HIBERNATE

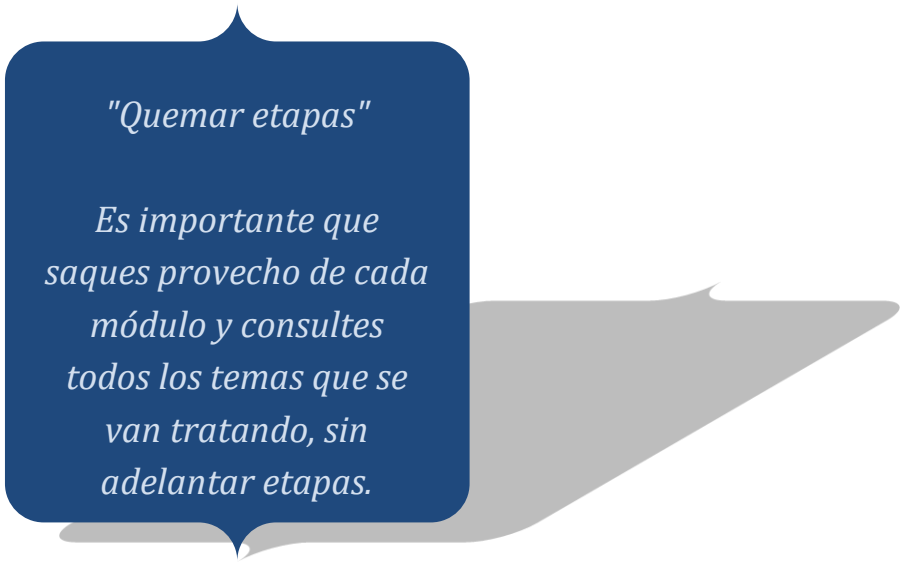
"Spring Web MVC - Hibernate"

Módulo 6 / 2

AUTOR: Andrés Guzmán Fontecilla.
Email: andresguzf@gmail.com
WEB: http://www.bolsadeideas.com

Objetivo

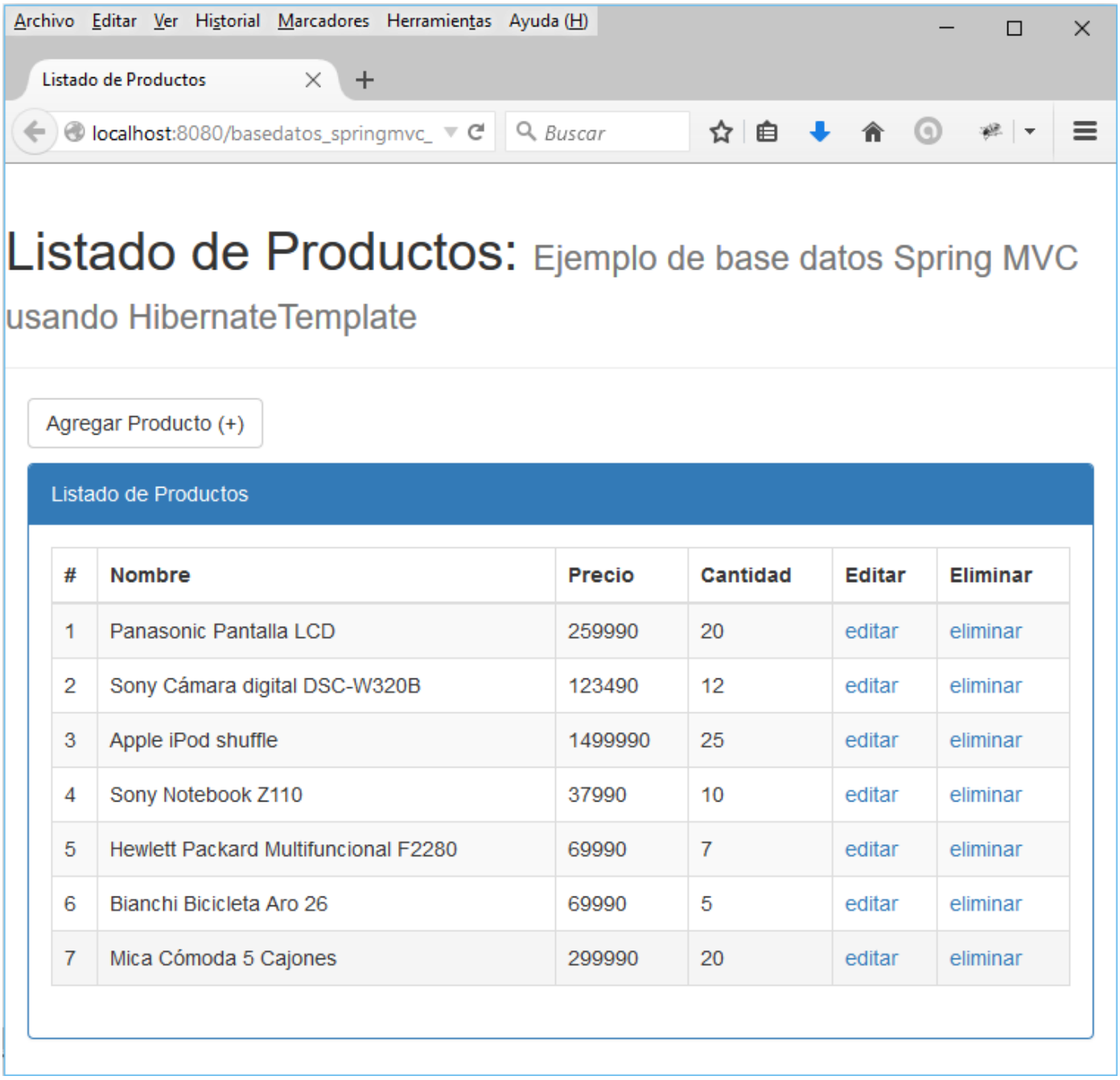
En este laboratorio veremos y aprenderemos todo lo relacionado a la persistencia y base de datos con **Spring Web MVC** con una completa aplicación CRUD (Crear, actualizar, Borrar y Listar), un mantenedor de Productos utilizando Hibernate.



