

| | | |
|---|---|---|
|  | <p style="text-align: center;">MINTIC 2022 Desarrollo de Aplicaciones Web CRUD SPRING BOOT CON MYSQL – JPA - THYMELEAF</p> |  |
|---|---|---|

Ejemplo de Spring Boot CRUD con Spring MVC - Spring Data JPA - ThymeLeaf - Hibernate - MySQL.

En este documento de Spring Boot, vamos a desarrollar una aplicación web Java que administra información en una base de datos, con operaciones CRUD estándar : Crear, Recuperar, Actualizar y Eliminar. Usamos las siguientes tecnologías:

- Spring Boot: permite el desarrollo rápido de aplicaciones con valores predeterminados sensibles para reducir el código repetitivo. Spring Boot también nos ayuda a crear una aplicación web Java ejecutable e independiente con facilidad.
- Spring MVC: simplifica la codificación de la capa del controlador. No más código repetitivo de las clases de Java Servlet.
- Spring Data JPA: simplifica la codificación de la capa de acceso a datos. No más código repetitivo de clases DAO.
- Hibernate: se utiliza como marco ORM: implementación de JPA. No más código JDBC repetitivo.
- ThymeLeaf: simplifica la codificación de la capa de visualización. No más etiquetas JSP y JSTL desordenadas.
- Y MySQL como base de datos.

Para el desarrollo de proyectos, utilizamos STS 4.0, JDK 8 y Maven.

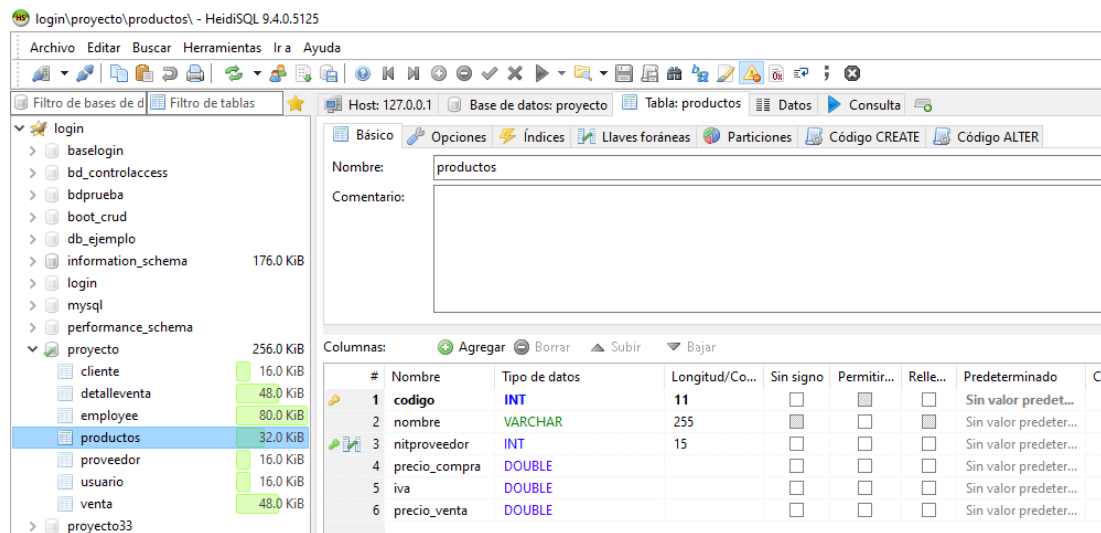
En la siguiente tabla de contenido mostramos la secuencia de nuestro proceso:

1. Cree una base de datos MySQL.
2. Crear proyecto Maven de Spring Boot.
3. Configurar las propiedades de la Base de datos en Spring Boot.
4. Clase de modelo de datos.
5. Código de la Interfaz de repositorio (DAO).
6. Código de la Clase Servicio.
7. Clase de controlador Spring MVC
8. Implementar la función de listar de productos

9. Implementar la función Crear producto
10. Implementar la función Editar producto
11. Implementar la función Eliminar producto
12. Pruebe y empaque la aplicación web Spring Boot CRUD.

