

Departamento de Informática

Acceso a Datos: Spring Web, Data y JPA en acción

Juan Gualberto Gutiérrez Marín

Abril 2024

Estos apuntes se han realizado para los alumnos de 2º de Ciclo Superior de Desarrollo de Aplicaciones Multiplataforma.

Este documento se encuentra bajo una licencia Creative Commons de Atribución-CompartirIgual (CC BY-SA).

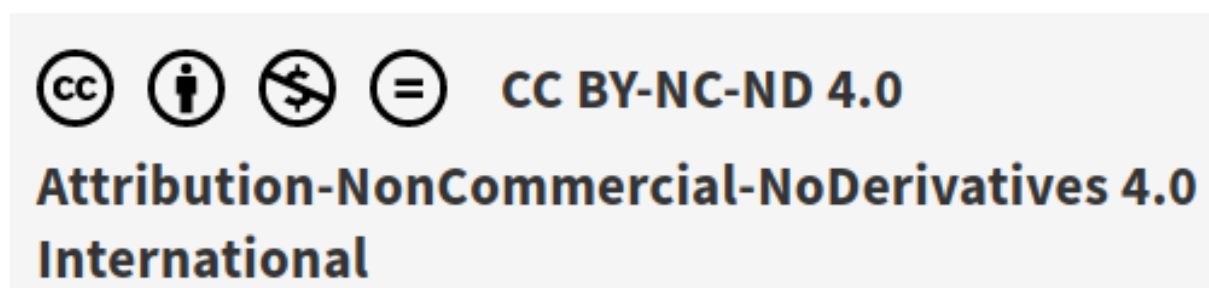


Figura 1: Atribución-CompartirIgual (CC BY-SA)

Esto significa que puedes:

- Compartir: copiar y redistribuir el material en cualquier medio o formato.
- Adaptar: remezclar, transformar y construir sobre el material para cualquier propósito, incluso comercialmente.

Bajo las siguientes condiciones:

- Atribución: debes dar crédito de manera adecuada, proporcionar un enlace a la licencia e indicar si se han realizado cambios. Puedes hacerlo de cualquier manera razonable, pero no de una manera que sugiera que el licenciante te respalda a ti o al uso que hagas del trabajo.
- Compartir igual: si remezclas, transformas o creas a partir del material, debes distribuir tus contribuciones bajo la misma licencia que el original.

Para más detalles, consulta la licencia completa.

Para una versión actualizada de este libro visita esta Web: <https://gitlab.iesvirgendelcarmen.com/juangu/tema05-spring-web-zapapp>.

Índice

1	ZapApp Spring Web	4
2	Herramientas ORM	5
2.1	Spring Data JPA	6
2.2	Spring MVC	6
2.3	Para ampliar...	7
3	Calentando el plato	9
3.1	Prerrequisitos	9
3.2	Puntos clave de Spring	10
3.3	Creación del proyecto tipo	11
3.4	Añadiendo Starters a Spring Initializr	14
3.4.1	Devtools	14
3.4.2	Spring JPA	14
3.4.3	Mysql Driver	14
3.4.4	Lombok	15
4	Contenedores Docker para acelerar el desarrollo	16
4.1	Mysql y Adminer	16
5	PlantUML	18
5.1	Marcando relaciones entre clases	19
6	Análisis	21
6.1	Diagrama de casos de uso	21
6.2	Diagrama de clases	22
7	Las clases entidad	29
7.1	Los POJOs del proyecto	30
7.1.1	Categoría	30
7.1.2	Código Postal	31
7.1.3	Dirección	31
7.1.4	Estado	32
7.1.5	Línea Pedido	32
7.1.6	Pedido	32
7.1.7	Producto	33
7.1.8	Rol	33

7.1.9	Rol Usuario	33
7.1.10	Teléfono	34
7.1.11	Usuario	34
8	Repositorios Spring	36
8.1	RepoCategoria	38
8.2	RepoCodigoPostal	39
8.3	RepoDireccion	39
8.4	RepoLineaPedido	39
8.5	RepoPedido	39
8.6	RepoProducto	39
8.7	RepoRolUsuario	40
8.8	RepoTelefono	40
8.9	RepoUsuario	40
9	Controladores	41
9.1	Listado de rutas de nuestra aplicación	42
9.1.1	Servicio usuario	42
9.1.2	Servicio producto	42
9.1.3	Servicio gestionar “mis pedidos”	51
9.1.4	Servicio envío (estados) pedidos	51
9.1.5	Servicio carro de la compra	52
9.2	Ejemplo de controlador sencillo	55
10	Vistas	56
10.1	Fragmentos	57
11	Seguridad	58
11.0.1	Spring Security	58
12	Bibliografía	62

1 ZapApp Spring Web

Spring Java nos permite desarrollar aplicaciones de manera más rápida, eficaz y corta, saltándonos tareas repetitivas y ahorrándonos líneas de código. En este proyecto vamos a desarrollar una aplicación Web que es una Zapatería online con Spring Boot.

Concretamente vamos a hacer uso de Spring Data y Spring JPA que internamente usan herramientas ORM pero simplifican aún más ofreciendo otra capa más de abstracción.

2 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.

2.1 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.

2.2 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.

2.3 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

Spring Java nos permite desarrollar aplicaciones de manera más rápida, eficaz y corta, saltándonos tareas repetitivas y ahorrándonos líneas de código. En este proyecto vamos a desarrollar una aplicación Web que es una Zapatería online con Spring Boot.

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.

3 Calentando el plato

En el argot de los programadores llamamos calentar el plato al tiempo que perdemos en hacer tareas repetitivas pero necesarias a la hora de empezar un nuevo proyecto.

3.1 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 que veremos más adelante:

Java Extension Pack

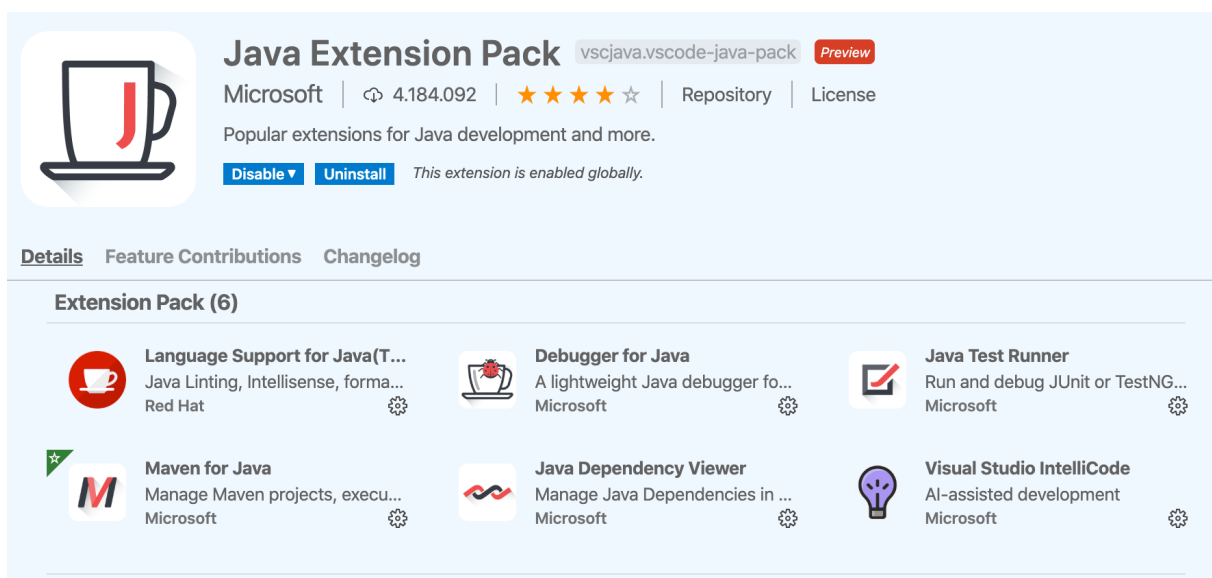
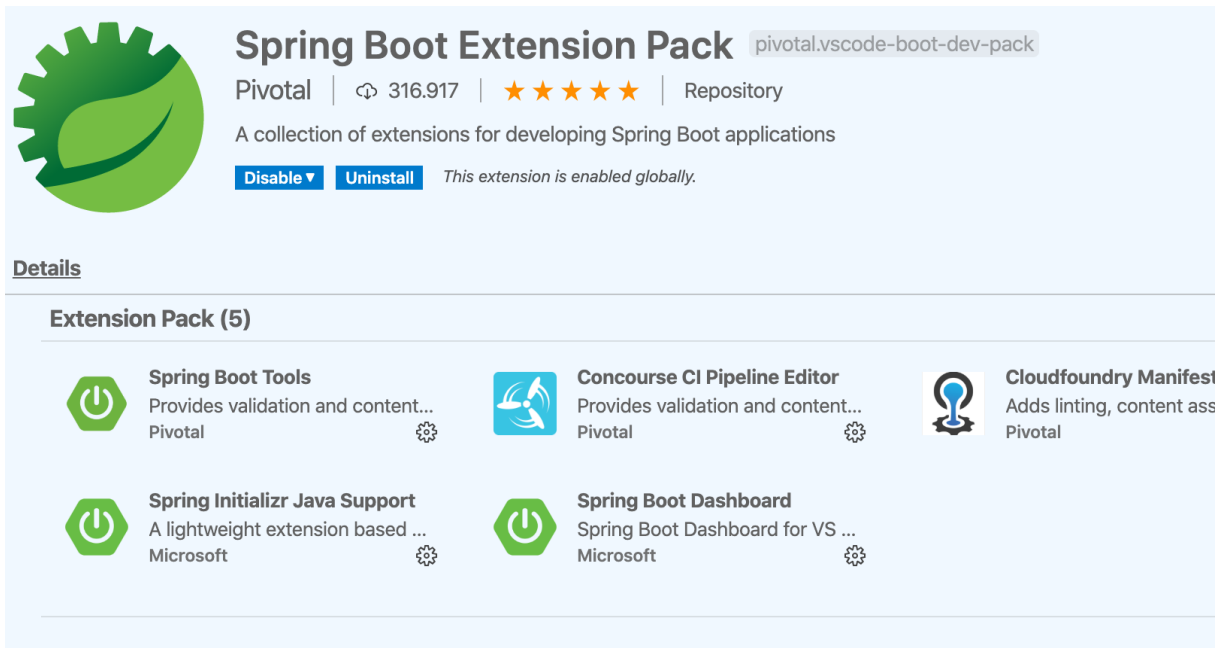


Figura 2: Java Extension Pack

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, a five-star rating, and the word 'Repository'. A description states: 'A collection of extensions for developing Spring Boot applications'. Below this, there are two buttons: 'Disable' (with a dropdown arrow) and 'Uninstall', followed by the text 'This extension is enabled globally.'.

