

Documentación WebShop

Proyecto personal Bootcamp Fullstack Java

Java Maven Spring MVC 5.3.3 Hibernate 5.4.2

1 Selecciona las columnas requeridas para presentar la información solicitada

R: como ejemplo tenemos la clase ProductoDao en el paquete DAO el cual tiene dos funciones una llamada getProductos y findById las cuales tiene sus consultas automatizadas por hibernate, pero básicamente se pueden interpretar de la siguiente manera

getProductos es una consulta general a todo el listado de productos
select * from productos;

findById es una consulta que trae todos los campos de un producto en particular restringiéndolo por su identificador.
Select * from productos WHERE
idProducto id = ?

```
@Component
public class ProductoDao {
    @Autowired
    HibernateTemplate hibernateTemplate;
    public List<Productos> getProductos() {
        // TODO Auto-generated method stub
        return hibernateTemplate.loadAll(Productos.class);
    }
    public Productos findById(Integer idProducto) {
        return hibernateTemplate.get(Productos.class, idProducto);
    }
}
```

2 Utiliza JOIN para relacionar la información de distintas tablas

Los join están declarados en las clases gracias al mapeo de hibernate en este ejemplo tenemos a la clase Productos de paquete MODEL, a su vez también se declaran el id e incluso los orderby para ordenarlos por uno o mas campos.

```
@Entity
public class Productos {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int idProducto;

    private String nombreProducto;

    private String descripcion;
    private int precio;
    private int stock;
    private java.sql.Timestamp creado;
    private java.sql.Timestamp actualizado;
    private String imagen;
    private String imagenSlider;
    @OrderBy("nombreProducto ASC")
    @JoinColumn(name = "categorias_idCategoria")
    private String categorias_idCategoria;
    @JoinColumn(name = "proveedores_idProveedor")
    private String proveedores_idProveedor;
    @OneToOne
    @JoinColumn(name = "estados_idEstado")
    private Estados estados;
    ...
}
```

3 Utiliza WHERE para filtrar la información requerida.

Al igual que otros casos las consultas las maneja el orm y en este caso para el WHERE si nos fijamos en la imagen 1 ProductoDao la segunda función findById esta filtrando por un id específico que a su vez se reduce a un WHERE id=?, o por ejemplo findByIdCriteria seria algo parecido a Select * from WHERE criteria(parametro) and criteria (parametro2)

```
public Usuario authenticate(String email, String password) throws AuthenticationException {
    DetachedCriteria criteria = DetachedCriteria.forClass(Usuario.class);
    criteria.add(Restrictions.eq("email", email));
    criteria.add(Restrictions.eq("password", password));
    List<Usuario> usuarios = (List<Usuario>) hibernateTemplate.findByCriteria(criteria);
    if (usuarios.isEmpty()) {
        throw new AuthenticationException("Correo electrónico o contraseña incorrectos.");
    }
    Usuario usuario = usuarios.get(0);

    if (usuario==null) {
        throw new AuthenticationException("Correo electrónico o contraseña incorrectos.");
    }

    if (!usuario.getPassword().equals(password)) {
        throw new AuthenticationException("Correo electrónico o contraseña incorrectos.");
    }

    return usuario;
}
```

4 Utiliza cláusulas de ordenamiento para presentar la información.

Como podemos apreciar en la imagen 2 aparece la siguiente línea de referencia

@OrderBy (nombreProducto ASC) esto quiere decir que las consultas serán ordenadas alfabéticamente por medio de la columna nombreProducto, otro ejemplo sería ordenar los usuarios por nombre de usuario

```
@Entity
@Table(name = "usuario")
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int idUsuario;
    private String nombreUsuario;
    private String email;
    private String telefono;
    private String password;
    private String direccion;
    private java.sql.Timestamp creado;
    private java.sql.Timestamp actualizado;
    private String token;
    private java.sql.Timestamp setExpirationDate;
    @OrderBy("nombreUsuario ASC")
    @OneToOne
    @JoinColumn(name = "roles_idRol")
    private Rol roles;
    @OneToOne
    @JoinColumn(name = "estados_idEstado")
    private Estados estados;
}
```