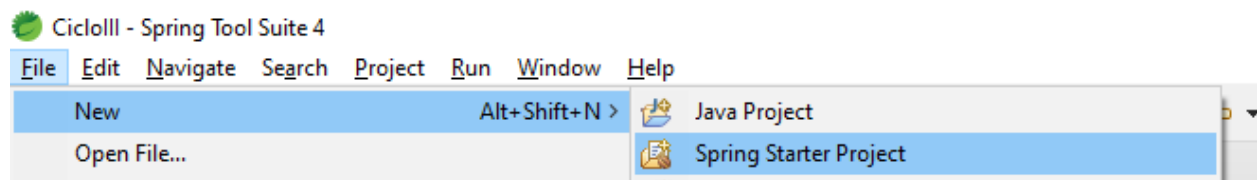
1. Cree una base de datos MySQL.

Supongamos que nuestra aplicación web Spring Boot administrará la información de la tabla producto que usamos en la base de datos del ciclo anterior. Podemos ingresar a algún gestor de MySQL como Heidi y revisar nuestra base de datos como se muestra en la siguiente figura:

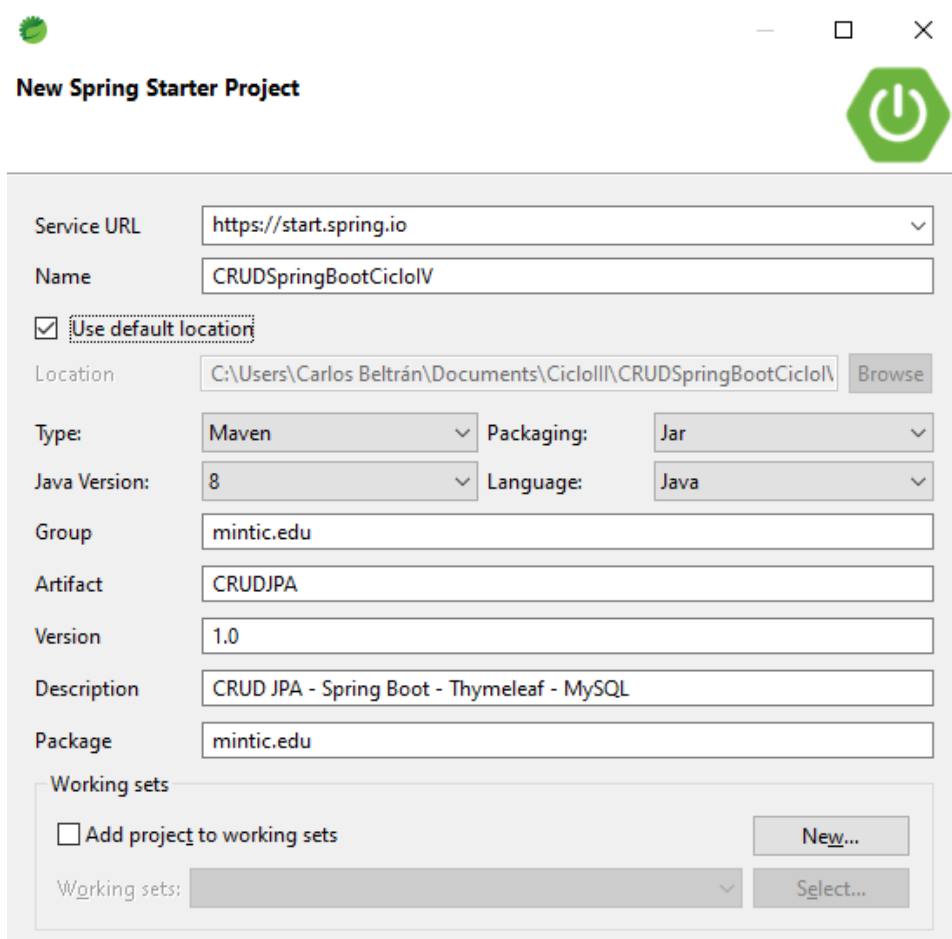


2. Crear proyecto Spring Boot con Maven.

Desde STS 4.0 creamos un nuevo proyecto, seleccionando la opción que se muestra en la figura siguiente:



Ahor definimos los datos deseados o similares a los que se muestran en la siguiente pantalla, vamos a crear un proyecto Maven, jar, con jdk 8 entre las características más importantes.