Details

Extension Pack (5)






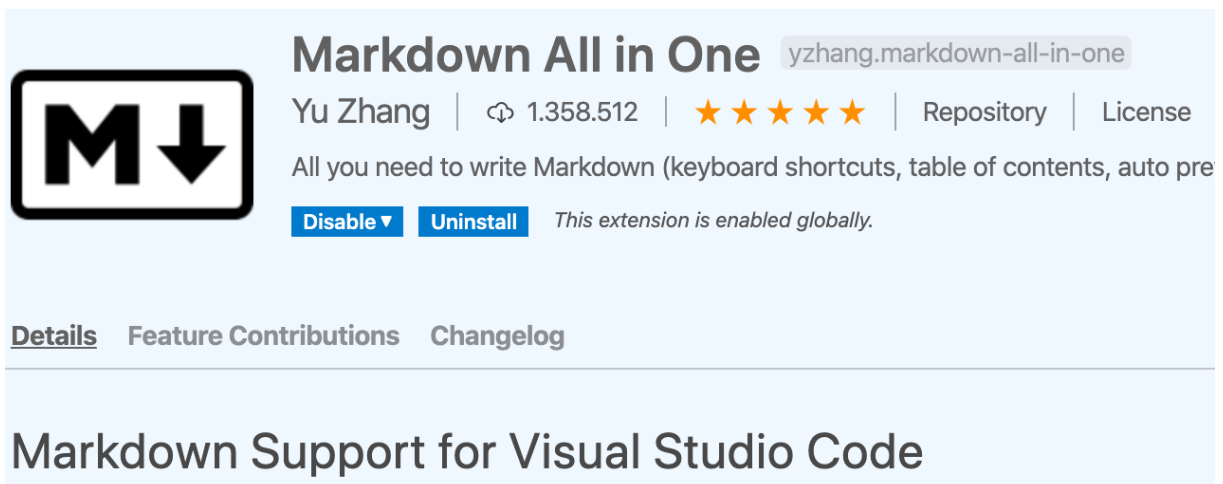
	Spring Boot Tools Provides validation and content... Pivotal		Concourse CI Pipeline Editor Provides validation and content... Pivotal		Cloudfoundry Manifest Adds linting, content ass... Pivotal
	Spring Initializr Java Support A lightweight extension based ... Microsoft		Spring Boot Dashboard Spring Boot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

Figura 3: 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 its right, the title 'Markdown All in One' is displayed in a large, bold font, followed by the repository name 'yzhang.markdown-all-in-one' in a smaller font. Below the title, the publisher 'Yu Zhang' is listed, along with a download count of 1,358,512, a five-star rating, and the words 'Repository' and 'License'. A description states: 'All you need to write Markdown (keyboard shortcuts, table of contents, auto pre...'. Below this, there are two buttons: 'Disable' (with a dropdown arrow) and 'Uninstall', followed by the text 'This extension is enabled globally.'.

Details **Feature Contributions** **Changelog**

Markdown Support for Visual Studio Code

Figura 4: Markdownallinone

3.2 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.

3.3 Creación del proyecto tipo

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

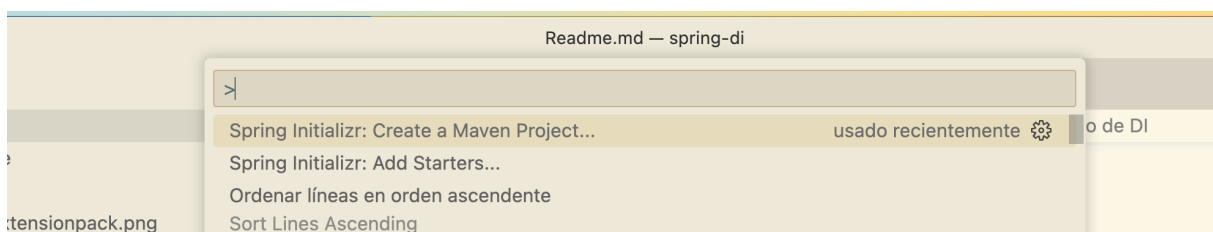


Figura 5: Creación de proyecto Spring con Maven

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

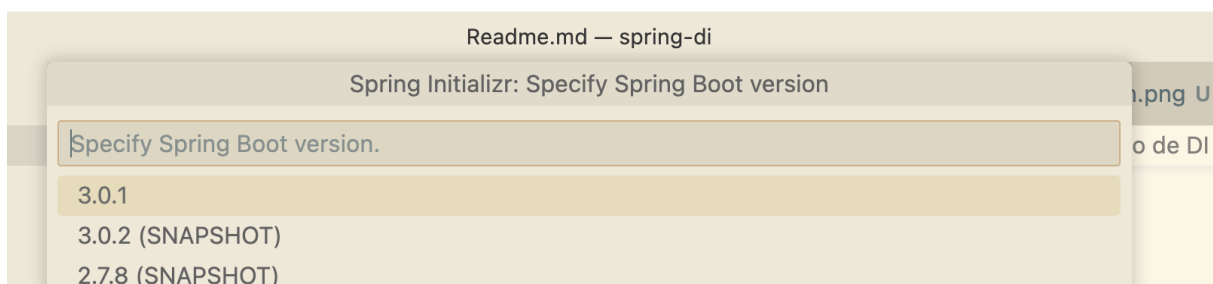


Figura 6: Selección Spring Boot

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

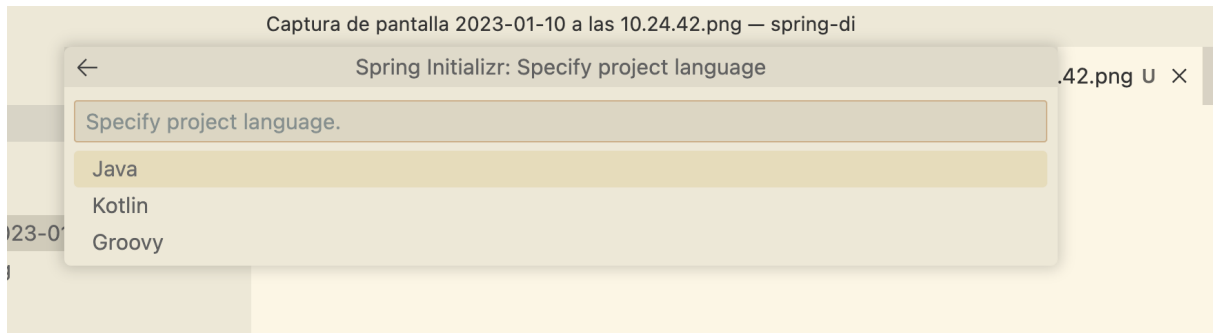


Figura 7: Selección del lenguaje de programación

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

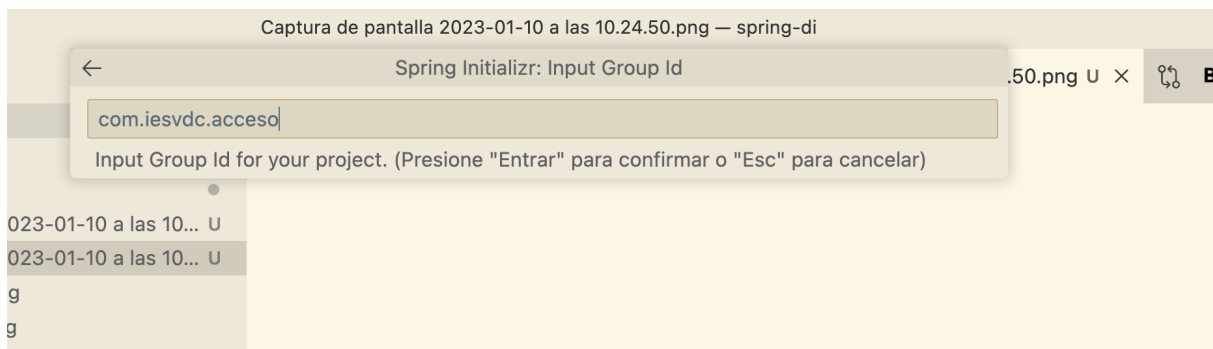


Figura 8: Indicamos el paquete

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

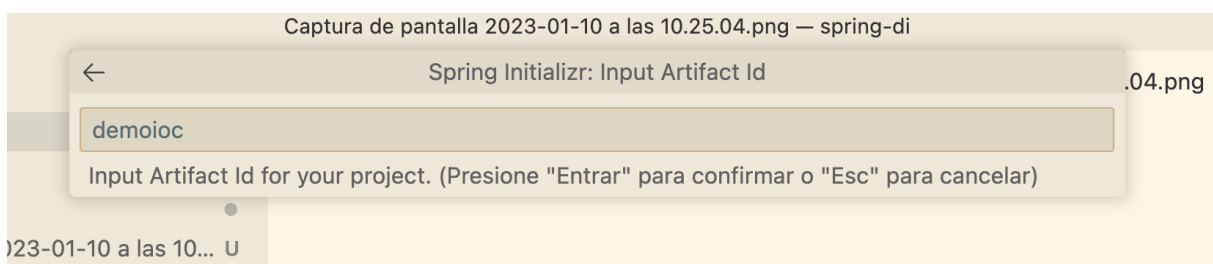


Figura 9: Indicamos el artefacto

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

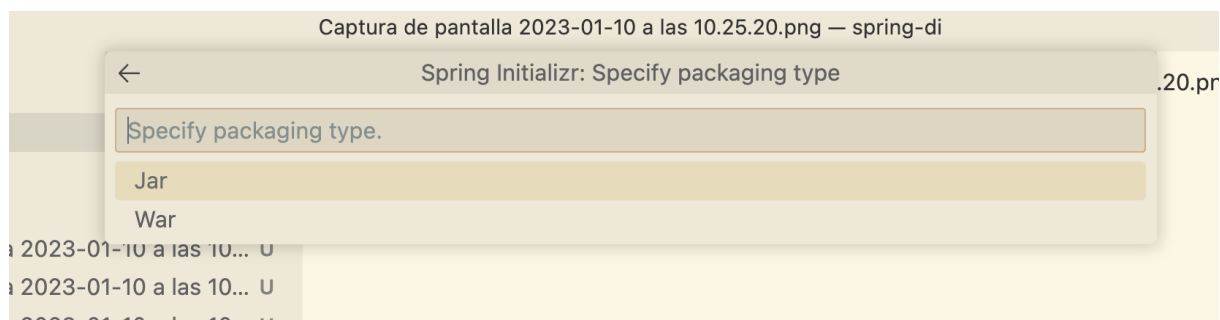


Figura 10: Empaquetado

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

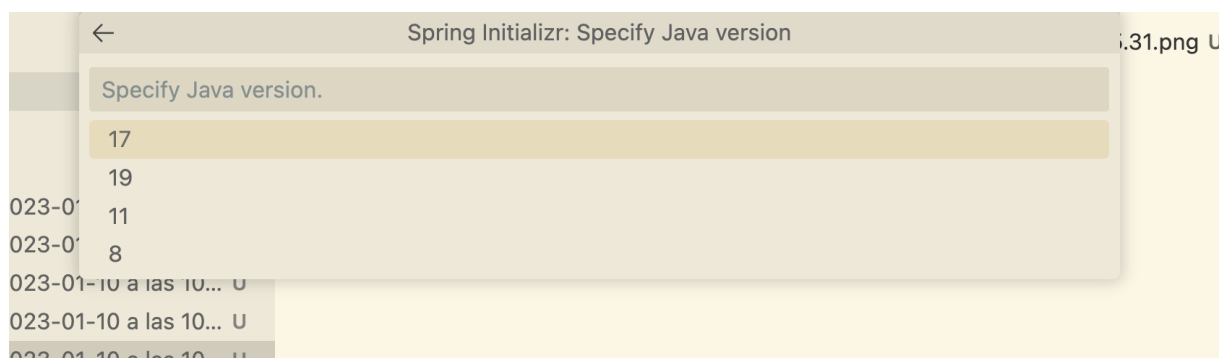


Figura 11: Selección de la versión de Java

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 12: Selección de dependencias Maven

