Ecommerce con Spring Boot

Creacion de la BD: La base de datos se creara en MySQL Workbench y se llamara Ecommerce Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Creacion del Proyecto : la configuración inicial del proyecto Java para un sistema de ecommerce utilizando Spring Initializr, una herramienta oficial para generar proyectos Spring Boot preconfigurados.

**Project**

* **Maven**: Se ha seleccionado *Maven* como herramienta de construcción. Esta permite gestionar dependencias, compilar el proyecto y empaquetarlo.

**Language**

* **Java**: Se ha elegido Java como lenguaje principal del proyecto, en lugar de Kotlin o Groovy.

**Spring Boot Version**

* **3.5.4**: Es la versión estable seleccionada de Spring Boot, compatible con Java 17 y con mejoras respecto a versiones anteriores.

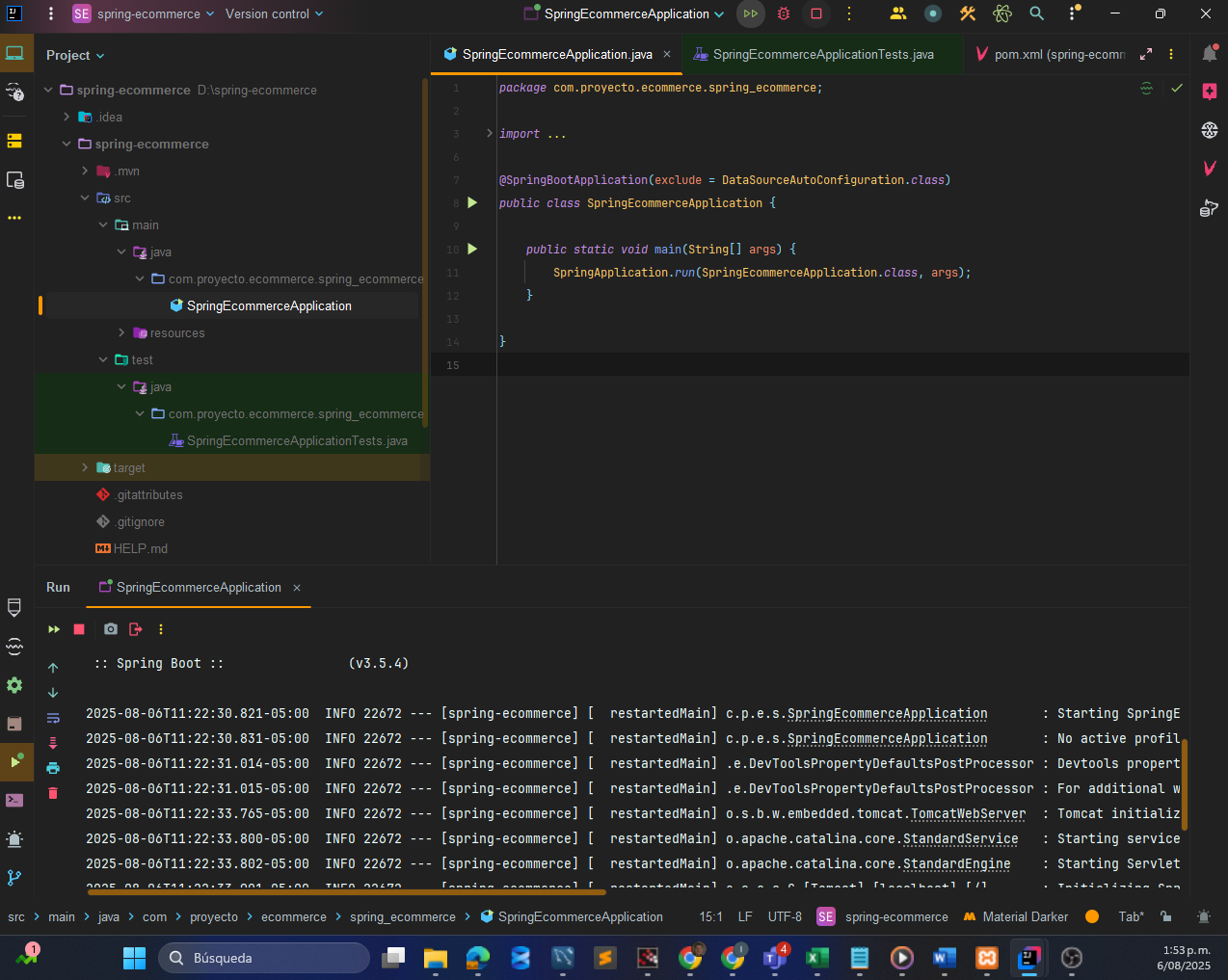
**Dependencias Incluidas**

Las siguientes librerías han sido seleccionadas al momento de generar el proyecto:

1. **Lombok (Developer Tools)**  
   Reduce el código repetitivo (boilerplate) mediante anotaciones como @Getter, @Setter, @Builder, etc.
2. **Spring Web (Web)**  
   Habilita la creación de aplicaciones web, especialmente RESTful APIs, usando Spring MVC y Tomcat como servidor embebido.
3. **Thymeleaf (Template Engines)**  
   Motor de plantillas que permite generar contenido HTML dinámico desde el servidor.
4. **Spring Data JPA (SQL)**  
   Facilita el acceso a bases de datos mediante Java Persistence API (JPA) y Hibernate.
5. **MySQL Driver (SQL)**  
   Controlador JDBC necesario para conectar la aplicación con bases de datos MySQL.
6. **Spring Boot DevTools (Developer Tools)**  
   Herramientas de desarrollo que permiten reinicios automáticos, recarga en caliente (LiveReload) y una experiencia de desarrollo más ágil.