5 Utiliza cláusulas de agrupación de información para obtener datos agregados

Una consulta simple encargada de generar un listado de la cantidad de productos por categoría para un aside que no llego a la versión actual y se quedo en desarrollo

```
public List<Object[]> getProductosByCategoria() {  
    Session session = hibernateTemplate.getSessionFactory().getCurrentSession();  
    Query query = session.createQuery("select c.nombreCategoria, count(p.idProducto) "  
        +" |from Categorías c inner join c.productos p group by c.idCategoria");  
    List<Object[]> results = query.getResultList();  
    return results;  
}
```

6 Utilización general del lenguaje, sintaxis, selección de tipos de datos, sentencias lógicas, expresiones, operaciones, comparaciones

En este fragmento de la función `agregarProductoACarrito()` en el controlador `CarritoController` tenemos declaración, asignación y sentencias lógicas mas ciclos iterativos.

```
if (!productoEncontrado) {  
    // Si el producto no está en el carrito, creamos un nuevo detalle y lo agregamos a la lista  
    DetalleCarrito nuevoDetalle = new DetalleCarrito();  
    nuevoDetalle.setCreado(new Timestamp(System.currentTimeMillis()));  
    nuevoDetalle.setActualizado(new Timestamp(System.currentTimeMillis()));  
    nuevoDetalle.setCarrito(carrito); // El carrito obtenido anteriormente  
    nuevoDetalle.setProducto(producto); // El producto que se va a agregar  
    nuevoDetalle.setCantidad(cantidad); // Cantidad fija por ahora  
    nuevoDetalle.setSubTotal(producto.getPrecio()*cantidad);  
    //carrito.setTotal(carrito.getSubTotal()+nuevoDetalle.getProducto().getPrecio());  
    detallesCarrito.add(nuevoDetalle);  
    carrito.setDetallesCarrito(detallesCarrito);  
    int total=0;  
    for(DetalleCarrito detallesCarritos: detallesCarrito ) {  
        total=total+detallesCarritos.getSubTotal();  
        carrito.setTotal(total);  
    }  
    carritoServices.updateCarritoTotal(carrito.getTotal(),carrito.getIdCarrito());  
    DetalleCarrito detalleCarritoGuardado = detalleCarritoServices.saveDetalleCarrito(nuevoDetalle);  
}
```

7 Utilización de sentencias repetitivas

```
List<DetalleCarrito> detallesCarrito = carrito.getDetallesCarrito();
boolean productoEncontrado = false;
for (DetalleCarrito detalle : detallesCarrito) {
    if (detalle.getProducto().getIdProducto() == idProducto) {
        // Si el producto ya está en el carrito, incrementamos la cantidad
        detalle.setCantidad(detalle.getCantidad() + cantidad);
        detalle.setSubTotal((detalle.getCantidad()) * (detalle.getProducto().getPrecio()));
        detalleCarritoServices.updateDetalleCarrito(detalle.getIdDetalleCarrito(), detalle.getCantidad(), detalle.getSubTotal());
        int total=0;
        for(DetalleCarrito detallesCarritos: detallesCarrito ) {
            total=total+detallesCarritos.getSubTotal();
            carrito.setTotal(total);
        }
        //falta agregar la fecha y hora de la modificacion en el carrito
        carritoServices.updateCarritoTotal(carrito.getTotal(),carrito.getIdCarrito());
        productoEncontrado = true;
        break;
    }
}
```

en este caso tenemos un ciclo repetitivo encargado de calcular el Total del carrito en caso de que el producto agregado ya se encuentre en el carro le suma la cantidad nueva de productos a la existente y actualiza el total, este fragmento fue tomado de la función agregarProductoACarrito() de CarritoController

8 Utilización de clases, encapsulamiento y responsabilidad única

Como podemos observar en la imagen 2 Productos es una clase que ya tiene cierto grado de abstracción y esta dedicada a establecer los atributos con los que cuentan los productos y también cuenta con encapsulamiento al declarar dichos atributos de manera privada y los métodos set y get ser los encargados de leer y escribir los datos dentro de un objeto de tipo Producto.

```
public Productos(int idProducto, String nombreProducto, String descripcion, int precio, int stock, Timestamp creado, Timestamp actualizado, String imagen, String imagenSlider, String categorias_idCategoria, String proveedores_idProveedor, Estados estados) {
    super();
    this.idProducto = idProducto;
    this.nombreProducto = nombreProducto;
    this.descripcion = descripcion;
    this.precio = precio;
    this.stock = stock;
    this.creado = creado;
    this.actualizado = actualizado;
    this.imagen = imagen;
    this.imagenSlider = imagenSlider;
    this.categorias_idCategoria = categorias_idCategoria;
    this.proveedores_idProveedor = proveedores_idProveedor;
    this.estados = estados;
}

public Productos() {
    super();
}

public int getIdProducto() {
    return idProducto;
}

public void setIdProducto(int idProducto) {
    this.idProducto = idProducto;
}

public String getNombreProducto() {
    return nombreProducto;
}

public void setNombreProducto(String nombreProducto) {
    this.nombreProducto = nombreProducto;
}

public String getDescripcion() {
    return descripcion;
}

public void setDescripcion(String descripcion) {
```