3.4 Añadiendo Starters a Spring Initializr

Cuando creamos un proyecto Maven con Spring, podemos añadir diferentes dependencias de manera nativa que ayudarán en la confección de nuestro proyecto. Veamos las que vamos a usar para nuestro ejemplo en concreto.

3.4.1 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.

3.4.2 Spring JPA

Para las anotaciones de las clases entidad (modelo).

3.4.3 Mysql Driver

Necesario para conectar a MySQL.

3.4.4 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.

4 Contenedores Docker para acelerar el desarrollo

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.

4.1 Mysql y Adminer

A partir de documentación de la imagen oficial 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 PlantUML

PlantUML es una herramienta de código abierto que permite generar diagramas UML (Unified Modeling Language) utilizando una sintaxis simple y legible en forma de texto. Con PlantUML, puedes describir la estructura y relaciones entre los elementos de un sistema o proceso y generar automáticamente diagramas UML correspondientes.

Supongamos que queremos representar un sistema de biblioteca con las siguientes clases:

- **Libro**: Representa un libro en la biblioteca con atributos como título, autor y año de publicación.
- **Biblioteca**: Representa la biblioteca con atributos como nombre y ubicación.
- **Lector**: Representa a una persona que es miembro de la biblioteca y puede tomar prestados libros.

Ahora, podemos utilizar la sintaxis de PlantUML para definir estas clases y sus relaciones:

```
1 @startuml
2 class Libro {
3   + título: string
4   + autor: string
5   + añoPublicación: int
6 }
7
8 class Biblioteca {
9   + nombre: string
10  + ubicación: string
11 }
12
13 class Lector {
14   + nombre: string
15   + númeroDeSocio: string
16 }
17
18 Biblioteca "1" -- "*" Libro
19 Lector "1" -- "*" Libro
20 @enduml
```

En este ejemplo, hemos definido tres clases (**Libro**, **Biblioteca** y **Lector**) junto con sus atributos. Luego, hemos establecido las relaciones entre estas clases utilizando la sintaxis de PlantUML:

- La relación entre **Biblioteca** y **Libro** indica que una biblioteca puede contener múltiples libros (relación uno a muchos), denotada por "1" -- "*".
- La relación entre **Lector** y **Libro** indica que un lector puede tomar prestados múltiples libros (relación uno a muchos), también denotada por "1" -- "*".

5.1 Marcando relaciones entre clases

A continuación vamos a ver unos ejemplos detallados para cada tipo de relación en UML, junto con su representación en PlantUML:

1. Asociación:

- Ejemplo: En un sistema de gestión de estudiantes, una clase **Estudiante** puede estar asociada con una clase **Curso** para indicar qué curso está tomando un estudiante.
- PlantUML:

```
1 @startuml
2 class Estudiante {
3   + nombre: string
4   + edad: int
5 }
6 class Curso {
7   + nombre: string
8   + credits: int
9 }
10 Estudiante -- Curso
11 @enduml
```

2. Agregación:

- Ejemplo: En un sistema de compra en línea, una clase **Carrito** puede estar agregada por múltiples instancias de la clase **Producto**.
- PlantUML:

```
1 @startuml
2 class Carrito {
3   + total: float
4   + productos: List<Producto>
5 }
6 class Producto {
7   + nombre: string
8   + precio: float
9 }
10 Carrito o-- Producto
11 @enduml
```

3. Composición:

- Ejemplo: En un sistema de dibujo, una clase **Dibujo** puede estar compuesta por múltiples instancias de la clase **Figura**, y estas figuras no existen fuera del dibujo.
- PlantUML:

```
1 @startuml
2 class Dibujo {
3   + nombre: string
4 }
5 class Figura {
6   + tipo: string
7 }
8 Dibujo *-- Figura
9 @enduml
```

4. Generalización (Herencia):

- Ejemplo: En un sistema de gestión de empleados, una clase `Empleado` puede ser generalizada por las clases `Gerente` y `EmpleadoTemporal`, que heredan atributos y métodos comunes de la clase `Empleado`.
- PlantUML:

```
1 @startuml
2 class Empleado {
3   + nombre: string
4   + salario: float
5 }
6 class Gerente {
7   + departamento: string
8 }
9 class EmpleadoTemporal {
10  + fechaTermino: date
11 }
12 Gerente --|> Empleado
13 EmpleadoTemporal --|> Empleado
14 @enduml
```

5. Dependencia:

- Ejemplo: En un sistema de reserva de vuelos, una clase `Reserva` puede depender de la clase `Vuelo` para conocer los detalles del vuelo reservado.
- PlantUML:

```
1 @startuml
2 class Reserva {
3   + codigo: string
4 }
5 class Vuelo {
6   + numero: string
7   + origen: string
8   + destino: string
9 }
10 Reserva ..> Vuelo
```

6 Análisis

6.1 Diagrama de casos de uso

Un diagrama de casos de uso modela las interacciones entre actores y casos de uso en un sistema, en concreto nosotros tendremos:

- **Actor:** Son las entidades externas que interactúan con el sistema. Necesitamos representar tres actores: Gestor, Operario y Cliente. Cada uno de ellos tiene un rol específico en el sistema.
- **Casos de uso:** Son las acciones o funcionalidades que realiza el sistema para satisfacer las necesidades de los actores. En el diagrama necesitamos varios casos de uso identificados, cada uno representado por un rectángulo. Los casos de uso incluyen:
 - `gestión precios y productos`: Relacionado con la gestión de precios y productos en el sistema.
 - `envío de pedidos`: Relacionado con el proceso de envío de pedidos.
 - `registrarse y gestionar su información`: Relacionado con las acciones que realiza un cliente para registrarse en el sistema y gestionar su información personal.
 - `gestionar pedidos`: Relacionado con la gestión de pedidos por parte de un cliente.
 - `carrito compra`: Relacionado con las funcionalidades de un carrito de compra en el sistema.
 - `login`: Relacionado con el proceso de inicio de sesión en el sistema.
- **Relaciones:** Las flechas entre actores y casos de uso representan las interacciones entre ellos. Por ejemplo:
 - El Gestor interactúa con el caso de uso `gestión precios y productos`.
 - El Operario interactúa con el caso de uso `envío de pedidos`.
 - El Cliente interactúa con los casos de uso `registrarse y gestionar su información`, `gestionar pedidos`, `carrito compra` y `login`.
- **Relaciones entre casos de uso:** Las líneas sólidas entre los casos de uso indican relaciones de inclusión o dependencia. En este caso, todos los casos de uso están relacionados con el caso de uso `login`, lo que indica que el proceso de inicio de sesión es necesario para acceder a cualquiera de las funcionalidades del sistema.

Código PlantUML para modelar el caso de uso:

```

1 @startuml Diagrama de casos de uso
2
3 usecase (gestión precios\n y productos) as Proc01
4 actor :gestor: as Gestor
5 Gestor --> Proc01
6
7 usecase (envío de pedidos) as Proc02
8 actor :operario \n pedidos : as Operario
9 Operario --> Proc02
10
11 usecase ( gestionar mi \n información) as Proc03
12 usecase (gestionar\n pedidos ) as Proc04
13 usecase (carrito compra) as Proc05
14 actor :cliente \n web : as Cliente
15 Cliente --> Proc03
16 Cliente --> Proc04
17 Cliente --> Proc05
18
19 usecase ( registro / login ) as Proc06
20
21 Proc01 .. Proc06
22 Proc02 .. Proc06
23 Proc03 .. Proc06
24 Proc04 .. Proc06
25 Proc05 .. Proc06
26
27 @enduml