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.



**Estructurar el proyecto y crear las clases para el modelo 02**

Para la estructura del proyecto se necesita una **estructura típica de paquetes (carpetas) organizada por capas según la arquitectura por capas o arquitectura de tipo MVC (Modelo - Vista - Controlador).**

**Paquetes que aparecen:**

**1. controller**

**Responsabilidad:** manejar las peticiones HTTP que vienen del cliente (navegador, Postman, etc.).

* Contiene las clases con anotaciones como @RestController o @Controller.
* Cada método se encarga de atender una URL y delega la lógica al servicio.

**2. model**

**Responsabilidad:** definir las entidades del dominio (por ejemplo, productos, usuarios, órdenes, etc.).

* Contiene clases anotadas con @Entity si usas JPA.
* Representan las tablas de tu base de datos.

**3. repository**

**Responsabilidad:** acceder a la base de datos.

* Contiene interfaces que extienden JpaRepository, CrudRepository, etc.
* Spring genera la implementación automáticamente.

**4. service**

**Responsabilidad:** contener la lógica de negocio.

* Aquí se hace el procesamiento de datos, validaciones, llamadas a los repositorios, etc.
* Usualmente están anotadas con @Service.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Dentro de la carpeta model creamos 4 clases de entidad que representan las tablas en la base de datos estas clases son Usuario, Producto. Orden, DetalleOrden y contienen la siguiente estructura

**Anotaciones de Lombok**

Estas anotaciones son parte de la librería Project Lombok que ayuda a reducir código repetitivo:

| **Anotación** | **¿Qué hace?** |
| --- | --- |
| @Getter | Genera automáticamente los métodos get para todos los atributos. |
| @Setter | Genera automáticamente los métodos set. |
| @AllArgsConstructor | Genera un constructor con todos los campos como parámetros. |
| @NoArgsConstructor | Genera un constructor vacío (sin argumentos). |

**Atributos de la clase**

La clase tiene los siguientes atributos:

Esto representa un usuario típico con:

* id: Identificador único.
* nombre: Nombre real.
* username: Nombre de usuario.
* mail: Correo electrónico.
* direccion: Dirección física.
* telefono: Número de contacto.
* tipo: Posiblemente el rol del usuario (ej: admin, cliente...).
* password: Contraseña (💡 no deberías mostrarla en toString por seguridad).

**🔁 Método toString()**

Este método sobrescrito (@Override) sirve para obtener una representación en texto del objeto, útil para logs o debugging.

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

Texto

El contenido generado por IA puede ser incorrecto.Texto

El contenido generado por IA puede ser incorrecto.Texto

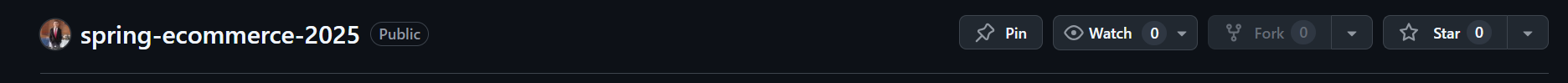
El contenido generado por IA puede ser incorrecto.

Texto

El contenido generado por IA puede ser incorrecto.

**Cómo versionar el proyecto usando Git y GitHub**

Lo Primero que haremos será crear el repositorio donde guardaremos todo el código que vamos a realizar para esto ingresamos a <https://github.com/Ivan-Cast> y creamos un repositorio en mi caso lo llame [**spring-ecommerce-2025**](https://github.com/Ivan-Cast/spring-ecommerce-2025)



Luego de esto desde gitbash o la terminal dentro del proyecto seleccionamos los siguientes comandos

cd /ruta/de/tu/proyecto

Inicializar Git (si aún no está inicializado)

git init

**Vincular el repositorio remoto que ya tienes en GitHub**

Copia la URL de tu repo de GitHub (por ejemplo: https://github.com/usuario/nombre-repo.git) y pon:

git remote add origin https://github.com/usuario/nombre-repo.git

Agregar los archivos al staging

git add .