9 Se utilizan correctamente interfaces o relaciones de herencia para hacer polimorfismo donde fuese necesario

Siguiendo el patrón MVC mi CarritoServices es una interfaz que se implementa en CarritoServicesImpl y a su vez cada uno de los servicios tiene una implementación

```
package BootCamp.WebShop.service;

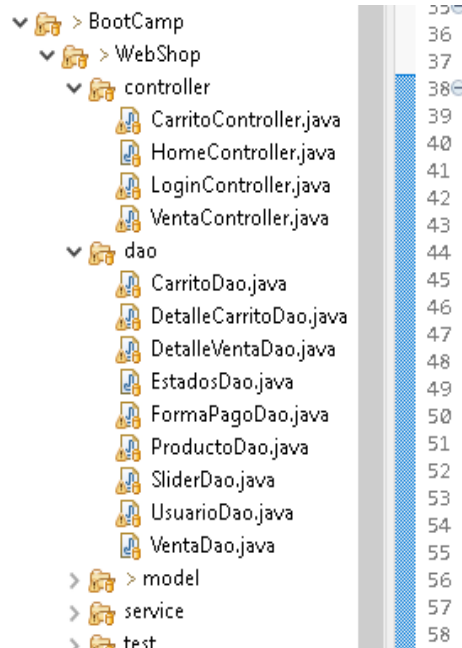
import java.util.List;

import org.springframework.stereotype.Service;
import BootCamp.WebShop.model.Carrito;
import BootCamp.WebShop.model.Estados;

@Service
public interface CarritoServices {
    public Carrito saveCarrito(Carrito carrito);
    public Carrito getCarritoById(Long idCarrito);
    public List<Carrito> getAllCarritos();
    public void deleteCarrito(Long idCarrito);
    public void updateCarritoTotal(int total, int idCarrito);
    public void actualizarEstado(int idCarrito, Estados estado);
}
```

10 Convenciones y estilos de programación

Siguiendo las convenciones de la programación en Java se utilizan UpperCamelCase para las clases y LowerCamelCase para las funciones y variables



```
36 private DetalleVentaServices detalleVentaServices;
37
38 @RequestMapping(value = "/procesaPago", method = RequestMethod.POST)
39 public ModelAndView procesaPago(HttpServletRequest request, HttpSession session) {
40     ModelAndView modelAndView = new ModelAndView();
41     Long idFormaPago = (long) 1;
42     idFormaPago = Long.parseLong(request.getParameter("idFormaPago"));
43
44
45     // Buscamos la forma de pago seleccionada en la base de datos
46     FormaPago formaPago = formaPagoService.findById(idFormaPago);
47
48     // Obtenemos el carrito de la sesión
49     Carrito carrito = (Carrito) session.getAttribute("carrito");
50     Venta venta;
51
52     // Obtenemos el usuario de la sesión
53     Usuario usuario = (Usuario) session.getAttribute("usuario");
54     if (usuario == null) {
55         // Si no hay un usuario en sesión, no podemos crear un carrito, redireccionamos a la
56         modelAndView.setViewName("redirect:/login");
57         return modelAndView;
58     }
}
```

11 Utilización de unidades de prueba

Para realizar pruebas unitarias instale JUNIT y cree el paquete de Test, a pesar de que no realice pruebas exhaustivas a funciones o las clases en general, si entendí su función e importancia, aunque sea a ultima hora

```
1 package BootCamp.WebShop.test;
2
3 import static org.junit.Assert.assertEquals;
4
5 import java.time.LocalDateTime;
6
7 import org.junit.*;
8
9 import BootCamp.WebShop.model.Usuario;
10
11 public class UsuarioTest {
12
13     @Test
14     public void testGetNombreUsuario() {
15         Usuario usuario = new Usuario();
16         usuario.setNombreUsuario("Juan");
17         assertEquals("Juan", usuario.getNombreUsuario());
18     }
19
20 }
```