```

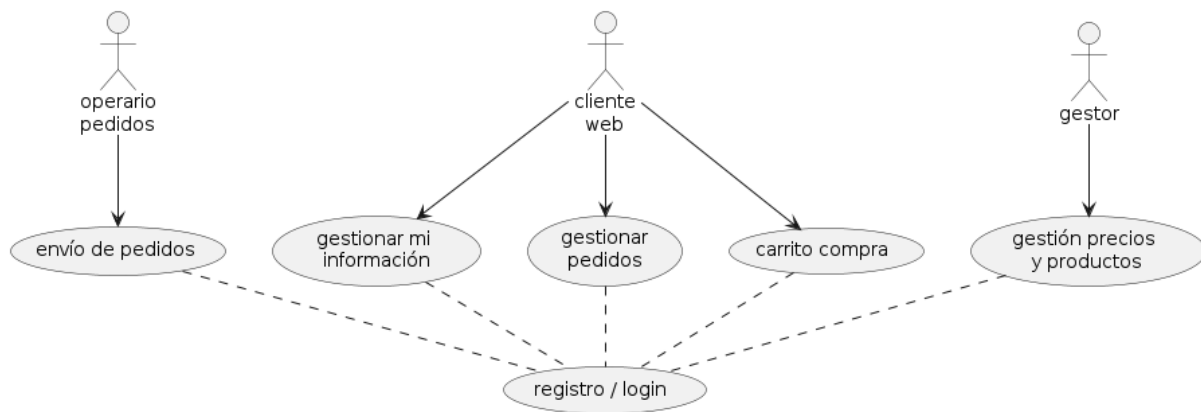


Figura 13: Diagrama de casos de uso

6.2 Diagrama de clases

Para realizar el diagrama de clases vamos a usar de nuevo PlantUML:

```
1 @startuml
2
3 class Usuario {
4     - id: Long
5     - nombre: String
6     - apellido: String
7     - email: String
8     - username: String
9     - password: String
10 }
11
12 class Producto {
13     - id: Long
14     - nombre: String
15     - descripción: String
16     - talla: String
17     - precio: Float
18 }
19
20
21 class Pedido {
22     - id: Long
23     - fecha: Date
24     - observaciones: String
25     - descuento: Float
26     - estado: Estado
27 }
28
29 enum Estado {
30     CARRITO
31     REALIZADO
32     PREPARANDO
33     ENVIADO
34     COMPLETADO
35     INCIDENCIA
36 }
37
38 class LineaPedido {
39     - id: Long
40     - precio: Float
41     - cantidad: Integer
42     ' - producto: Producto '
43 }
44
45 class Categoría {
46     - id: Long
47     - nombre: String
48     - descripcion: String
49     ' - padre: Categoría '
50 }
```



```
51
52 class Dirección {
53     - id: Long
54     - tipoVia: String
55     - nombreVia: String
56     - número: String
57     - planta: String
58     - puerta: String
59     - portal: String
60     - nombre: String
61     ' - codpos: CódigoPostal '
62     ' - usuario: Usuario '
63 }
64
65 class Telefono {
66     - id: Long
67     - códigoPais: Long
68     - número: Long
69     - nombre: String
70     ' - usuario: Usuario '
71 }
72
73 enum Rol {
74     OPERARIO
75     CLIENTE
76     GESTOR
77 }
78
79 class RolUsuario {
80     - rol: Rol
81     - usuario: Usuario
82 }
83
84 class CódigoPostal {
85     - id: Long
86     - CP: Integer
87     - Localidad: String
88     - Municipio: String
89     - Comunidad: String
90     - Pais: String
91 }
92
93 Usuario "1" -- "0..*" Pedido : Realiza_cliente
94 Usuario "1" o-- "1..*" Dirección : Tiene
95 Usuario "1" o-- "1..*" Telefono : Tiene
96 Usuario "1" -- "1..*" RolUsuario : Pertenece
97 Rol "1" -- "1..*" RolUsuario: Tiene
98
99 Pedido "1" -- "0..*" LíneaPedido : Contiene
100 Pedido "0..*" -- "1" Usuario : Asignado_operario
101 Dirección "1" -- "1..*" Pedido: Enviado_a
```

```
102
103 Producto "0..*" -- "0..*" LineaPedido : Forma_parte_de
104 Categoría "1" -- "0..*" Producto : Pertenece_a
105
106 Categoría "1" -- "0..*" Categoría : Pertenece_a
107
108 Dirección "1" -- "0..1" CódigoPostal : Tiene
109 LineaPedido "1" -- "0..1" Estado: Tiene
110
111
112 @enduml
```

Este diagrama de clases en PlantUML describe la estructura de nuestro sistema de gestión de pedidos y productos, donde tenemos las siguientes:

1. Clases:

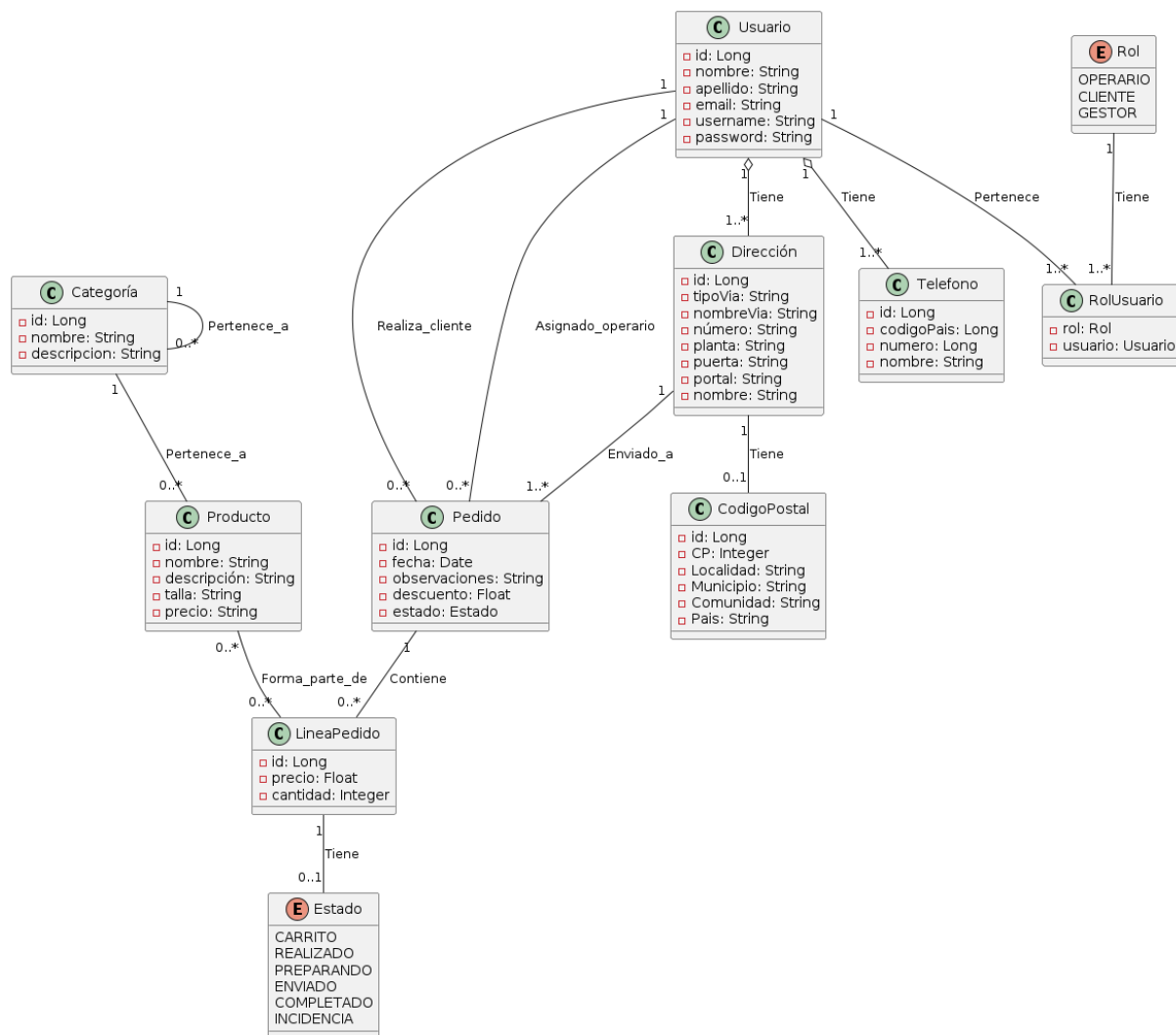
- **Usuario:** Representa un usuario del sistema con atributos como `id`, `nombre`, `apellido`, `email`, `username` y `password`.
- **Producto:** Representa un producto con atributos como `id`, `nombre`, `descripción`, `talla` y `precio`.
- **Pedido:** Representa un pedido realizado por un usuario con atributos como `id`, `fecha`, `observaciones`, `descuento` y `estado`.
- **LineaPedido:** Representa una línea individual dentro de un pedido con atributos como `id`, `precio`, `cantidad` y una referencia al producto.
- **Categoría:** Representa una categoría a la que pueden pertenecer los productos, con atributos como `id`, `nombre`, `descripción` y una referencia opcional al padre de la categoría.
- **Dirección:** Representa la dirección de un usuario con atributos como `id`, `tipoVia`, `nombreVia`, `número`, `planta`, `puerta`, `portal`, `nombre` y referencias opcionales al código postal y al usuario propietario.
- **Telefono:** Representa un teléfono asociado a un usuario con atributos como `id`, `códigoPaís`, `numero`, `nombre` y una referencia al usuario propietario.
- **Rol:** Enumeración que define los roles posibles de los usuarios: `OPERARIO`, `CLIENTE` y `GESTOR`.
- **RolUsuario:** Relaciona un usuario con su rol correspondiente.

2. Relaciones:

- **Usuario** tiene una relación de uno a muchos con **Pedido**, indicando que un usuario puede realizar varios pedidos.
- **Usuario** también tiene relaciones uno a uno con **Dirección** y **Telefono**, indicando que un usuario puede tener una dirección y varios teléfonos.

- **Usuario** tiene una relación de uno a muchos con **RolUsuario**, lo que significa que un usuario puede tener varios roles en el sistema.
- **Rol** tiene una relación uno a uno con **RolUsuario**, lo que significa que un rol está asociado a un único usuario.
- **Pedido** tiene una relación de uno a muchos con **LineaPedido**, lo que significa que un pedido puede contener múltiples líneas de pedido.
- **Pedido** también tiene una relación de muchos a uno con **Usuario**, indicando que un pedido puede tener un usuario asignado como operario.
- **Producto** tiene una relación de muchos a muchos con **LineaPedido**, lo que significa que un producto puede estar presente en varias líneas de pedidos y una línea de pedido puede contener varios productos.
- **Categoría** tiene una relación de uno a muchos con **Producto**, lo que significa que una categoría puede contener múltiples productos.
- **Categoría** tiene una relación de uno a muchos consigo misma, lo que indica que una categoría puede tener subcategorías.
- **Dirección** tiene una relación opcional de uno a uno con **CodigoPostal**, lo que indica que una dirección puede tener asociado un código postal.
- **LineaPedido** tiene una relación opcional de uno a uno con **Estado**, lo que indica que una línea de pedido puede tener un estado asociado.

Este diagrama describe la estructura del sistema de gestión de pedidos y productos, así como las relaciones entre las diferentes entidades que componen el sistema. A continuación vemos una representación gráfica del mismo (si tienes en VS Code el fichero `.puml` con el código anterior y pulsas ALT+d se genera esto):

**Figura 14:** Diagrama de clases

Fíjate cómo almacenamos el precio de los productos en la entidad `lineaPedido` por si varía con el tiempo. Podríamos haber usado una entidad `precioProducto` que, según las fechas del pedido, nos de el precio del producto, una posibilidad muy usada también y con menos redundancia de datos. No obstante, esta opción debemos usarla sólo con un disparador que impida modificar precios de un tiempo pasado, pues podría dar lugar a incongruencias en los pedidos pasados o aún en proceso. Hoy día el almacenamiento es muy económico pero sin embargo, el tiempo de CPU y/o consultas de disco están sujetos a costes adicionales en servidores en la nube (AWS, Azure, etc.).