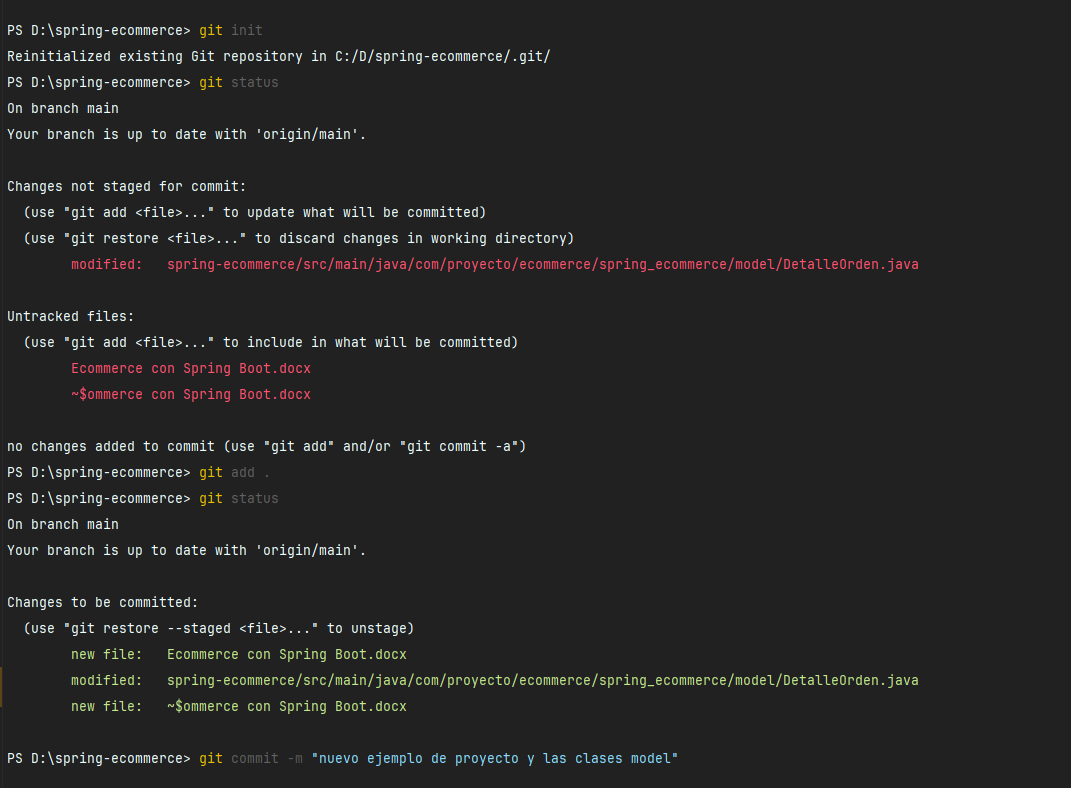
**Hacer el primer commit**

git commit -m "Subiendo proyecto inicial"

**Subirlo al repositorio remoto**

Si tu repositorio remoto ya tiene contenido y quieres fusionar con lo que tienes:

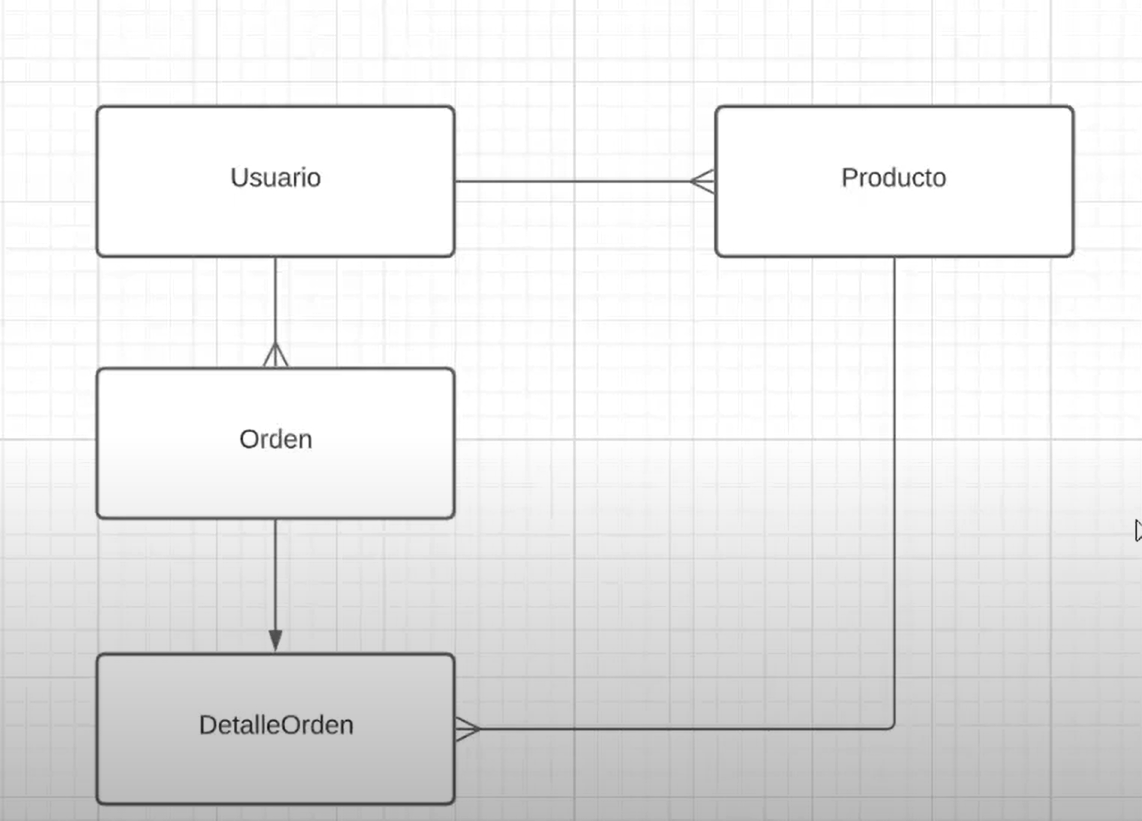
git pull origin main --rebase

****

**Texto

El contenido generado por IA puede ser incorrecto.**

**Mapeo de clases a tablas y relaciones JPA**

****

En este diagrama de entidad relación se muestra las relaciones que van a tener las diferentes tablas de nuestra Base de Datos

Un usuario puede tener de uno a varios productos, un usuario como administrador puede subir a la plataforma uno o varios productos

Un usuario puede tener varias ordenes por que puede realizar varias compras

Una orden puede tener únicamente un detalleOrden

En un detalleOrden puede haber uno o varios productos

**Tabla:** Usuarios

**Relaciones:**

* **Uno a muchos con Producto** → un usuario puede publicar varios productos.
* **Uno a muchos con Orden** → un usuario puede hacer varias órdenes.