12 Utilización de tags html, estilos y responsividad

Para las vistas utilice Html + Bootstrap5.3 CDN y también un poco de JS y CSS personalizado para que no fuera puramente Bootstrap ya que le da un aspecto muy genérico, pero hay que tener en cuenta que el front no es mi fuerte aun asi considero que esta decente, en algunas vistas también agregue Sweet Alert para los mensajes

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

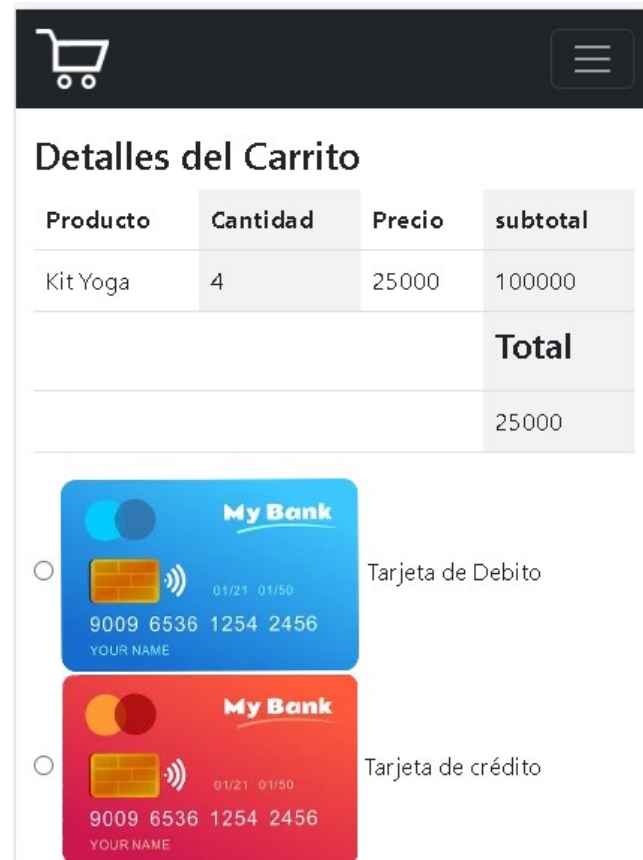
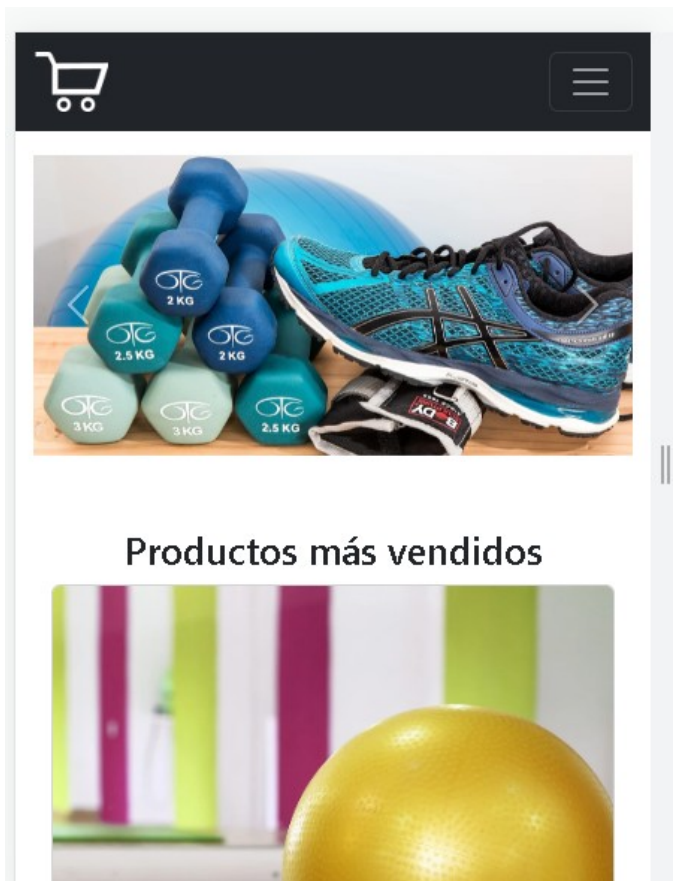
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Web Shop</title>
<link rel="shortcut icon" type="image/x-icon"
    href="${pageContext.request.contextPath}/img/favicon.ico">
<meta name="viewport" content="width=device-width, initial-scale=1">
<!-- Agregar enlaces de Bootstrap 5.3 CDN -->
<link
    href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha3/dist/css/bootstrap.min.css"
    rel="stylesheet"
    integrity="sha384-KK94CHFLLe+nY2dnCWqG91rCGa5gtU4mk92HdvYe+M/SXH301p5ILy+dN9+nJZ"
    crossorigin="anonymous">
<link rel="stylesheet"
    href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.10.5/font/bootstrap-icons.css">
<script src="${pageContext.request.contextPath}/js/webshop.js"></script>
<link rel="stylesheet"
    href="${pageContext.request.contextPath}/css/webshop.css">
</head>
<body>
<!-- Incrusta header -->
<jsp:include page="/WEB-INF/views/partials/header.jsp" />