Ahora que hemos terminado de modelar nuestro proceso de negocio en los casos de uso y conocemos las entidades gracias al modelado de clases, pasamos a la siguiente fase: creación de los POJOs (a

partir de del diagrama de clases) e identificación de los servicios y end-points de la aplicación Web.

7 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.

Para facilitar el trabajo usaremos anotaciones de Lombok y JPA.

Anotaciones Lombok:

- `@Data`: Genera todos los getters y setters, `toString`, `hashCode` y `compare`. Genera el constructor con todos los atributos.
- `@NoArgsConstructor`: Genera el constructor vacío que hace falta para JAXB, por ejemplo.

Anotaciones `jakarta.persistence` (antiguamente `javax.persistence`):

- `@Entity`: El objeto (clase) marcado con esta anotación será una tabla en la base de datos.
- `@Id`: El atributo marcado con esta anotación será la clave primaria de la tabla correspondiente al objeto o clase.
- `@ManyToOne`: Marcamos así atributos que son a su vez entidades (clases modelo) con los que tenemos una relación de muchos a uno. Lo ponemos en atributos que son a su vez clases entidad que yo he definido (para indicar clave foránea)
- `@ManyToOne`: Igual que anterior pero cuando puede ser NULL la clave foránea referenciada
- `@ManyToMany`: Muchos a muchos, se creará una entidad intermedia como cuando en un modelo entidad-relación tenemos una relación de muchos a muchos.
- `@OneToMany`: Marcamos así atributos que son a su vez entidades (clases modelo) con los que tenemos una relación de uno a muchos (será una lista o similar este atributo).
- `@OneToOne`: Relación de uno-a-uno.
- `@GeneratedValue(strategy = GenerationType.IDENTITY)`: Unido a `@Id`, será el `AUTO_INCREMENT` de MySQL (en la tabla asociada a ese objeto).
- `@Column(length = 25)`: En un atributo de una clase entidad fija la longitud del VARCHAR, por ejemplo.

Veamos la siguiente clase 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 }
```

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.

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.

7.1 Los POJOs del proyecto

A continuación vamos a ver cómo hemos generado cada una de las clases modelo del proyecto de la zapatería a partir del diagrama de clases del apartado anterior.

7.1.1 Categoría

Esta entidad nos servirá para gestionar las categorías de calzado:

```
1 @Entity
2 @Data
3 @NoArgsConstructor
4 public class Categoria {
5     @Id
```

```
6     @GeneratedValue(strategy = GenerationType.IDENTITY)
7     private Long id;
8     private String nombre;
9     private String descripcion;
10    @ManyToOne
11    private Categoria padre;
12 }
```

7.1.2 Código Postal

Esta clase sirve para los códigos postales de las ciudades (para dirección). Puedes descargar un listado de todos los códigos postales en este repositorio de Github.

```
1 @Entity
2 @Data
3 @NoArgsConstructor
4 public class CodigoPostal {
5     @Id
6     @GeneratedValue(strategy = GenerationType.IDENTITY)
7     private Long id;
8     private Integer CP;
9     private String localidad;
10    private String municipio;
11    private String comunidad;
12    private String pais;
13 }
```

7.1.3 Dirección

Entidad para gestionar las direcciones de un usuario de la aplicación:

```
1 @Entity
2 @Data
3 @NoArgsConstructor
4 public class Direccion {
5     @Id
6     @GeneratedValue(strategy = GenerationType.IDENTITY)
7     private Long id;
8     private String tipoVia;
9     private String nombreVia;
10    private String número;
11    private String planta;
12    private String puerta;
13    private String portal;
14    private String nombre;
15    @ManyToOne
16    private CodigoPostal codpos;
```



```
17     @ManyToOne
18     private Usuario usuario;
19 }
```

7.1.4 Estado

Estados posibles por los que pasa un pedido:

```
1 public enum Estado {
2     CARRITO, REALIZADO, PREPARANDO, ENVIADO, COMPLETADO, INCIDENCIA
3 }
```

7.1.5 Línea Pedido

En un pedido tenemos varios productos (con la cantidad de ese producto y su precio en el momento de la compra). A cada una de estas entidades las llamamos “línea pedido”.

```
1 @Entity
2 @Data
3 @NoArgsConstructor
4 public class LineaPedido {
5     @Id
6     @GeneratedValue(strategy = GenerationType.IDENTITY)
7     private Long id;
8     private Float precio;
9     private Integer cantidad;
10    @ManyToOne
11    private Producto producto;
12    @ManyToOne
13    private Pedido pedido;
14 }
```

7.1.6 Pedido

Hay que destacar en los pedidos que un carro de la compra es un pedido con estado “CARRITO”. La lógica de la aplicación sólo debe permitir un carro de la compra por usuario, es decir, un pedido en estado “CARRITO”. Por seguridad esto se puede controlar también con un disparador, de manera que por cada usuario

```
1 @Entity
2 @Data
3 @NoArgsConstructor
4 public class Pedido {
5     @Id
```

```
6     @GeneratedValue(strategy = GenerationType.IDENTITY)
7     private Long id;
8     private LocalDate fecha;
9     private String observaciones;
10    private Float descuento;
11    @Enumerated(EnumType.STRING)
12    private Estado estado;
13    @OneToMany(mappedBy = "pedido")
14    private List<LineaPedido> lineaPedidos;
15    @ManyToOne
16    private Usuario asignadoOperario;
17    @ManyToOne
18    private Direccion direccion;
19 }
```

7.1.7 Producto

La entidad producto nos ayuda a gestionar la información de los zapatos de la base de datos:

```
1 @Entity
2 @Data
3 @NoArgsConstructor
4 public class Producto {
5     @Id
6     @GeneratedValue(strategy = GenerationType.IDENTITY)
7     private Long id;
8     private String nombre;
9     private String descripción;
10    private String talla;
11    private Float precio;
12    @ManyToOne
13    private Categoria categoria;
14 }
```

7.1.8 Rol

Listado de roles posibles para los usuarios:

```
1 public enum Rol {
2     OPERARIO, CLIENTE, GESTOR
3 }
```

7.1.9 Rol Usuario

Entidad para asignar roles a los usuarios:

```
1 @Entity
2 @Data
3 @NoArgsConstructor
4 public class RolUsuario {
5     @Id
6     @GeneratedValue(strategy = GenerationType.IDENTITY)
7     private long id;
8     @Enumerated(EnumType.STRING)
9     private Rol rol;
10    @ManyToOne
11    private Usuario usuario;
12 }
```

7.1.10 Teléfono

Entidad para los teléfonos de cada usuario:

```
1 @Entity
2 @Data
3 @NoArgsConstructor
4 public class Telefono {
5     @Id
6     @GeneratedValue(strategy = GenerationType.IDENTITY)
7     private Long id;
8     private Long codigoPais;
9     private Long numero;
10    private String nombre;
11    @ManyToOne
12    private Usuario usuario;
13 }
```

7.1.11 Usuario

Entidad para gestión de usuarios de la aplicación.

```
1 @Entity
2 @Data
3 @NoArgsConstructor
4 public class Usuario {
5     @Id
6     @GeneratedValue(strategy = GenerationType.IDENTITY)
7     private Long id;
8     private String nombre;
9     private String apellido;
10    private String email;
11    private String username;
```

```
12     private String password;  
13  
14     @OneToMany(mappedBy = "usuario")  
15     private List<Direccion> direcciones;  
16     @OneToMany(mappedBy = "usuario")  
17     private List<Telefono> telefonos;  
18     @OneToMany(mappedBy = "usuario")  
19     private List<RolUsuario> roles;  
20  
21 }
```

8 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.

Este archivo de configuración se utiliza fundamentalmente para indicar 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.

A continuación vamos a ver cómo definir los repositorios. Esto es sólo la definición, para ver cómo usarlos puedes echar un ojo al siguiente apartado: **Controladores**.

8.1 RepoCategoria

Cada categoría viene identificada por un código numérico único, un nombre y una descripción. Además cada categoría tiene una (y sólo una) categoría padre. Para preguntar a una categoría por sus hijos usaremos su repositorio.

```
1 package com.iesvdc.acceso.zapateria.zapapp.repositorios;
2
3 import java.util.List;
4
5 import org.springframework.data.jpa.repository.JpaRepository;
6 import org.springframework.data.jpa.repository.Query;
7 import org.springframework.stereotype.Repository;
8
9 import com.iesvdc.acceso.zapateria.zapapp.modelos.Categoria;
10
11 @Repository
12 public interface RepoCategoria extends JpaRepository<Categoria, Long> {
13     @Query("SELECT c FROM Categoria c WHERE c.padre = :padre")
14     List<Categoria> findByPadre(Categoria padre);
15 }
```

Observa la consulta `@Query("SELECT c FROM Categoria c WHERE c.padre = :padre")`, en esta consulta JPQL:

- `SELECT c`: Indica que queremos seleccionar entidades `Categoria`.
- `FROM Categoria c`: Especifica la entidad de la que queremos seleccionar, que es `Categoria` y la abreviamos como `c`.
- `WHERE c.padre = :padre`: Filtra las categorías basadas en su atributo `padre`, que debe ser igual al objeto `Categoria` proporcionado como parámetro `padre`.

Con esta consulta JPQL personalizada, obtenemos todas las categorías que tienen el mismo padre que el objeto `Categoria` proporcionado.

No obstante este código es redundante pues Spring tiene la capacidad de generar por nosotros cualquier **"findBy"**, en verdad basta con hacer:

```
1 package com.iesvdc.acceso.zapateria.zapapp.repositorios;
2
3 import java.util.List;
4
5 import org.springframework.data.jpa.repository.JpaRepository;
6 import org.springframework.stereotype.Repository;
7
8 import com.iesvdc.acceso.zapateria.zapapp.modelos.Categoria;
9
10 @Repository
11 public interface RepoCategoria extends JpaRepository<Categoria, Long> {
```

```
12     List<Categoria> findByPadre(Categoria padre);  
13 }
```

8.2 RepoCodigoPostal

Los códigos postales que tenemos en la base de datos son accesibles mediante este repositorio:

```
1 @Repository  
2 public interface RepoCodigoPostal extends JpaRepository <CodigoPostal,  
   Long> {  
3  
4 }
```

8.3 RepoDireccion

```
1 @Repository  
2 public interface RepoDireccion extends JpaRepository<Direccion, Long> {  
3  
4 }
```

8.4 RepoLineaPedido

```
1 @Repository  
2 public interface RepoLineaPedido extends JpaRepository<LineaPedido,  
   Long> {  
3  
4 }
```

8.5 RepoPedido

```
1 @Repository  
2 public interface RepoProducto extends JpaRepository<Producto, Long> {  
3  
4 }
```

8.6 RepoProducto