**Claves foráneas:** las FK estarán en las tablas Productos y Ordenes apuntando al id de Usuarios.

Texto

El contenido generado por IA puede ser incorrecto.

**Tabla:** Productos

**Relaciones:**

* **Muchos a uno con Usuario** → muchos productos pueden pertenecer al mismo usuario.

**Clave foránea:** usuario\_id en Productos apuntando a Usuarios.id.

Texto

El contenido generado por IA puede ser incorrecto.

* **Tabla:** Ordenes
* **Relaciones:**
  + **Muchos a uno con Usuario** → muchas órdenes pueden ser hechas por un usuario.
  + **Uno a uno con DetalleOrden** → aquí asumes que cada orden tiene un solo detalle (en la vida real podría ser @OneToMany).
* **Clave foránea:** usuario\_id en Ordenes apuntando a Usuarios.id.

Texto

El contenido generado por IA puede ser incorrecto.

* **Tabla:** Detalles
* **Relaciones:**
  + **Uno a uno con Orden** → cada detalle pertenece a una sola orden.
  + **Uno a uno con Producto** → cada detalle hace referencia a un producto específico.
* **Claves foráneas:**
  + orden\_id en Detalles apuntando a Ordenes.id
  + producto\_id en Detalles apuntando a Productos.id

Texto

El contenido generado por IA puede ser incorrecto.

**Nombre de la aplicación**

spring.application.name=spring-ecommerce

* Define el nombre de tu aplicación en Spring Boot.
* Se usa internamente para logs, tracing o cuando usas servicios como Spring Cloud.

**Configuración de la conexión a MySQL**

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

* Indica el driver JDBC de MySQL que Spring Boot usará para conectarse.
* com.mysql.cj.jdbc.Driver es el driver moderno de MySQL (compatible con MySQL 8 y posteriores).

spring.datasource.url=jdbc:mysql://localhost:3306/ecommerce

* Es la URL de conexión a la base de datos.
* localhost → el servidor MySQL está en tu PC.
* 3306 → el puerto por defecto de MySQL.
* ecommerce → el nombre de la base de datos.

spring.datasource.username=root

spring.datasource.password=

* Usuario y contraseña de MySQL.
* Aquí el usuario es root y no hay contraseñ

**Configuración de Hibernate / JPA**

spring.jpa.hibernate.ddl-auto=update

* Indica cómo Hibernate debe manejar el esquema de la base de datos:
  + update → actualiza tablas y columnas según tus entidades, sin borrar datos.
  + Otras opciones: create, create-drop, validate, none.

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

* Le dice a Hibernate qué "dialecto" SQL usar (en este caso, el de MySQL).
* Esto permite que Hibernate genere SQL compatible con MySQL.

spring.jpa.show-sql=true

* Muestra en consola las consultas SQL que ejecuta Hibernate.

spring.jpa.properties.hibernate.format\_sql=true

* Formatea las consultas SQL para que se vean más legibles en la consola.

**Estrategia de nombres de tablas y columnas**

spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl

* Define cómo Hibernate va a convertir nombres de atributos/clases a nombres de columnas/tablas en la base de datos.
* PhysicalNamingStrategyStandardImpl significa que usará exactamente los nombres que pongas en las anotaciones (@Table, @Column), sin modificar a snake\_case.

Texto

El contenido generado por IA puede ser incorrecto.

Texto

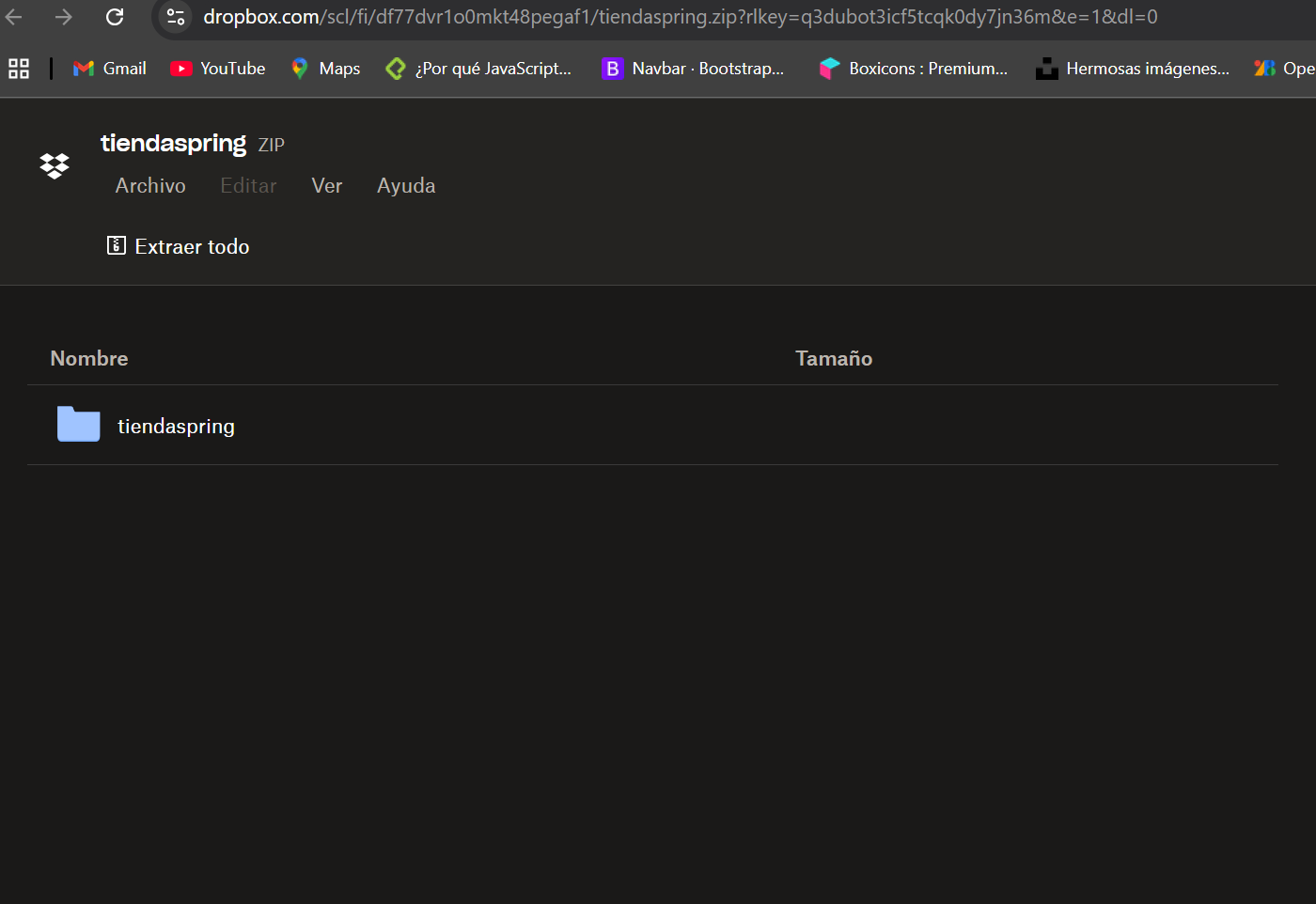
El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