New Spring Starter Project

Service URL:

Name:

☒ Use default location

Location:

Type: Packaging:

Java Version: Language:

Group:

Artifact:

Version:

Description:

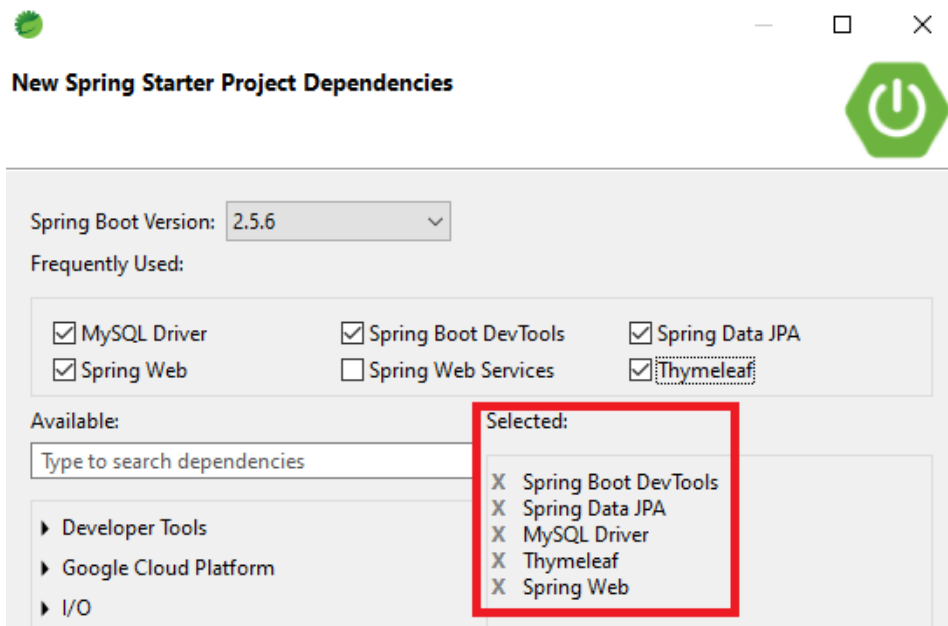
Package:

Working sets

☐ Add project to working sets

Working sets:

Seleccionamos las dependencias que se muestran en la siguiente figura y presionamos Finish para crear el proyecto:



Como se puede ver en el archivo pom.xml del proyecto, con Spring Boot tenemos que especificar solo algunas dependencias: Spring Boot Starter Web, Spring Boot Data JPA, Spring Boot Thymeleaf y el controlador MySQL JDBC.

3. Configurar la Base de Datos en Spring Boot.

Cree el archivo application.properties en el directorio src / main / resources con el siguiente contenido:

```
1 spring.jpa.hibernate.ddl-auto=none
2 spring.datasource.url=jdbc:mysql://localhost:3306/proyecto
3 spring.datasource.username=root
4 spring.datasource.password=mintic
5 logging.level.root=WARN
```

La primera línea le dice a Hibernate que no realice cambios en la base de datos. Especificamos las propiedades de conexión de la base de datos en las siguientes 3 líneas (cambie los valores de acuerdo con su configuración). La última línea establecemos el nivel de registro en WARN para evitar una salida demasiado detallada en la consola.