```
1 @Repository  
2 public interface RepoProducto extends JpaRepository<Producto, Long> {  
3     List<Producto> findByCategoria(Categoria padre);
```



```
4 }
```

8.7 RepoRolUsuario

```
1 @Repository
2 public interface RepoRolUsuario extends JpaRepository<RolUsuario, Long>
3     {
4 }
```

8.8 RepoTelefono

```
1 @Repository
2 public interface RepoTelefono extends JpaRepository<Telefono, Long> {
3
4 }
```

8.9 RepoUsuario

```
1 @Repository
2 public interface RepoUsuario extends JpaRepository<Usuario, Long> {
3
4 }
```

9 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 peticiones:** 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.

Los controladores devuelven una cadena de caracteres que es el nombre del archivo de la vista, es decir si devuelvo la cadena "index" quiere decir que en la carpeta **resources/templates** existe un archivo **index.html** que será el que se vea en el navegador al hacer la petición **/index**.

9.1 Listado de rutas de nuestra aplicación

Para completar la aplicación, se definen una serie de end-points que serán completados en sucesivos sprints (un sprint en la metodología SCRUM de trabajo es una tarea que al completarla obtenemos un subproducto funcional).

9.1.1 Servicio usuario

RUTA	VERBO	DATOS	COMENTARIOS
/usuario/telefonos	GET	nada	Muestra los tel. del usuario que ha hecho login
/usuario/telefonos/add	GET	nada	Muestra formulario para añadir tel. al usuario que ha hecho login
/usuario/telefonos/add	POST	body (tel)	Añade tel. al usuario que ha hecho login
/usuario/telefonos/delete/{GET_id}	GET	nada	Muestra formulario para borrar tel. al usuario que ha hecho login
/usuario/telefonos/delete/{POST_id}	POST	body (tel)	Borra tel. al usuario que ha hecho login
/usuario/telefonos/update/{GET_id}	GET	nada	Muestra formulario para editar el tel. del usuario que ha hecho login
/usuario/telefonos/update/{POST_id}	POST	body (tel)	Edita tel. del usuario que ha hecho login

9.1.2 Servicio producto

Rutas del servicio para el rol GESTOR:

RUTA	METODO	ROL	Observaciones
/admin/producto	GET	gestor	Mostrar listado productos
/admin/producto/create	GET	gestor	Mostrar formulario alta productos
/admin/producto/create	POST	gestor	Crear en la BBDD el producto
/admin/producto/update	GET	gestor	Mostrar formulario modificación productos
/admin/producto/update	POST	gestor	Modificar en la BBDD el producto
/admin/producto/delete	GET	gestor	Mostrar formulario borrar productos

RUTA	METODO	ROL	Observaciones
/admin/producto/delete	POST	gestor	Borrar en la BBDD el producto
/admin/categoria	GET	gestor	Mostrar listado de categorías
/admin/categoria/create	GET	gestor	Mostrar formulario alta categoria
/admin/categoria/create	POST	gestor	Crear reserva en la BBDD
/admin/categoria/delete	GET	gestor	Mostrar formulario borrar categoria
/admin/categoria/delete	POST	gestor	Borra categoría de la BBDD
/admin/categoria/:id/productos	GET	gestor	Mostrar los productos de una categoría
/admin/categoria/producto/:id	POST	gestor	Mostrar maestro-detalle de categoria para un usuario
/admin/categoria/producto/:id	GET	gestor	Mostrar categoria para ese producto
/admin/categoria/producto/:id	POST	gestor	Mostrar maestro-detalle de categoria para una categoría

Para saber cómo añadir seguridad a estos endpoints, es decir, que sólo usuarios con el rol “GESTOR” puedan verlos, avanza al apartado de **“Seguridad”**.

Para saber cómo hacer las vistas gestionadas por estos controladores, avanza a **“Vistas”**.

Archivo ControCategoria.java

```

1  @Controller
2  @RequestMapping("/admin")
3  public class ControCategoria {
4
5      @Autowired
6      RepoCategoria repoCategoria;
7
8      @GetMapping("categoria")
9      public String findAll(Model model) {
10         model.addAttribute("categorias", repoCategoria.findAll());
11         return "admin/categorias";
12     }
13
14     @GetMapping("categoria/hijos/{id}")
15     public String findChilds(
16         Model model,
17         @PathVariable(name = "id") Long id) {
18
19         Optional<Categoria> oCategoria = repoCategoria.findById(id);
20     }

```

```
21         if(oCategoria.isPresent()) {
22             Categoria padre = oCategoria.get();
23             model.addAttribute("categorias", repoCategoria.findByPadre(
24                 padre));
25             return "admin/categorias";
26         } else {
27             model.addAttribute("titulo", "Categoria: ERROR");
28             model.addAttribute("mensaje", "No puedo encontrar esa
29                 categoría en la base de datos");
30             return "error";
31         }
32     }
33     @GetMapping("categoria/add")
34     public String addForm(Model modelo) {
35         modelo.addAttribute("categorias", repoCategoria.findAll());
36         modelo.addAttribute("categoria", new Categoria());
37         return "admin/categorias-add";
38     }
39
40     @PostMapping("categoria/add")
41     public String postMethodName(
42         @ModelAttribute("categoria") Categoria categoria) {
43         repoCategoria.save(categoria);
44         return "redirect:/admin/categoria";
45     }
46
47     @GetMapping("categoria/delete/{id}")
48     public String deleteForm(
49         @PathVariable(name = "id") @NonNull Long id,
50         Model modelo) {
51         try {
52             Optional<Categoria> categoria = repoCategoria.findById(id);
53             if (categoria.isPresent()){
54                 // si existe la categoria
55                 modelo.addAttribute(
56                     "categoria", categoria.get());
57                 return "admin/categorias-del";
58             } else {
59                 return "error";
60             }
61         } catch (Exception e) {
62             return "error";
63         }
64     }
65 }
66
67
68 @PostMapping("categoria/delete/{id}")
69 public String delete(
```

```
70         @PathVariable("id") @NonNull Long id) {
71     try {
72         repoCategoria.deleteById(id);
73     } catch (Exception e) {
74         return "error";
75     }
76
77     return "redirect:/admin/categoria";
78 }
79
80
81 @GetMapping("categoria/edit/{id}")
82 public String editForm(
83     @PathVariable @NonNull Long id,
84     Model modelo) {
85
86     Optional<Categoria> categoria =
87         repoCategoria.findById(id);
88     List<Categoria> categorias =
89         repoCategoria.findAll();
90
91     if (categoria.isPresent()){
92         modelo.addAttribute("categoria", categoria.get());
93         modelo.addAttribute("categorias", categorias);
94         return "admin/categorias-edit";
95     } else {
96         modelo.addAttribute(
97             "mensaje",
98             "Categoria no encontrada");
99         modelo.addAttribute(
100             "titulo",
101             "Error en categorías.");
102         return "error";
103     }
104 }
105
106 }
```

Este controlador Spring MVC maneja las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) para la entidad `Categoria` y en él tenemos:

1. Anotaciones en el controlador:

- `@Controller`: Indica que esta clase es un controlador de Spring MVC.
- `@RequestMapping("/admin")`: Especifica el prefijo de la URL para todas las solicitudes manejadas por este controlador.

2. Inyección de dependencias:

- `@Autowired RepoCategoria repoCategoria`; Inyecta automáticamente una

instancia de `RepoCategoria` en este controlador. `RepoCategoria` es un repositorio de Spring Data JPA que se utiliza para interactuar con la base de datos para la entidad `Categoria`.

3. Método `findAll`:

- `@GetMapping("categoria")`: Maneja las solicitudes GET en la URL `/admin/categoria`.
- `repoCategoria.findAll()`: Recupera todas las categorías de la base de datos y las agrega al modelo.
- Retorna la vista `admin/categorias`.

4. Método `findChilds`:

- `@GetMapping("categoria/hijos/{id}")`: Maneja las solicitudes GET en la URL `/admin/categoria/hijos/{id}` donde `{id}` es el ID de la categoría padre.
- `repoCategoria.findById(id)`: Busca la categoría padre por su ID.
- `repoCategoria.findByPadre(padre)`: Recupera todas las categorías hijas de la categoría padre y las agrega al modelo.
- Retorna la vista `admin/categorias`.
- Si no se encuentra la categoría padre, retorna una vista de error.

5. Métodos `addForm` y `postMethodName`:

- `addForm` maneja las solicitudes GET para mostrar el formulario de agregar una nueva categoría.
- `postMethodName` maneja las solicitudes POST para procesar el formulario de agregar una nueva categoría.
- `@ModelAttribute("categoria") Categoria categoria`: Enlaza los datos enviados desde el formulario a un objeto `Categoria` automáticamente.

6. Métodos `deleteForm` y `delete`:

- `deleteForm` maneja las solicitudes GET para mostrar el formulario de confirmación de eliminación de una categoría.
- `delete` maneja las solicitudes POST para eliminar una categoría por su ID.
- Si la categoría no se encuentra o si ocurre un error durante la eliminación, se muestra una vista de error.

7. Método `editForm`:

- `editForm` maneja las solicitudes GET para mostrar el formulario de edición de una categoría.

- Si la categoría no se encuentra, se muestra una vista de error.

Fíjate que no hay un método ni ruta para el POST del formulario editar. Resulta que si una categoría existe en la base de datos se actualizará y si no existe, se crea una nueva. Esto lo sabemos si el **id** del categoría tiene un valor o si es **null** se trata de una categoría nueva.

Archivo **ControProducto.java**

```
1 @Controller
2 @RequestMapping("/admin")
3 public class ControProducto {
4
5     @Autowired
6     RepoCategoria repoCategoria;
7     @Autowired
8     RepoProducto repoProducto;
9
10
11     @GetMapping("producto")
12     public String findAll(Model model) {
13         model.addAttribute("productos", repoProducto.findAll());
14         return "admin/productos";
15     }
16
17     @GetMapping("producto/categoria/{id}")
18     public String findByCategoria(
19         Model model,
20         @PathVariable(name = "id") Long id) {
21
22         Optional<Categoria> oCategoria = repoCategoria.findById(id);
23
24         if(oCategoria.isPresent()) {
25             Categoria padre = oCategoria.get();
26             List<Categoria> lCategorias = repoCategoria.findAll();
27             model.addAttribute("productos", repoProducto.
28                 findByCategoria(padre));
29             model.addAttribute("categorias", lCategorias);
30             model.addAttribute("categoria", padre);
31             return "admin/productos-cat";
32         } else {
33             model.addAttribute("titulo", "Producto: ERROR");
34             model.addAttribute("mensaje", "No puedo encontrar esa
35                 categoría en la base de datos");
36             return "error";
37         }
38
39     @GetMapping("producto/categoria")
40     public String findByCategorias(Model model) {
41
```



