se incluye un bloque `<th:block>` que hace referencia a otro fragmento llamado “headerfiles” definido en un archivo externo llamado “general.html”. **Fíjate como siempre dejamos fuera el `<title>` de los fragmentos o plantillas, esto es así para poder dar a cada vista o página de vista el título correcto.**
- El contenido principal de la página se encuentra dentro del elemento `<body>`. Comienza con un contenedor fluido `<div class="container-fluid">` que establece un diseño flexible para el contenido.
- Dentro del contenedor, se incluye otro fragmento llamado “navigation” mediante el uso del atributo `th:replace`. Este fragmento se define en el archivo “general.html” y contiene la navegación de la página, como la barra de navegación con enlaces a diferentes secciones.
- A continuación, se muestra un encabezado `<h3>` que indica “Listar Incidencias”.
- Después, se presenta una tabla `<table>` con encabezados de columna `<thead>` y filas de



datos `<tbody>`. Los encabezados de columna incluyen: ID, Inventariable, Asunto, Descripción, Usuario, Operario y Estado.

- Mediante la directiva `th:each`, se realiza un bucle sobre la lista de incidencias `#{incidencias}` y se crea una fila en la tabla para cada incidencia.
- Dentro de cada fila, se utilizan expresiones Thymeleaf `th:text` para mostrar los valores correspondientes de cada incidencia en las celdas de la tabla.
- Además, en la última columna de cada fila se agregan enlaces `<a>` que permiten editar y eliminar la incidencia. Estos enlaces utilizan la sintaxis `th:href` para establecer las URL correspondientes, con parámetros dinámicos como el ID de la incidencia.
- Al final de la tabla, hay un enlace `<a>` para crear una nueva incidencia. Este enlace apunta a la URL `/incidencias/create` y se muestra como un botón verde con el estilo `btn-success`.
- Finalmente, se incluye el fragmento “footer” mediante la directiva `th:replace`. Este fragmento se define en el archivo `general.html` y contiene información de pie de página, como el mensaje de derechos de autor y la inclusión de archivos JavaScript necesarios.

## 9.6 Maestro-detalle para crear incidencia

En programación, el término “maestro-detalle” se refiere a una relación entre dos conjuntos de datos, donde un conjunto es considerado el “maestro” y el otro es el “detalle”. Esta relación se utiliza para representar una estructura jerárquica entre los datos.

El conjunto de datos “maestro” contiene información principal o de nivel superior, mientras que el conjunto de datos “detalle” contiene información relacionada y específica asociada a cada elemento del conjunto “maestro”. Por lo general, esta relación se establece mediante un identificador único que vincula los elementos del conjunto “detalle” al elemento correspondiente del conjunto “maestro”.

Un ejemplo común de maestro-detalle es una base de datos que almacena información sobre clientes y sus pedidos. En este caso, el conjunto de datos “maestro” sería la tabla de clientes, donde cada registro representa un cliente con su información básica como nombre, dirección, etc. El conjunto de datos “detalle” sería la tabla de pedidos, donde cada registro representa un pedido realizado por un cliente específico, vinculado a través del identificador único del cliente.

Al utilizar la estructura maestro-detalle, se pueden mostrar los datos de manera jerárquica, por ejemplo, mostrando la lista de clientes y permitiendo expandir cada cliente para ver sus pedidos asociados. Esto facilita la navegación y comprensión de la información relacionada. Además, al realizar operaciones como la inserción, actualización o eliminación de datos, se deben mantener las relaciones maestro-detalle para garantizar la integridad de los datos.

Nosotros vamos a hacer un maestro detalle para crear incidencias, ¿cómo? pues primero seleccionamos la estancia donde estamos y eso nos filtrará los activos inventariables que existen. Así vamos a tener