**Integrar Boostrap y la home del administrador**

Lo primero que haremos será descargar el siguiente paquete que contendrá todas las plantillas que vamos a utilizar para nuestro aplicativo que se encuentra en el siguiente enlace <https://www.dropbox.com/scl/fi/df77dvr1o0mkt48pegaf1/tiendaspring.zip?rlkey=q3dubot3icf5tcqk0dy7jn36m&e=1&dl=0>



Posteriormente crearemos las siguientes carpetas

**static/**

* Ubicada en src/main/resources/static/.
* Todo lo que guardes aquí se sirve tal cual al navegador, sin procesarse por el servidor.
* Aquí normalmente van **CSS, JS e imágenes**.

**css/**

* Contendrá tus estilos personalizados (.css) que tú mismo crees para tu sitio.

**vendor/**

* Carpeta para **librerías externas** (vendor = proveedor), por ejemplo Bootstrap y jQuery.
* Evita que se mezclen tus archivos con los de terceros.

**bootstrap/**

* **📂 css/** → Archivos de estilo de Bootstrap (bootstrap.min.css, etc.).
* **📂 js/** → Archivos JavaScript de Bootstrap (bootstrap.bundle.min.js, etc.).

**jquery/**

Aquí están los scripts de jQuery, en diferentes versiones:

* **jquery.js** → Versión completa y no comprimida (útil para desarrollo porque es legible).
* **jquery.min.js** → Versión minificada (comprimida) para producción, más ligera y rápida.
* **jquery.min.map** → Archivo de *source map* que sirve para depuración; permite al navegador mapear el código minificado a su versión original.
* **jquery.slim.js / jquery.slim.min.js** → Versión reducida de jQuery (sin AJAX ni efectos), útil cuando no necesitas todas las funcionalidades.
* **jquery.slim.min.map** → *Source map* de la versión slim minificada.

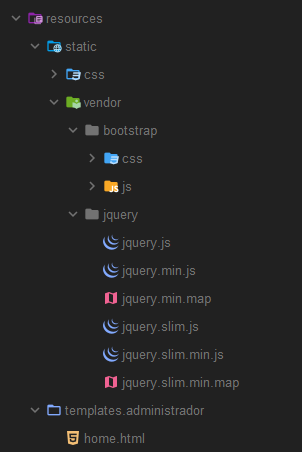
**templates.administrador/**

* En Spring Boot, las vistas HTML suelen ir en src/main/resources/templates/.
* Aquí tienes un subdirectorio administrador para las páginas del panel administrativo.

**home.html**

* Es una plantilla HTML que probablemente se carga desde un controlador con Thymeleaf

Tener en cuenta que estos componentes están en descargable, solo los movemos a la carpeta de static y de templates



Dentro de la plantilla HTML modificaremos las siguientes líneas

**<html lang="es" xmlns:th="http://www.thymeleaf.org">**

* **lang="es"** → Indica que el contenido del documento está en español. Esto ayuda a:
  + Los buscadores (SEO).
  + Lectores de pantalla.
  + Corrección ortográfica.
* **xmlns:th="http://www.thymeleaf.org"** →
  + xmlns significa "XML namespace".
  + Aquí se define el espacio de nombres th para Thymeleaf.
  + Esto permite usar atributos especiales como th:href, th:src, th:text, etc., que Thymeleaf procesará en el servidor antes de enviar el HTML al navegador.