4. Clase Modelo de Datos.

Cree la clase de modelo de dominio Productos (debe tener el mismo nombre de la Tabla), sobre un nuevo package llamado modelo sobre el package por defecto del proyecto, para mapear con la tabla de productos en la base de datos de la siguiente manera:

```
1 package mintic.edu.modelo;
2
3 import javax.persistence.Entity;
4
5
6
7
8 @Entity          Asocia a la Tabla
9 public class Productos {
10
11     // Atributos
12
13     @Id           Llave Primaria
14     private int codigo;
15     private String nombre;
16     private int nitproveedor;
17     private double precio_compra;
18     private double iva;
19     private double precio_venta;
20
21     // Constructores
22     public Productos() {
23         super();
```

Agregar Constructor Completo
Setter y Getter
toString

Agregamos el constructor con todos los atributos, los setter, getter y el toString para usar en caso de ser necesario.

Esta es una clase de entidad JPA simple con el nombre de la clase y los nombres de los campos son idénticos a los nombres de las columnas del producto de la tabla en la base de datos, para minimizar las anotaciones utilizadas.

5. Código de la Interfaz de repositorio(DAO).

A continuación, cree la interfaz ProductoRepositorio sobre un nuevo package llamado modeloDAO de la siguiente manera:

```

1 package mintic.edu.modeloDAO;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4 import org.springframework.stereotype.Repository;
5
6 import mintic.edu.modelo.Productos;
7
8 public interface ProductoRepositorio extends JpaRepository<Productos, Integer> {
9
10 }
11

```

Clase
Tipo de Dato llave primaria

6. Código de la Clase Servicio(DAO).

A continuación, necesitamos codificar la clase ProductoServicio en la capa de servicio / negocio con el siguiente código:

```

1 package mintic.edu.servicio;
2
3 import java.util.List;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Service;
7 import org.springframework.transaction.annotation.Transactional;
8
9 import mintic.edu.modelo.Productos;
10 import mintic.edu.modeloDAO.ProductoRepositorio;
11
12 @Service
13 @Transactional
14 public class ProductoServicio {
15
16     @Autowired
17     private ProductoRepositorio repo;
18
19     public List<Productos> listar() {
20         // TODO Auto-generated method stub
21         return repo.findAll();
22     }
23
24     public Productos listarId(int id) {
25         // TODO Auto-generated method stub
26         return repo.findById(id).get();
27     }
28
29     public void save(Productos p) {
30         // TODO Auto-generated method stub
31         repo.save(p);
32     }
33
34     public void delete(int id) {
35         // TODO Auto-generated method stub
36         repo.deleteById(id);
37     }
38 }
39
40
41

```

En esta clase, inyectamos una instancia de ProductoRepositorio a través de un campo privado usando la anotación @Autowired, esta anotación evita el tener que instanciar la variable. En tiempo de ejecución, Spring Data JPA generará una instancia de proxy de ProductoRepositorio y la inyectará en la instancia de la clase ProductoServicio.

Es posible que vea que esta clase de servicio es redundante ya que delega todas las llamadas a ProductRepository.

De hecho, la lógica empresarial sería más compleja con el tiempo, por ejemplo, llamar a dos o más instancias de repositorio.

7. Clase Controlador(Método Listar).

A continuación, cree la clase AppControlador que actúa como un controlador Spring MVC para manejar las solicitudes de los clientes, con el código inicial de la siguiente manera:

```

1  package mintic.edu.controlador;
2
3  import java.util.List;
4
5  import org.springframework.beans.factory.annotation.Autowired;
6  import org.springframework.stereotype.Controller;
7  import org.springframework.ui.Model;
8  import org.springframework.web.bind.annotation.RequestMapping;
9
10 import mintic.edu.modelo.Productos;
11 import mintic.edu.servicio.ProductoServicio;
12
13 @Controller
14 public class AppControlador {
15
16     @Autowired
17     private ProductoServicio servicio;
18
19     @RequestMapping("/")
20     public String verIndex(Model model) {
21         List<Productos> listaProductos = servicio.listar();
22         model.addAttribute("listaProductos", listaProductos);
23         return "index";
24     }
25
26 }
27

```

Como se puede ver, inyectamos una instancia de la clase ProductoServicio a este controlador: Spring creará una automáticamente en tiempo de ejecución. Escribiremos código para los métodos del controlador al implementar cada operación CRUD, el único que implementamos por ahora

es el de listar.