dos formularios en la página, uno que hace el filtro con un GET y otro que realmente hace el POST, es decir, crea la incidencia.

```
1 <!DOCTYPE html>
2 <html lang="es">
3
4 <head>
5   <title>Gestión Inventarios: Alta nueva incidencia</title>
6   <th:block th:include="fragments/general.html :: headerfiles"></th:
   block>
7 </head>
8
9 <body>
10   <div class="container-fluid">
11     <div th:replace="fragments/general.html :: navigation"></div>
12
13     <h1>Crear Incidencia</h1>
14
15     <form th:action="@{/incidencias/create}" method="get">
16       <div class="form-group">
17         <label for="estanciaId">Estancia:</label>
18         <select id="estanciaId" name="estanciaId" class="form-
           control" onchange="this.form.submit()">
19           <option> Seleccione una opción </option>
20           <option th:each="estancia : ${estancias}" th:value=
             "${estancia.id}" th:text="${estancia.nombre}"></
           option>
21           <option value="-1"> Mostrar todas las estancias </
           option>
22         </select>
23       </div>
24     </form>
25
26     <div th:if="${estancia != null}">
27       <h4 th:text="${estancia.nombre}"></h4>
28     </div>
29
30     <form th:action="@{/incidencias/guardar}" th:object="${
       incidencia}" method="post">
31       <div class="form-group">
32         <label for="inventario">Item inventariable:</label>
33         <select id="inventario" name="inventario" class="form-
           control">
34           <option th:each="inventario : ${inventarios}" th:
             value="${inventario.id}" th:text="${inventario.
             nombre}"></option>
35
36         </select>
37       </div>
38       <div class="form-group">
39         <label for="asunto">Asunto:</label>
```

```

40         <input type="text" id="asunto" th:field="*{asunto}"
           class="form-control">
41     </div>
42     <div class="form-group">
43         <label for="descripcion">Descripción:</label>
44         <textarea id="descripcion" th:field="*{descripcion}"
           class="form-control"></textarea>
45     </div>
46     <div class="form-group">
47         <label for="usuario">Usuario:</label>
48         <select id="usuario" th:field="*{usuario}" class="form-
           control">
49             <option th:each="usuario : ${usuarios}" th:value="$
               {usuario.id}" th:text="${usuario.username}"></
               option>
50         </select>
51     </div>
52     <div class="form-group">
53         <label for="operario">Operario:</label>
54         <select id="operario" th:field="*{operario}" class="
           form-control">
55             <option th:each="operario : ${usuarios}" th:value="
               ${operario.id}" th:text="${operario.username}"><
               /option>
56         </select>
57     </div>
58
59     <div class="form-group">
60         <label for="estado">Estado:</label>
61         <select id="estado" name="estado" class="form-control">
62             <option value="ABIERTA" th:selected="${incidencia.
               estado == 'ABIERTA'}">ABIERTA</option>
63             <option value="CERRADA" th:selected="${incidencia.
               estado == 'CERRADA'}">CERRADA</option>
64             <option value="ASIGNADA" th:selected="${incidencia.
               estado == 'ASIGNADA'}">ASIGNADA</option>
65             <option value="PROCESANDO" th:selected="${
               incidencia.estado == 'PROCESANDO'}">PROCESANDO</
               option>
66         </select>
67     </div>
68
69     <button type="submit" class="btn btn-primary">Guardar</
       button>
70 </form>
71
72 <div th:replace="fragments/general.html :: footer"></div>
73 </div>
74 </body>
75
76 </html>