**<link th:href="@{vendor/bootstrap/css/bootstrap.min.css}" rel="stylesheet">**

* **<link>** → Sirve para enlazar archivos externos de CSS.
* **th:href** → Thymeleaf reemplaza la ruta de manera dinámica para que siempre sea correcta, incluso si la app está en un subdirectorio o detrás de un proxy.
* **@{...}** → Sintaxis de Thymeleaf para rutas de recursos estáticos.
* Ruta usada: vendor/bootstrap/css/bootstrap.min.css (dentro de src/main/resources/static).
* **rel="stylesheet"** → Indica que el archivo es una hoja de estilos.

**<link th:thref="@{css/heroic-features.css}" rel="stylesheet">**

* Enlaza una hoja de estilos personalizada heroic-features.css ubicada en static/css/.
* th:href asegura que Thymeleaf resuelva la URL correctamente.

**<script th:src="@{vendor/jquery/jquery.min.js}"></script>**

* **<script>** → Sirve para cargar un archivo JavaScript.
* **th:src** → Thymeleaf genera la ruta absoluta correcta.
* Archivo cargado: vendor/jquery/jquery.min.js desde static/vendor/jquery/.
* Esto carga **jQuery**, una librería JS que simplifica manipulación del DOM y AJAX.

**<script th:src="@{vendor/bootstrap/js/bootstrap.bundle.min.js}"></script>**

* Carga el **Bootstrap JS bundle**, que incluye:
  + Scripts de Bootstrap.
  + Popper.js (necesario para tooltips, popovers, menús desplegables).
* También está en static/vendor/bootstrap/js/.



Texto

El contenido generado por IA puede ser incorrecto.

Texto

El contenido generado por IA puede ser incorrecto.

Por ultimo dentro de la carpeta de Controller vamos a crear una clase controlador que se llamara AdministradorController dentro de ella colocaremos los siguiente

**@Controller**

Indica que esta clase es un Controlador en Spring MVC.

Un controlador recibe las solicitudes (requests) desde el navegador o cliente y devuelve una respuesta, normalmente una vista (HTML) o datos (JSON/XML).

Aquí, como devuelve un String con el nombre de una vista, Spring usará un motor de plantillas (en tu caso probablemente Thymeleaf) para renderizarla.

**@RequestMapping("/administrador")**

Este es el prefijo de la ruta para todos los métodos de esta clase.

Significa que todas las URLs que empiecen por /administrador serán manejadas por este controlador.

Ejemplo: si luego un método tiene @GetMapping("/productos"), la ruta completa sería:

/administrador/productos

**@GetMapping("")**

**public String home(){**

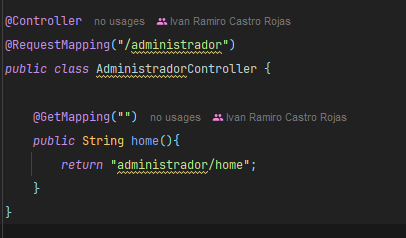
**return "administrador/home";**

**}**

@GetMapping("") significa que se manejarán las peticiones GET a /administrador (porque el prefijo ya lo definiste en la clase).

El método home() devuelve el nombre lógico de la vista:

administrador/home

****

**Crear el template, header y footer**

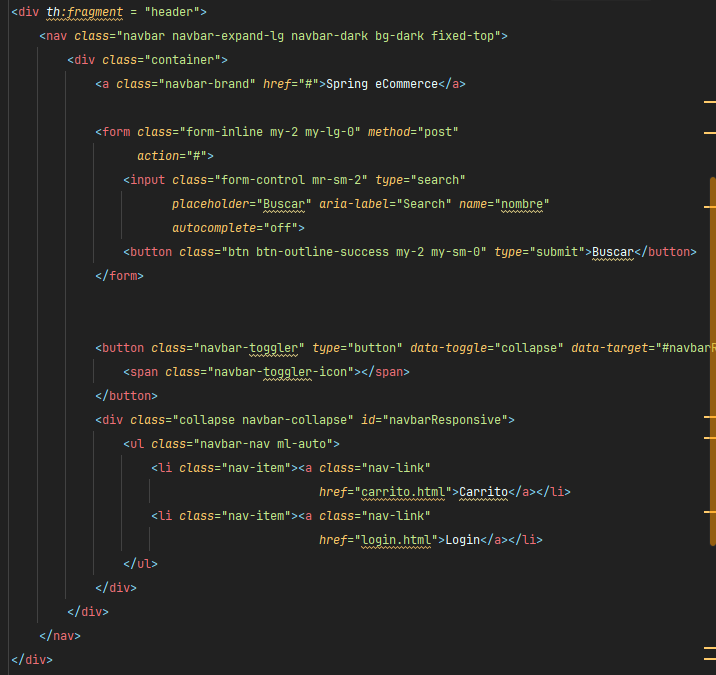
Lo que haremos será aplicar el concepto de **fragments** en **Thymeleaf** para reutilizar código HTML (en tu caso, el **header** y el **footer**) sin tener que copiarlos en cada página.

Crearemos un archivo **template\_admin.html**

En este archivo tienes el HTML base con el header y el footer.  
Los marcaremos como fragmentos con th:fragment:

th:fragment = Le da un **nombre** a esa parte del HTML para que luego pueda ser reutilizada en otras plantillas.

Texto

El contenido generado por IA puede ser incorrecto.

**Archivo home.html**

En vez de volver a escribir todo el código del header y footer, los **incluyes** desde template\_admin.html usando:

Significado:

* administrador/ → es la carpeta donde está el archivo.
* template\_admin.html → el nombre del archivo que contiene los fragmentos.
* ::header → indica **qué fragmento** del archivo quieres traer.
* ::footer → lo mismo, pero para el footer.