8. Implementar la función listar productos.

La página de inicio del sitio web muestra una lista de todos los productos, gracias al método de controlador en la clase de controlador de Spring.

Usamos ThymeLeaf en lugar de JSP, así que cree el directorio de plantillas en src/main/resources para almacenar archivos de plantilla (HTML).

Cree el archivo index.html en src/main/resources/templates con el siguiente código:

```

1 <!DOCTYPE html>
2 <html xmlns="https://www.thymeleaf.org">
3 <head>
4 <meta charset="UTF-8">
5 <title>Gestión de Productos</title>
6 </head>
7 <body>
8 <div align="center">
9 <h1>Listado de Productos</h1>
10 <a href="/new">Crear un nuevo Producto</a> <br />
11 <br />
12 <table border="1">
13 <thead>
14 <tr>
15 <td>Código</td>
16 <td>Nombre</td>
17 <td>Nit Proveedor</td>
18 <td>Precio Compra</td>
19 <td>Iva</td>
20 <td>Precio Venta</td>
21 </tr>
22 </thead>
23 <tbody>
24 <tr th:each="producto : ${listaProductos}">
25 <td th:text="${producto.codigo}"></td>
26 <td th:text="${producto.nombre}"></td>
27 <td th:text="${producto.nitproveedor}"></td>
28 <td th:text="${producto.precio_compra}"></td>
29 <td th:text="${producto.iva}"></td>
30 <td th:text="${producto.precio_venta}"></td>
31 </tr>
32 </tbody>
33 </table>
34 </div>
35 </body>
36 </html>

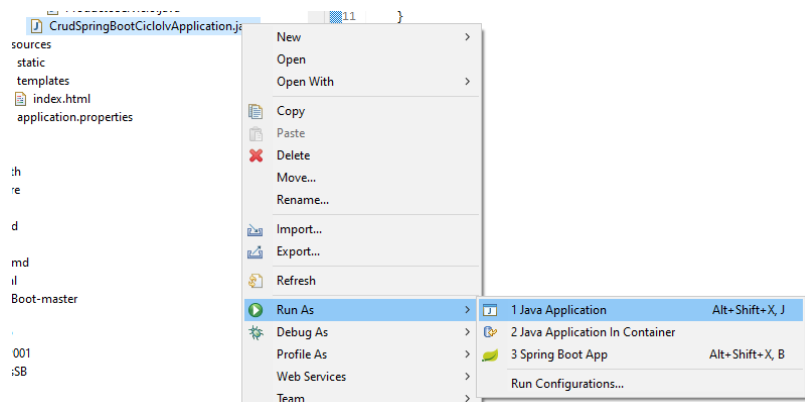
```


El código anterior es el correspondiente a la vista que representa la lista de productos de nuestro ejemplo en una tabla, básicamente es HTML con algunos atributos «raros» que comienzan con th: (th:each, th:text, th:href) que son los propios de Thymeleaf y dentro de estas hay «cosas» que están dentro de `${...}` o de `@{...}`.

El th:each sirve para iterar sobre listas de objetos (sí el `${...}` es para acceder a los objetos) y repetirá el bloque de código dentro de la etiqueta en la se encuentra tantas veces como sea necesario (en este caso pinta una nueva fila de la tabla).

Con th:text podemos mostrar el texto correspondiente al objeto o propiedad de este y con th:href podemos crear enlaces y para no tener que indicar la url completa utilizaremos `@{...}` para solo tener que indicar la ruta relativa desde donde nos encontremos.

Ahora podemos ejecutar la aplicación, usando el archivo que contiene la clase Main llamado "`CrudSpringBootCicloIvApplication`" para probar nuestra aplicación web de Spring Boot. Sobre el nombre del archivo presionamos click derecho, seleccionamos Run As y escogemos Java Application como se muestra en la figura.