<div class="container mt-3">
<!-- Slider -->
<div id="carouselExampleControls" class="carousel slide">
<!-- Productos más vendidos -->
<div class="container my-5">

<!-- Footer -->
<footer class="bg-light text-center text-lg-start">
```

13 Utilización de Botstrap

Como se demuestra el la imagen anterior estoy utilizando bootstrap 5.3, ademas las imágenes están en formato webp para optimizar la carga y facilitar la navegación, cada una de las vistas tiene resposividad siguiendo el patrón de grillas y columnas de bootstrap



14 Utilización de Controllers

Los controladores son la parte esencial de la estructura lógica y se encargan de procesar y servir la información derivando sus resultados consumiendo la capa de servicio y recibiendo la información que se recoge de los formularios en las vistas

```
@Controller
@RequestMapping("/carrito")
public class CarritoController {

    private ProductoServices productoServices;

    private CarritoServices carritoServices;

    private DetalleCarritoServices detalleCarritoServices;
    @Autowired
    private FormaPagoServices formaPagoServices;

    public ModelAndView verCarrito(HttpSession session) {}

    @RequestMapping(value = "/agregar", method = RequestMethod.POST)
    public ModelAndView agregarProductoACarrito(HttpServletRequest request, HttpSession session) {
        ModelAndView modelAndView = new ModelAndView();
        Long idProducto = Long.parseLong(request.getParameter("idProducto"));
        Integer cantidad = Integer.parseInt(request.getParameter("cantidad"));
        // Obtenemos el producto a agregar al carrito
        Productos producto = productoServices.getProductoById(idProducto);
        // Verificamos si el producto existe
        if (producto == null) {
            modelAndView.addObject("error", "El producto no existe");
            modelAndView.setViewName("error");
            return modelAndView;
        }
        // Obtenemos el carrito de la sesión actual
        Carrito carrito = (Carrito) session.getAttribute("carrito");
        if (carrito == null) {
            // Si no existe un carrito en la sesión, creamos uno
            Usuario usuario = (Usuario) session.getAttribute("usuario");
            if (usuario == null) {
                // Si no hay un usuario en sesión, no podemos crear un carrito, redireccionamos a la página de login
                modelAndView.setViewName("redirect:/login");
                return modelAndView;
            }
        }
    }
}
```

15 Utilización de vistas JSP y Taglib

Al igual que otras vistas donde se llenan de manera dinámica la vista del carrito tiene el taglib prefix C para escribir de manera recursiva cada uno de los productos en el carrito y su detalle o en otras vistas para cargar los sliders desde la BD (ruta a las imágenes) o cargar el listado de productos en promoción

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<html>
<head>
<body>
    <!-- Incrusta header -->
    <jsp:include page="/WEB-INF/views/partials/header.jsp" />

    <div class="container mt-3">
        <div class="carrito">
            <h2>Detalles del Carrito</h2>
            <c:if test="${mensaje != null}">
                <div>${mensaje}</div>
            </c:if>
            <c:if test="${mensaje == null}">
                <table class="table table-striped-columns">
                    <thead>
                        <tr>
                            <th>Producto</th>
                            <th>Cantidad</th>
                            <th>Precio</th>
                            <th>Subtotal</th>
                        </tr>
                    </thead>
                    <tbody>
                        <c:forEach items="${carrito.getDetallesCarrito()}"
                            var="detalle">
                            <tr>
                                <td>${detalle.getProducto().getNombreProducto()}</td>
                                <td>${detalle.getCantidad()}</td>
                                <td>${detalle.getProducto().getPrecio()}</td>
                                <td>${detalle.getSubTotal()}</td>
                            </tr>
                        </c:forEach>
                    </tbody>
                </table>
            </c:if>
        </div>
    </div>
</body>
</html>
```