**Ventajas de hacerlo así**

* **Reutilización**: Cambias el header o footer en un solo archivo y se actualiza en todas las páginas que lo usan.
* **Menos código duplicado**: Mantienes tu HTML más limpio.
* **Mantenimiento más fácil**: Ideal para proyectos grandes.

Nota: En Thymeleaf moderno (Thymeleaf 3), muchas veces se recomienda usar **th:replace** en lugar de th:include para que el contenido reemplazca el div completamente en vez de anidarse. Ejemplo:

Texto

El contenido generado por IA puede ser incorrecto.

Texto

El contenido generado por IA puede ser incorrecto.

**Crear el repository y clase servicio CRUD producto**

Dentro de la carpeta de Repository vamos a crear una Interfaz llamada ProductoRepository dentro de ella colocamos lo siguiente

**@Repository**

* Es una **anotación de Spring** que marca esta interfaz como un **componente de acceso a datos** (DAO – Data Access Object).
* Spring la detecta automáticamente gracias al **component scanning** y la registra como un **bean** dentro del contenedor.
* También le da la capacidad de manejar excepciones relacionadas con la base de datos (por ejemplo, convertir excepciones específicas de SQL en excepciones genéricas de Spring).

**public interface ProductoRepository**

* Aquí defines un **repositorio** para la entidad Producto.
* Como es una **interfaz**, no necesitas implementar los métodos; Spring Data JPA se encarga de generar la implementación en tiempo de ejecución.

**extends JpaRepository<Producto, Integer>**

* Estás extendiendo de JpaRepository, que es una interfaz que trae muchos métodos listos para usar.
* JpaRepository<T, ID> recibe dos parámetros genéricos:
  + T → la entidad sobre la que trabaja (en este caso Producto).
  + ID → el tipo de la clave primaria (aquí Integer).

Esto significa que ProductoRepository **ya tiene implementados métodos CRUD** como:

* save(Producto p) → para guardar un producto (crear o actualizar).
* findById(Integer id) → buscar por id.
* findAll() → traer todos los productos.
* deleteById(Integer id) → eliminar por id.
* count() → contar registros.

En conclusión:  
Con este ProductoRepository no necesitas escribir SQL ni implementar clases para las operaciones básicas de la base de datos; **Spring Data JPA lo hace automáticamente**.

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

A continuación dentro de la carpeta de Service vamos a crear una Interfaz llamada ProductoService dentro de ella colocamos lo siguiente

**Definición de la interfaz**

* Se define una interfaz en Java llamada ProductoService.
* En una arquitectura de Spring Boot, esta capa corresponde al servicio, es decir, la lógica de negocio relacionada con los productos.
* La idea es definir qué operaciones debe tener el servicio, sin decir cómo se implementan todavía.

**Método save**

* Guarda un producto en la base de datos.
* Retorna el producto guardado (ya con su id generado, si aplica).

**Método get**

* Busca un producto por su id.
* Devuelve un Optional<Producto> porque:
* Si el producto existe → devuelve el producto.
* Si no existe → devuelve vacío (evita NullPointerException).

**Método update**

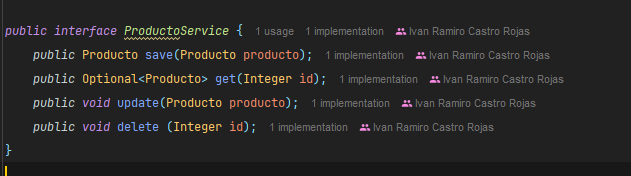
* Actualiza un producto existente.
* Recibe un objeto Producto con la información nueva.
* No devuelve nada (void) porque solo ejecuta la acción.

**Método delete**

* Elimina un producto de la base de datos por su id.
* No devuelve nada (void).

**En resumen**

* La interfaz define un **contrato de operaciones CRUD** (Create, Read, Update, Delete) para la entidad Producto.



Por ultimo en esta misma carpeta vamos a crear una clase que se llama ProductoServiceImpl dentro de esta clase colocaremos lo siguiente

**La anotación @Service**

* @Service indica a **Spring** que esta clase es un **componente de servicio** (lógica de negocio).
* ProductoServiceImpl implementa la interfaz ProductoService, es decir, aquí se define **cómo** se realizan las operaciones que antes solo estaban declaradas.

**Inyección de dependencias con @Autowired**

* Aquí se inyecta (Spring la crea automáticamente) una instancia de ProductoRepository.
* ProductoRepository es el que se encarga de acceder a la base de datos.
* Esto se llama inyección de dependencias → en lugar de crear el objeto con new, Spring lo administra.

**Implementación de los métodos CRUD**

Guardar producto

* Usa el método save del JpaRepository para **guardar un producto**.
* Devuelve el producto ya persistido.

Obtener producto por ID

* Usa findById del repositorio.
* Devuelve un Optional<Producto> (puede estar vacío si no existe).

Actualizar producto

* Para actualizar, también se usa save.
* Si el producto ya tiene un id en la base de datos, lo sobrescribe con los nuevos datos.

Eliminar producto

* Elimina el producto de la base de datos usando su id.

En resumen:  
ProductoServiceImpl es la **clase que ejecuta realmente las operaciones CRUD**, delegando el trabajo al ProductoRepository.

Texto

El contenido generado por IA puede ser incorrecto.