Debería ver aparecer el logotipo de Spring Boot en la vista de la consola de STS:

Vamos a un navegador y abrimos sobre localhost el puerto que corresponda al tomcat embebido en el proyecto, debe aparecer un resultado como el siguiente:

Listado de Productos

[Crear un nuevo Producto](#)

| Código | Nombre | Nit Proveedor | Precio Compra | Iva | Precio Venta |
|--------|-------------------|---------------|---------------|------|--------------|
| 1 | Melocotones | 1 | 25505.0 | 19.0 | 30351.0 |
| 2 | Manzanas | 3 | 18108.0 | 19.0 | 21549.0 |
| 3 | Plátanos | 4 | 29681.0 | 19.0 | 35320.0 |
| 4 | Lechuga | 3 | 29788.0 | 19.0 | 35448.0 |
| 5 | Tomates | 1 | 12739.0 | 19.0 | 15159.0 |
| 6 | Calabaza | 1 | 21315.0 | 19.0 | 25365.0 |
| 7 | Apio | 2 | 19249.0 | 19.0 | 22906.0 |
| 8 | Pepino | 2 | 10958.0 | 19.0 | 13040.0 |
| 9 | Champiñones | 2 | 11046.0 | 19.0 | 13145.0 |
| 10 | Leche | 5 | 21150.0 | 19.0 | 25169.0 |
| 11 | Queso | 5 | 26571.0 | 19.0 | 31619.0 |
| 12 | Huevos | 2 | 12445.0 | 19.0 | 14810.0 |
| 13 | Requesón | 1 | 14329.0 | 19.0 | 17052.0 |
| 14 | Crema agria | 1 | 14856.0 | 19.0 | 17679.0 |
| 15 | Yogur | 5 | 14941.0 | 19.0 | 17780.0 |
| 16 | Ternera | 5 | 29335.0 | 19.0 | 34909.0 |
| 17 | Salmón salvaje | 5 | 11878.0 | 19.0 | 14135.0 |
| 18 | Patas de cangrejo | 1 | 29951.0 | 19.0 | 35642.0 |

9. Implementar la función Crear producto.

Puedes ver en el index.html, tenemos un hipervínculo que permite al usuario crear un nuevo producto:

```
10 <a href="/new">Crear un nuevo Producto</a> <br />
```

La URL relativa nueva se maneja mediante el siguiente método en la clase AppControlador:

```
33 @RequestMapping("/new")
34 public String mostrarPaginaNuevoProducto(Model model) {
35     Productos producto = new Productos();
36     model.addAttribute("producto", producto);
37
38     return "nuevo_producto";
39 }
```

Para la vista, cree el archivo nuevo_producto.html con el siguiente código:

```

1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org">
3 <head>
4 <meta charset="utf-8" />
5 <title>Crear Nuevo Producto</title>
6 </head>
7 <body>
8 <div align="center">
9 <h1>Crear Nuevo Producto</h1>
10 <br />
11 <form action="#" th:action="@{/save}" th:object="${producto}" method="post">
12
13 <table border="0" cellpadding="10">
14 <tr>
15 <td>Código:</td>
16 <td><input type="text" th:field="*{codigo}" /></td>
17 </tr>
18 <tr>
19 <td>Nombre de Producto:</td>
20 <td><input type="text" th:field="*{nombre}" /></td>
21 </tr>
22 <tr>
23 <td>Nit Proveedor:</td>
24 <td><input type="text" th:field="*{nitproveedor}" /></td>
25 </tr>
26 <tr>
27 <td>Precio de Compra:</td>
28 <td><input type="text" th:field="*{precio_compra}" /></td>
29 </tr>
30 <tr>
31 <td>Iva:</td>
32 <td><input type="text" th:field="*{iva}" /></td>
33 </tr>
34 <tr>
35 <td>Precio de Venta:</td>
36 <td><input type="text" th:field="*{precio_venta}" /></td>
37 </tr>
38 <tr>
39 <td colspan="2"><button type="submit">Grabar</button> </td>
40 </tr>
41 </table>
42 </form>
43 </div>
44 </body>
45 </html>

```

Como puede ver, aquí usamos la sintaxis Thymeleaf para el formulario en lugar de las etiquetas de formulario Spring.