```

- En la sección `<head>`, se establece el título de la página como “Gestión Inventarios: Página Principal”. Además, se incluye un bloque `<th:block>` que hace referencia a otro fragmento llamado “headerfiles” definido en un archivo externo llamado “general.html”.
- El contenido principal de la página se encuentra dentro del elemento `<body>`. Comienza con un contenedor fluido `<div class="container-fluid">` que establece un diseño flexible para el contenido.
- Dentro del contenedor, se incluye otro fragmento llamado “navigation” mediante el uso del atributo `th:replace`. Este fragmento se define en el archivo “general.html” y contiene la navegación de la página, como la barra de navegación con enlaces a diferentes secciones.
- Se muestra un encabezado `<h1>` que indica “Crear Incidencia”.
- A continuación, se presenta un formulario `<form>` para crear una nueva incidencia. El formulario envía los datos al servidor utilizando la acción `th:action` establecida como “/incidencias/-create” y el método HTTP “get”.
- Dentro del formulario, se encuentra un grupo de formulario `<div class="form-group">` que contiene una etiqueta `<label>` y un elemento `<select>` que permite al usuario seleccionar una estancia. Los valores de las opciones se generan dinámicamente utilizando la directiva `th:each` para iterar sobre la lista de estancias `${estancias}`. También se agrega una opción adicional para mostrar todas las estancias.
- Después del primer formulario, hay un `<div>` que se muestra solo si `estancia` no es nulo. Dentro de este div, se muestra el nombre de la estancia utilizando la expresión Thymeleaf `th:text="${estancia.nombre}"`.
- A continuación, se presenta otro formulario `<form>` para guardar la incidencia. El formulario utiliza la acción `th:action` establecida como “/incidencias/guardar” y el método HTTP “post”. Además, se utiliza la directiva `th:object` para vincular el objeto de incidencia `${incidencia}` con los campos del formulario.
- Dentro del formulario de incidencia, hay varios grupos de formulario `<div class="form-group">` que contienen etiquetas `<label>` y elementos `<select>`, `<input>` y `<textarea>` para ingresar los detalles de la incidencia. Algunos de los elementos `<select>` generan las opciones dinámicamente utilizando la directiva `th:each` para iterar sobre las listas de inventarios y usuarios.
- El grupo de formulario para el campo “Estado” utiliza un elemento `<select>` y opciones estáticas. La opción seleccionada se determina utilizando la expresión Thymeleaf `th:selected` y la comparación con el estado actual de la incidencia `${incidencia.estado}`.
- Al final del formulario, hay un botón `<button>` con la clase “btn btn-primary” que permite al usuario guardar la incidencia.
- Finalmente, se incluye el fragmento “footer” mediante la directiva `th:replace`. Este fragmento se define en el archivo “general.html” y contiene información de pie de página, como el mensaje de derechos de autor y la inclusión de archivos JavaScript necesarios.



```
2 public class SecurityConfiguration {
3
4     @Autowired
5     DataSource dataSource;
6
7     @Autowired
8     public void configure(AuthenticationManagerBuilder auth) throws
9         Exception {
10         auth.jdbcAuthentication()
11             .dataSource(dataSource)
12             .usersByUsernameQuery(
13                 "select username, password, enabled " +
14                 "from usuario where username = ? ")
15             .authoritiesByUsernameQuery(
16                 "select u.username, ur.authority " +
17                 "from usuario u, usuario_rol ur where u.id=ur.
18                 usuario_id and u.username = ? ");
19     }
20
21     @Bean
22     public BCryptPasswordEncoder passwordEncoder() {
23         return new BCryptPasswordEncoder();
24     }
25
26     @Bean
27     public SecurityFilterChain filterChain(HttpSecurity http) throws
28         Exception {
29
30         http.httpBasic().and().authorizeHttpRequests().anyRequest().
31             authenticated()
32             .and().csrf().disable().cors().and()
33             .formLogin().and()
34             .logout()
35             .logoutUrl("/logout")
36             .invalidateHttpSession(true)
37             .deleteCookies("JSESSIONID");
38
39         return http.build();
40     }
41
42     @Bean
43     public WebSecurityCustomizer webSecurityCustomizer() {
44         return (web) -> web.ignoring().requestMatchers("/images/**", "/"
45             js/**", "/webjars/**");
46     }
47 }
```

Ejemplo de controlador para gestionar el CRUD de los usuarios del sistema con el usuario administrador:

```
1 @RestController
2 @RequestMapping("/admin")
3 public class ServiUsuario {
4     @Autowired
5     RepoUsuario repoUsuario;
6     @Autowired
7     RepoUsuarioRol repoUsuarioRol;
8
9     @GetMapping("usuario")
10    List<Usuario> findAll(){
11        return repoUsuario.findAll();
12    }
13
14    @GetMapping("usuario/{id}")
15    Usuario findById(@PathVariable (value = "id") Integer id){
16        return repoUsuario.findById(id).get();
17    }
18
19    @PostMapping("usuario")
20    Usuario create(@RequestBody Usuario u){
21        String passwdPlano = u.getPassword();
22        BCryptPasswordEncoder bpe = new BCryptPasswordEncoder();
23        u.setPassword(bpe.encode(passwdPlano));
24        Usuario user = repoUsuario.save(u);
25        UsuarioRol ur = new UsuarioRol();
26        ur.setAuthority("ROLE_USER");
27        ur.setUsuario(u);
28        repoUsuarioRol.save(ur);
29        return user;
30    }
31
32    @DeleteMapping("usuario/{id}")
33    void delete(@PathVariable (value = "id") Integer id){
34        repoUsuario.deleteById(id);
35    }
36 }
```

## 11 Login Social/OAuth

TO-DO