16 Creación Servicio Spring

En Spring, los servicios se implementan típicamente utilizando la anotación `@Service`, que marca una clase como un componente de servicio.

```
1 package BootCamp.WebShop.service;
2
3 import java.util.List;
4
5 import org.springframework.stereotype.Service;
6 import org.springframework.transaction.annotation.Transactional;
7 import BootCamp.WebShop.model.Carrito;
8 import BootCamp.WebShop.model.Estados;
9 @Service
10 public interface CarritoServices {
11     public Carrito saveCarrito(Carrito carrito);
12     public Carrito getCarritoById(Long idCarrito);
13     public List<Carrito> getAllCarritos();
14     public void deleteCarrito(Long idCarrito);
15     public void updateCarritoTotal(int total, int idCarrito);
16     public void actualizarEstado(int idCarrito, Estados estado);
17 }
18
```

17 Creación DAO acceso a datos

En una aplicación Java con el framework Spring, la capa DAO (Data Access Object) se utiliza para gestionar la interacción con la base de datos. La capa DAO actúa como una abstracción entre la lógica de negocio de la aplicación y los detalles de acceso a datos, en este caso al estar utilizando hibernate las consultas están estandarizadas y son realizadas en background

```
1 package BootCamp.WebShop.dao;
2 import java.util.List;
3 import javax.transaction.Transactional;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.orm.hibernate5.HibernateTemplate;
6 import org.springframework.stereotype.Component;
7 import BootCamp.WebShop.model.Productos;
8
9 @Component
10 public class ProductoDao {
11     @Autowired
12     HibernateTemplate hibernateTemplate;
13     public List<Productos> getProductos() {
14         // TODO Auto-generated method stub
15         return hibernateTemplate.loadAll(Productos.class);
16     }
17     public Productos findById(Integer idProducto) {
18         return hibernateTemplate.get(Productos.class, idProducto);
19     }
20
21 }
22
```

18 Creación del proyecto y configuración

El archivo XML de configuración de Spring define la estructura y las dependencias de los beans de la aplicación. Los beans son los componentes principales de Spring, representando objetos gestionados por el contenedor de Spring. Estos objetos pueden ser controladores, servicios, repositorios, componentes personalizados u otros objetos necesarios para la aplicación.

```
<context:component-scan base-package="BootCamp.WebShop" />

<tx:annotation-driven />
<mvc:annotation-driven />

<mvc:resources mapping="/WEB-INF/resources/**" location="/WEB-INF/resources/" />
<mvc:resources mapping="/img/**" location="/resources/static/img/" />
<mvc:resources mapping="/img/**/*.webp" location="/resources/static/img/" />
<mvc:resources mapping="/js/**" location="/resources/static/js/" />
<mvc:resources mapping="/css/**" location="/resources/static/css/" />

<!-- View Resolver -->
<bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/" />
    <property name="suffix" value=".jsp" />
</bean>


<!-- data source -->
<bean
    class="org.springframework.jdbc.datasource.DriverManagerDataSource"
    name="ds">

    <property name="driverClassName"
        value="com.mysql.jdbc.Driver" />
    <property name="url"
        value="jdbc:mysql://localhost:3306/web_shop_bd" />
    <property name="username" value="root" />
    <property name="password" value="12345" />
</bean>
```

19 Funcionamiento general del aplicativo

En general, WebShop es un sistema de compra online con carrito. Aunque aún quedan algunas horas de trabajo para completarlo, se pueden agregar muchos módulos adicionales y especializarse en algún campo en particular. Precisamente por esta razón, se creó desde un principio de manera genérica.

Productos más vendidos




Kit Yoga
Set de implementos para la práctica de Yoga
Precio: 25000

- 1 + [Agregar](#)



Kit Mancuernas
Set de mancuernas de diferentes pesos
Precio: 35000

- 1 + [Agregar](#)



Ab Wheel
Rueda de ejercicio para trabajar el abdomen
Precio: 15000

- 1 + [Agregar](#)

20 Creación servicio Rest

Al día de la evaluación el proyecto no tenía creado o consumía ningún api/api rest