```
42     List <Categoria> lCategorias = repoCategoria.findAll();
43     model.addAttribute("productos", repoProducto.findAll());
44     model.addAttribute("categorias", lCategorias);
45
46     return "admin/productos-cat";
47 }
48
49 @GetMapping("producto/add")
50 public String addForm(Model modelo) {
51     modelo.addAttribute("productos", repoProducto.findAll());
52     modelo.addAttribute("producto", new Producto());
53     modelo.addAttribute("categorias", repoCategoria.findAll());
54     return "admin/productos-add";
55 }
56
57 @PostMapping("producto/add")
58 public String postMethodName(
59     @ModelAttribute("producto") Producto producto) {
60     repoProducto.save(producto);
61     return "redirect:/admin/producto";
62 }
63
64 @GetMapping("producto/delete/{id}")
65 public String deleteForm(
66     @PathVariable(name = "id") @NonNull Long id,
67     Model modelo) {
68     try {
69         Optional<Producto> producto = repoProducto.findById(id);
70         if (producto.isPresent()){
71             // si existe la producto
72             modelo.addAttribute(
73                 "producto", producto.get());
74             return "admin/productos-del";
75         } else {
76             return "error";
77         }
78     } catch (Exception e) {
79         return "error";
80     }
81 }
82
83
84
85 @PostMapping("producto/delete/{id}")
86 public String delete(
87     @PathVariable("id") @NonNull Long id) {
88     try {
89         repoProducto.deleteById(id);
90     } catch (Exception e) {
91         return "error";
92     }
93 }
```

```
93
94     return "redirect:/admin/producto";
95 }
96
97
98 @GetMapping("producto/edit/{id}")
99 public String editForm(
100     @PathVariable @NonNull Long id,
101     Model modelo) {
102
103     Optional<Producto> producto =
104         repoProducto.findById(id);
105     List<Producto> productos =
106         repoProducto.findAll();
107
108     if (producto.isPresent()){
109         modelo.addAttribute("producto", producto.get());
110         modelo.addAttribute("productos", productos);
111         return "admin/productos-edit";
112     } else {
113         modelo.addAttribute(
114             "mensaje",
115             "Producto no encontrada");
116         modelo.addAttribute(
117             "titulo",
118             "Error en productos.");
119         return "error";
120     }
121 }
122
123 }
```

Este otro controlador Spring MVC maneja las operaciones CRUD para la entidad `Producto`:

1. Anotaciones en el controlador:

- `@Controller`: Indica que esta clase es un controlador de Spring MVC.
- `@RequestMapping("/admin")`: Especifica el prefijo de la URL para todas las solicitudes manejadas por este controlador.

2. Inyección de dependencias:

- `@Autowired RepoCategoria repoCategoria`;: Inyecta automáticamente una instancia de `RepoCategoria` en este controlador. `RepoCategoria` es un repositorio de Spring Data JPA que se utiliza para interactuar con la base de datos para la entidad `Categoria`.
- `@Autowired RepoProducto repoProducto`;: Inyecta automáticamente una instancia de `RepoProducto` en este controlador. `RepoProducto` es un repositorio

de Spring Data JPA que se utiliza para interactuar con la base de datos para la entidad `Producto`.

3. Método `findAll`:

- `@GetMapping("producto")`: Maneja las solicitudes GET en la URL `/admin/producto`.
- `repoProducto.findAll()`: Recupera todos los productos de la base de datos y los agrega al modelo.
- Retorna la vista `admin/productos`.

4. Método `findByCategoria`:

- `@GetMapping("producto/categoria/{id}")`: Maneja las solicitudes GET en la URL `/admin/producto/categoria/{id}` donde `{id}` es el ID de la categoría.
- `repoCategoria.findById(id)`: Busca la categoría por su ID.
- `repoProducto.findByCategoria(padre)`: Recupera todos los productos asociados a la categoría y los agrega al modelo.
- Retorna la vista `admin/productos-cat`.
- Si la categoría no se encuentra, muestra una vista de error.

5. Método `findByCategorias`:

- `@GetMapping("producto/categoria")`: Maneja las solicitudes GET en la URL `/admin/producto/categoria`.
- `repoCategoria.findAll()`: Recupera todas las categorías de la base de datos y las agrega al modelo.
- Retorna la vista `admin/productos-cat` con todos los productos y categorías.

6. Métodos `addForm` y `postMethodName`:

- `addForm` maneja las solicitudes GET para mostrar el formulario de agregar un nuevo producto.
- `postMethodName` maneja las solicitudes POST para procesar el formulario de agregar un nuevo producto.
- `@ModelAttribute("producto") Producto producto`: Enlaza los datos enviados desde el formulario a un objeto `Producto` automáticamente.

7. Métodos `deleteForm` y `delete`:

- `deleteForm` maneja las solicitudes GET para mostrar el formulario de confirmación de eliminación de un producto.
- `delete` maneja las solicitudes POST para eliminar un producto por su ID.

- Si el producto no se encuentra o si ocurre un error durante la eliminación, muestra una vista de error.

8. Método `editForm`:

- `editForm` maneja las solicitudes GET para mostrar el formulario de edición de un producto.
- Si el producto no se encuentra, muestra una vista de error.

Fíjate que no hay un método ni ruta para el POST del formulario editar. Resulta que si un producto existe en la base de datos se actualizará y si no existe, se crea uno nuevo. Esto lo sabemos si el `id` del producto está instanciado o si es `null` se trata de un producto nuevo.

9.1.3 Servicio gestionar “mis pedidos”

Un cliente puede gestionar (ver) el listado de pedidos así como el detalle de los mismos. Deberemos implementar un listado y un maestro-detalle de pedidos.

RUTA	METODO	ROL	Observaciones
/mis-pedidos	GET	cliente	Mostrar listado pedidos
/mis-pedidos/detalle/{id}	GET	cliente	Mostrar detalle del pedido con ese ID
/mis-pedidos/detalle	POST	cliente	Mostrar detalle del pedido con el ID que se pasa como parámetro

9.1.4 Servicio envío (estados) pedidos

El operario gestiona los pedidos. Ve el listado de pedidos sin procesar y en el momento que cambia un pedido a “en proceso” se le asigna y será el encargado de enviarlo. Igualmente también debe tener un listado de pedidos “en preparación”, “enviados” y “completados”. Tendremos un maestro-detalle en función del estado y del operario.

RUTA	METODO	Datos	Observaciones
/pedidos	GET	nada	Listado de pedidos en estado

“REALIZADO” /pedidos/{id} | GET | ID del pedido | Formulario ¿desea servir este pedido? /pedidos/{id} | POST | ID del pedido | Se le asigna al operario que hizo login ese pedido /pedidos/estado/{estado} | GET | estado | Listado de pedidos en ese estado para el operario que ha hecho LOGIN /pedidos/operario/{idOpe} | GET | ID operario y estado | Listado de pedidos en ese estado para ese operario /pedidos/operario/{idOpe}/estado/{estado} | GET | ID operario y estado | Listado de pedidos en ese estado para ese operario

9.1.5 Servicio carro de la compra

Un usuario con perfil cliente gestiona su carro de la compra:

RUTA	METODO	Datos	Observaciones
/carro	GET	Nada	Ver la cesta de la compra
/carro/add/{id}	GET	ID de producto y cantidad	Formulario para añadir productos al carro
/carro/add/{id}	POST	ID de producto y cantidad	Añade realmente el producto al carro
/productos	GET	nada	Listado de productos de la tienda
/productos/{id}	GET	El ID del producto	Formulario para añadir el producto con ese ID a la cesta (post a /carro/add)
/carro/delete	GET	Nada	Formulario vaciar cesta de la compra (¿seguro?)
/carro/delete	POST	Nada	Vacía la cesta de la compra
/carro/delete/{id}	GET	ID de linea producto	Formulario eliminar de la cesta de la compra un producto (¿seguro?)
/carro/delete/{id}	POST	ID de linea producto	Eliminar de la cesta de la compra un producto
/carro/confirm	GET	nada	Formulario confirmar compra
/carro/confirm	POST	nada	Confirma compra

Antes de comenzar, al tratarse de un proceso de compra, que es individual e intransferible, vamos a necesitar un método en el controlador que nos diga el usuario que ha iniciado la sesión, para ello del contexto de la aplicación podemos obtener información de la autenticación:

```
1  /**
2   * Este método obtiene, del contexto de la aplicación, información
3   * sobre la autenticación.
4   * Devuelve un objeto de tipo Usuario que es además quien ha
5   * entrado en la aplicación.
6   * @return Usuario
7   */
8  private Usuario getLoggedUser() {
9      // Del contexto de la aplicación obtenemos el usuario
10     Authentication authentication = SecurityContextHolder.
11         getContext().getAuthentication();
12     String username = authentication.getName();
13     // obtenemos el usuario del repositorio por su "username"
14     Usuario cliente = repoUsuario.findByUsername(username).get(0);
15
16     return cliente;
17 }
```

Seguidamente vamos a explicar qué debe hacer cada endpoint así como la lógica asociada que lo resuelve.

Endpoint /carro

Muestra el carro de la compra para el usuario que hizo login.

Un carro de la compra es un pedido en un estado especial que aún no tiene fecha, ni hora, ni otros datos.

Este controlador añade los datos del carro de la compra a la vista.