Y a continuación tenemos un formulario con otros 3 nuevos atributos (th:action, th:object y th:field), el th:action se corresponde al action típico de HTML, en th:object ponemos el objeto queremos rellenar (el que enviaremos al controlador) y con th:field rellenamos sus campos.

La página Crear nuevo producto tiene este aspecto:

Crear Nuevo Producto

| | |
|---------------------------------------|----------------------------------|
| Código: | <input type="text" value="0"/> |
| Nombre de Producto: | <input type="text"/> |
| Nit Proveedor: | <input type="text" value="0"/> |
| Precio de Compra: | <input type="text" value="0.0"/> |
| Iva: | <input type="text" value="0.0"/> |
| Precio de Venta: | <input type="text" value="0.0"/> |
| <input type="button" value="Grabar"/> | |

Y necesitamos codificar otro método de manejo para guardar la información del producto en la base de datos:

```
41 @PostMapping("/save")
42 public String saveProducto(@Validated Productos producto, Model model) {
43     servicio.save(producto);
44
45     return "redirect:/";
46 }
```

Una vez que el producto se inserta en la base de datos, se redirige a la página de inicio para actualizar la lista de productos.

10. Implementar la función Editar producto.

En la página de inicio (index.html) debemos agregar una acción que permita editar el producto, lo agregamos a la tabla sobre una nueva columna, puede ver que hay un hipervínculo que permite a los usuarios editar un producto:


```

30<td>Iva:</td>
31<td><input type="text" th:field="*{iva}" /></td>
32</tr>
33<tr>
34<td>Precio de Venta:</td>
35<td><input type="text" th:field="*{precio_venta}" /></td>
36</tr>
37<tr>
38<td colspan="2"><button type="submit">Grabar</button> </td>
39</tr>
40</table>
41</form>
42</div>
43</body>
44</html>

```

La página de edición del producto debería verse así:

Editar Producto

| | |
|---------------------------------------|--|
| Código: | <input type="text" value="9"/> |
| Nombre de Producto: | <input type="text" value="Champiñones"/> |
| Nit Proveedor: | <input type="text" value="2"/> |
| Precio de Compra: | <input type="text" value="11046.0"/> |
| Iva: | <input type="text" value="19.0"/> |
| Precio de Venta: | <input type="text" value="13145.0"/> |
| <input type="button" value="Grabar"/> | |

Haga clic en el botón Guardar para actualizar la información del producto en la base de datos. En este caso, se reutiliza el método de manejo “save” creado en el controlador.

11. Implementar Eliminar producto.

Puede ver el hipervínculo para eliminar un producto en la página de inicio:

```

<a th:href="@{'/delete/' + ${producto.codigo}}">Eliminar</a>

```

Así que codifique el método del controlador en la clase del controlador de la siguiente manera:

```
--
54 @RequestMapping("/delete/{codigo}")
55 public String deleteProduct(@PathVariable(name = "codigo") int codigo) {
56     servicio.delete(codigo);
57     return "redirect:/";
58 }
```

Cuando el usuario hace clic en el hipervínculo Eliminar, la información del producto correspondiente se elimina de la base de datos y se actualiza la página de inicio.

12. Pruebe y empaquete la aplicación web Spring Boot CRUD.

Para probar la aplicación web Spring Boot que hemos desarrollado en Eclipse, ejecute la clase "CrudSpringBootCicloIvApplication" como aplicación Java.

Para empaquetar la aplicación web como un archivo JAR de ejecución en STS, haga clic con el botón derecho en el proyecto y seleccione Run As > Maven build ... luego ingrese paquete como el nombre del objetivo y haga clic en Ejecutar. Si la compilación tuvo éxito, verá que se genera un archivo JAR en el directorio de destino del proyecto, con el nombre como CRUDJPA-1.0.jar .

Ahora puede usar el comando java para ejecutar este archivo JAR:

```
java -jar CRUDJPA-1.0.jar.
```