**Conceptos**

@SpringBootApplication(exclude = DataSourceAutoConfiguration.*class*):

**¿Qué hace @SpringBootApplication?**

Esa anotación es un **atajo** que combina varias anotaciones de Spring Boot:

@Configuration

@EnableAutoConfiguration

@ComponentScan

Es decir: le dice a Spring Boot:

“Configura automáticamente mi aplicación, escanea los componentes y usa esta clase como configuración principal.”

**¿Y qué significa exclude = DataSourceAutoConfiguration.class?**

Spring Boot tiene una característica llamada **"auto-configuración"**, donde **detecta automáticamente** que tienes una base de datos y configura la conexión por ti.

Pero si no tienes una base de datos configurada (como es tu caso), **esa auto-configuración falla**.

Al hacer esto:

@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)

Le estás diciendo a Spring Boot:

“❌ No intentes configurar una base de datos automáticamente.”

@Entity

**Qué hace:**  
Indica que la clase es una entidad JPA, es decir, que se va a mapear a una tabla en la base de datos.

**Dónde se usa:**  
Encima de la clase.

**Efecto:**  
Hibernate/JPA creará una tabla (o usará una existente) para almacenar objetos de esta clase.

@Table(name = "...")

**Qué hace:**  
Especifica el nombre de la tabla en la base de datos que representa a la entidad.

**Dónde se usa:**  
Encima de la clase, junto con @Entity.

**Efecto:**  
La tabla se llamará Usuarios en la base de datos, en lugar de usar el nombre por defecto (usuario).

@Id

**Qué hace:**  
Marca el campo que es la clave primaria de la tabla.

**Dónde se usa:**  
En un atributo de la entidad.

**Efecto:**  
Ese campo será la columna de clave primaria.

@GeneratedValue(strategy = ...)

**Qué hace:**  
Indica que el valor de la clave primaria se genera automáticamente.

**Estrategias comunes:**

* + IDENTITY: Usa autoincremento de la base de datos (MySQL, SQL Server).
  + SEQUENCE: Usa una secuencia (PostgreSQL, Oracle).
  + AUTO: Hibernate elige la estrategia según la base de datos.

@OneToMany(mappedBy = "...")

**Qué hace:**  
Define una relación **uno a muchos**.  
mappedBy indica el **nombre del atributo** en la otra entidad que posee la relación (el lado propietario).

**Dónde se usa:**  
En colecciones (List, Set).

**Efecto:**  
Un usuario puede tener muchos productos, pero el campo usuario en Producto es quien guarda la clave foránea.

@ManyToOne

**Qué hace:**  
Define una relación **muchos a uno** (muchos registros de esta entidad pertenecen a un registro de otra entidad).

**Dónde se usa:**  
En el lado que tiene la clave foránea.

**Efecto:**  
En la tabla de esta entidad, habrá una columna usuario\_id que apunta a la tabla Usuarios.

@OneToOne

**Qué hace:**  
Define una relación **uno a uno** entre dos entidades.

**Dónde se usa:**  
En ambos lados de la relación, pero uno es propietario y otro usa mappedBy.

**Efecto:**  
Una fila en una tabla se relaciona con exactamente una fila en otra tabla.

mappedBy

**Qué hace:**  
Indica que la relación **es inversa** y que el campo mencionado es el que realmente guarda la relación en la base de datos.

@Controller

* Marca la clase como un **Controlador Spring MVC**.
* Spring la detecta automáticamente gracias al **component scanning** (búsqueda de componentes) y la registra como un bean dentro del **contexto de la aplicación**.
* Sirve para manejar **peticiones web** y devolver una respuesta.
* Normalmente devuelve:
  + Un **String** con el nombre de una vista (HTML/JSP/Thymeleaf).
  + O un objeto/datos (pero si quieres datos directamente en JSON/XML usarías @RestController o @ResponseBody).

@RequestMapping("/administrador")

* Define el **prefijo de ruta** para todas las peticiones manejadas en esta clase.
* Puede usarse:
  + A **nivel de clase** (como en tu ejemplo) → afecta a todos los métodos.
  + A **nivel de método** → solo afecta a ese endpoint específico.
* Permite definir:
  + Método HTTP (GET, POST, etc.)
  + Producción/consumo de formatos (produces, consumes)
  + Parámetros obligatorios de la URL

@GetMapping("")

* Es un **atajo** de @RequestMapping(method = RequestMethod.GET).
* Sirve para manejar peticiones **GET** (las que normalmente se hacen desde el navegador al escribir una URL).
* En tu caso:
  + La ruta base es /administrador (por la anotación en la clase).
  + Como dentro del @GetMapping("") está vacío, la ruta final será /administrador.

@Repository

* Es una **anotación de Spring** que se coloca sobre clases o interfaces que acceden a la **base de datos**.
* Forma parte del **estereotipo de anotaciones de Spring**, junto con @Controller, @Service y @Component.

@Service

* Marca la clase como un **componente de servicio** dentro de Spring.
* Es parte de la **capa de negocio** (donde se define la lógica).
* Spring detecta esta clase y la **gestiona automáticamente** en el contenedor de beans.

@Autowired

* Indica a Spring que debe inyectar automáticamente una instancia de la clase o interfaz marcada como @Repository, @Service o @Component.
* Aquí Spring crea un objeto de ProductoRepository y lo pone dentro de productoRepository.
* Evita que tengas que usar new ProductoRepository().
* Hoy en día, se recomienda más usar inyección por constructor en vez de campo, para evitar el warning de *"Field dependency injection should be avoided"* en Sonar.