```
1  /**
2   * Este método muestra el carro de la compra para el usuario que
3   * hizo login.
4   * Un carro de la compra es un pedido en un estado especial que aún
5   * no tiene
6   * fecha, ni hora, ni otros datos.
7   * Este controlador añade los datos del carro de la compra a la
8   * vista.
9   * @param modelo
10  * @return el carro de la compra con los productos que se hayan
11  *         metido (si existen)
12  */
13  @GetMapping("/carro")
14  public String findCarro(Model modelo) {
15      List<LineaPedido> lineaPedidos = null;
16  }
```

```
14     Usuario cliente = getLoggedUser();
15
16     long total = 0;
17
18     // Para el usuario que hizo login, buscamos un pedido (sólo
19     // puede haber uno) en estado "CARRITO"
20     List<Pedido> pedidos = repoPedido.findByEstadoAndCliente(Estado
21     .CARRITO, cliente);
22     if (pedidos.size()>0) {
23         lineaPedidos = repoLineaPedido.findByPedido(pedidos.get(0))
24         ;
25         for (LineaPedido lp : pedidos.get(0).getLineaPedidos()) {
26             total += lp.getCantidad()*lp.getProducto().getPrecio();
27         }
28     }
29
30     // mandamos a la vista los modelos: Pedido y su lista de
31     // LineaPedido
32     modelo.addAttribute("pedido", pedidos.size()>0 ? pedidos.get(0)
33     : new Pedido());
34     modelo.addAttribute("lineapedidos", lineaPedidos);
35     modelo.addAttribute("total", total);
36
37     // modelo.addAttribute("productos", productos );
38     return "carro/carro";
39 }
```

End-point /carro/edit/{id}

Para editar la cantidad de un calzado concreto que tenemos en el pedido tenemos este endpoint, que para un método GET muestra el formulario y para el método POST lo lleva a cabo.

En este end-point el ID hace referencia a un objeto de tipo *LineaPedido*, es decir el detalle de un producto concreto para un pedido en cuestión.

Por seguridad hay que comprobar que ese objeto *LineaPedido* pertenece a un *Pedido* del usuario que hizo login.

Para entenderlo vamos a explocar esta consulta para ver el detalle del pedido lineaPedido con ID=1 y el usuario con ID=2:

```
1 select username, pedido.id as "id_pedido", cantidad, producto.nombre
2 from usuario
3 inner join pedido on pedido.cliente_id = usuario.id
4 inner join linea_pedido on linea_pedido.pedido_id = pedido.id
5 inner join producto on linea_pedido.producto_id = producto.id
6 where usuario.id=2 and linea_pedido.id=1;
```

Que daría como resultado algo similar a esto (sólo una línea):

username	id_pedido	cantidad	nombre
cliente	1	2	Zapatillas Running Asics Gel-Nimbus

Endpoint `/carro/confirm`

En este método vemos dos métodos, GET o POST, en el GET nos mostrará un formulario de confirmación del pedido donde pondremos la dirección de envío y el teléfono de contacto para que el operador que prepare después el pedido pueda indicarlo al servicio de paquetería.

El método POST confirma el pedido, guarda los precios actuales en *LíneaPedido*, así como el total en *Pedido*. Igualmente le asocia al pedido la dirección de envío, el teléfono de contacto y pone el estado a *REALIZADO*.

9.2 Ejemplo de controlador sencillo

10 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.

10.1 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>
```

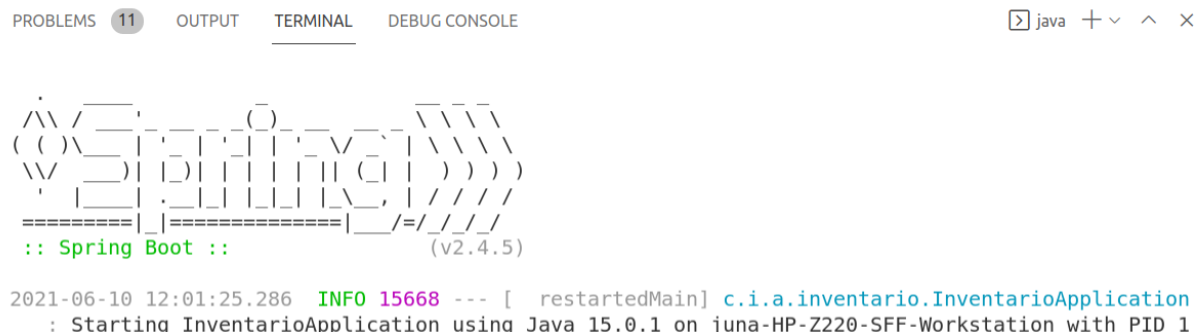
En este ejemplo, el fragmento con el nombre “nombreDelFragmento” definido en el archivo `fragmento.html` se inserta en la plantilla `otra plantilla.html` utilizando la directiva `th:insert`. El contenido del fragmento se renderizará en el lugar donde se inserta.

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.

11 Seguridad

Por defecto Spring incorpora su propio sistema de login que está habilitado simplemente al añadir la dependencia en el pom.xml.

El usuario por defecto es **user** y la contraseña la puedes ver diferente y generada en cada arranque de la aplicación:



```

PROBLEMS 11 OUTPUT TERMINAL DEBUG CONSOLE
[icon] java + ^ x

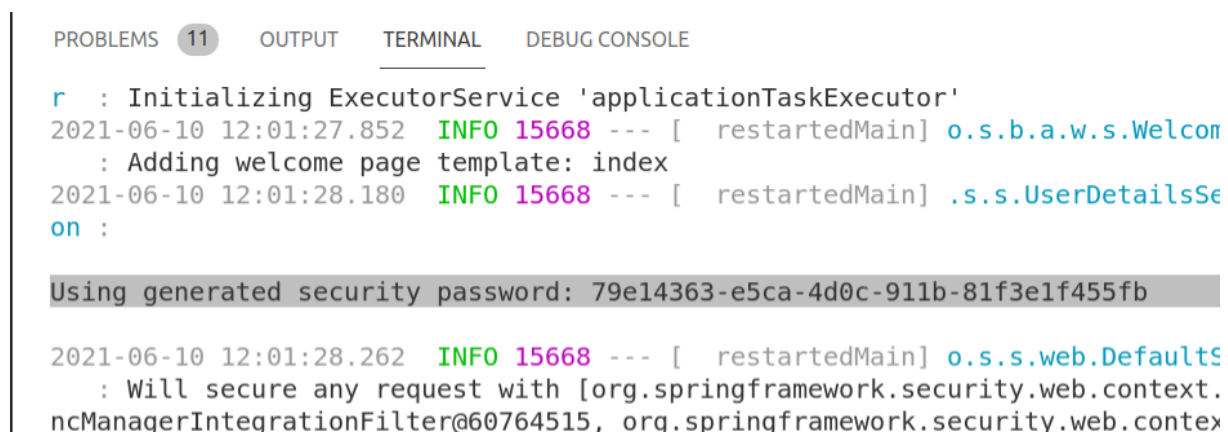
  ____ _
 / ___ \ | |
/ /   \ \| |
\ \   / \| |
 \___/____|_|
:: Spring Boot :: (v2.4.5)

2021-06-10 12:01:25.286 INFO 15668 --- [ restartedMain] c.i.a.inventario.InventarioApplication
: Starting InventarioApplication using Java 15.0.1 on iuna-HP-Z220-SFF-Workstation with PID 1

```

Figura 15: Arranque Spring Boot

Así aparece el password:



```

PROBLEMS 11 OUTPUT TERMINAL DEBUG CONSOLE

r : Initializing ExecutorService 'applicationTaskExecutor'
2021-06-10 12:01:27.852 INFO 15668 --- [ restartedMain] o.s.b.a.w.s.Welcom
: Adding welcome page template: index
2021-06-10 12:01:28.180 INFO 15668 --- [ restartedMain] .s.s.UserDetailsSe
on :

Using generated security password: 79e14363-e5ca-4d0c-911b-81f3e1f455fb

2021-06-10 12:01:28.262 INFO 15668 --- [ restartedMain] o.s.s.web.DefaultS
: Will secure any request with [org.springframework.security.web.context.
ncManagerIntegrationFilter@60764515, org.springframework.security.web.context

```

Figura 16: Password Spring Boot

11.0.1 Spring Security

Las contraseñas deberán estar cifradas con BCrypt, puedes usar este ejemplo para hacer pruebas:

Plaint text password	Hashed Password
Secreto_123	\$2a10PMDcjYqXJxGsVInve1t9Jug2DkDDckvUDl8.vF4Dc6yg0FMjovsXO

Para dar seguridad a la aplicación podemos crear una clase de configuración donde inyectamos los *beans* encargados de la seguridad. Además sería recomendable crear nuestros formularios de login y actualización de nuestros datos.

Ejemplo de Bean de configuración (puede ser código o un archivo XML):

```

1  @Configuration
2  @EnableWebSecurity
3  public class SecurityConfiguration {
4
5      @Autowired
6      DataSource dataSource;
7
8      @Autowired
9      public void configure(AuthenticationManagerBuilder amb) throws
10         Exception {
11         amb.jdbcAuthentication()
12             .dataSource(dataSource)
13             .usersByUsernameQuery("select username, password, enabled "
14                 +
15                 "from usuario where username = ?")
16             .authoritiesByUsernameQuery("select u.username, r.rol as '
17                 authority' " +
18                 "from usuario u, rol_usuario r " +
19                 "where u.id=r.usuario_id and username = ?");
20     }
21
22     @Bean
23     BCryptPasswordEncoder passwordEncoder(){
24         return new BCryptPasswordEncoder();
25     }
26
27     @Bean
28     public SecurityFilterChain filter(HttpSecurity http) throws
29         Exception {
30
31         // Con Spring Security 6.2 y 7: usando Lambda DSL
32
33         return http
34             .authorizeHttpRequests((requests) -> requests
35                 .requestMatchers("/webjars/**", "/img
36                     /**", "/js/**", "/register/**", "/
37                     ayuda/**", "/login", "/denegado")
38                 .permitAll()

```

```

33         .requestMatchers("/admin/**", "/admin
34         /*/**" , "/admin/*//**")
35         //.authenticated()
36         .hasAuthority("GESTOR")
37         .requestMatchers("/pedidos/**", "/
38         pedidos/*/**", "/pedidos/*//**", "/
39         pedidos/*/*/*/**", "/pedidos
40         /*/*/*/*/**")
41         //.authenticated()
42         .hasAuthority("OPERARIO")
43         .requestMatchers("/mis-pedidos/**", "/
44         mis-pedidos/*/**",
45         "/productos/**", "/productos/*/**",
46         "/carro/**", "/carro/*/**")
47         //.authenticated()
48         .hasAuthority("CLIENTE")
49         // .anyRequest().permitAll()
50         // ).headers(headers -> headers
51         // .frameOptions(frameOptions ->
52         frameOptions
53         // .sameOrigin())
54         // ).sessionManagement((session) -> session
55         // .sessionCreationPolicy(
56         SessionCreationPolicy.STATELESS)
57         ).exceptionHandling((exception)-> exception.
58         accessDeniedPage("/denegado") )
59         .formLogin((formLogin) -> formLogin
60         // .loginPage("/login")
61         .permitAll()
62         ).rememberMe(
63         Customizer.withDefaults()
64         ).logout((logout) -> logout
65         .invalidateHttpSession(true)
66         .logoutSuccessUrl("/")
67         // .deleteCookies("JSESSIONID") // no
68         es necesario, JSESSIONID se hace por
69         defecto
70         .permitAll()
71         ).csrf((protection) -> protection
72         .disable()
73         // ).cors((protection)-> protection
74         // .disable()
75         ).build();
76     }
77 }

```

Aquí vemos cómo buscar en la base de datos el usuario, contraseña y rol. Fíjate como Spring nos obliga a tener un campo *enabled* para el usuario.

Para cada autoridad decimos qué rutas son accesibles. Aunque no lo vemos, estamos creando un filtro

para un servlet.

12 Bibliografía