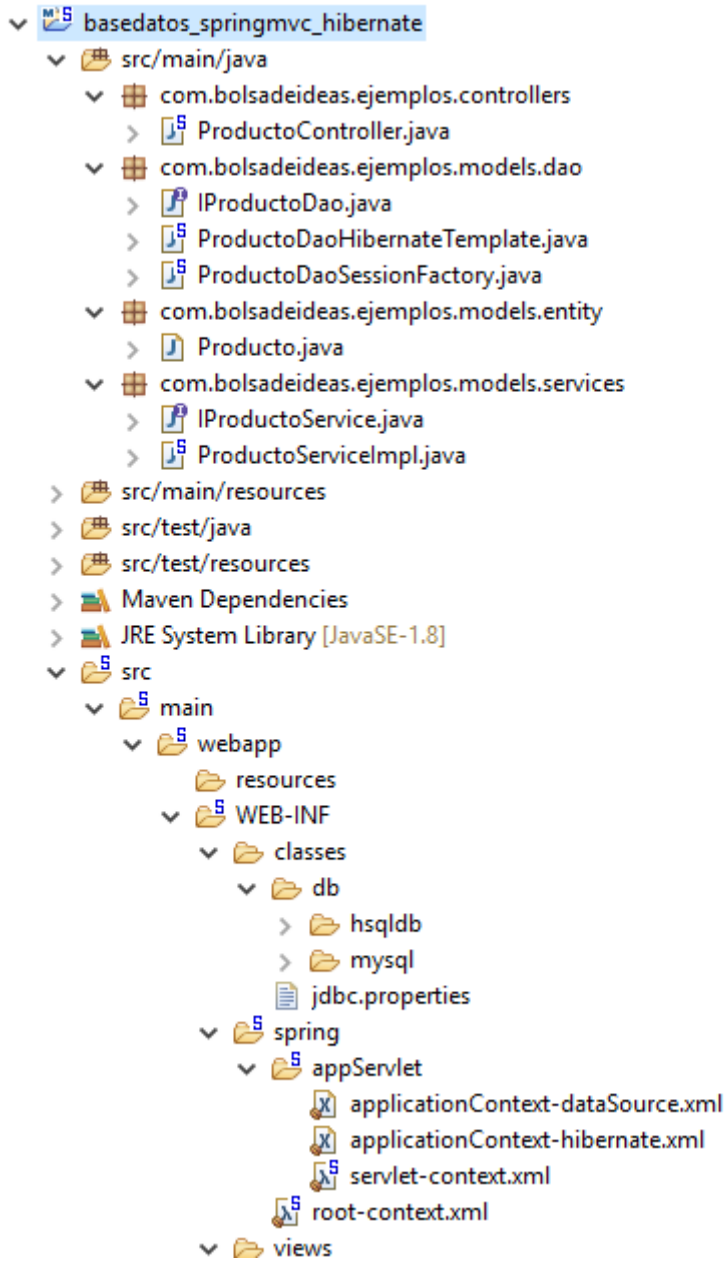
AUTOR: Andrés Guzmán Fontecilla.
Email: andresguzf@gmail.com
WEB: http://www.bolsadeideas.com

Ejercicio 1: Generar y ejecutar el ejemplo "basedatos_springmvc_hibernate"

1. Clic derecho sobre el proyecto **Run As->Maven Clean** y **Run As->Maven Install**.
2. Clic derecho sobre el proyecto y **Maven->Update Project…**
3. Clic derecho sobre el proyecto **basedatos_springmvc_hibernate-> Run As on Server**
4. Observe el resultado en el navegador



5. Estudiemos la estructura del proyecto:



6. Abrir y estudiar la clase de Dominio o Entity **/src/main/java/com.bolsadeideas.ejemplos.models.entity/Producto.java** (usando anotaciones)

```
package com.bolsadeideas.ejemplos.models.entity;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="productos")
public class Producto implements Serializable{

    private static final long serialVersionUID = 1L;

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

     private String nombre;
     private Integer precio;
     private Integer cantidad;

     public Integer getCantidad() {
         return cantidad;
     }

     public void setCantidad(Integer cantidad) {
         this.cantidad = cantidad;
     }

     public int getId() {
         return id;
     }

     public void setId(int id) {
         this.id = id;
     }

     public String getNombre() {
         return nombre;
     }

     public void setNombre(String nombre) {
         this.nombre = nombre;
     }
```

```
        public Integer getPrecio() {
            return precio;
        }

        public void setPrecio(Integer precio) {
            this.precio = precio;
        }
    }
```

- **@Entity**: Indica que es una clase POJO de Entidad (representa una tabla en la base de datos)
- **@Table**: Específica la tabla principal relacionada con la entidad.
    - name – nombre de la tabla, por defecto el de la entidad si no se especifica.
    - catalog – nombre del catálogo.
    - schema – nombre del esquema.
    - uniqueConstraints – constrains entre tablas relacionadas con la anotación @Column y @JoinColumn
- **@Column**: Especifica una columna de la tabla a mapear con un campo de la entidad.
    - name - nombre de la columna.
    - unique - si el campo tiene un único valor.
    - nullable - si permite nulos.
    - insertable - si la columna se incluirá; en la sentencia INSERT generada.
    - updatable - si la columna se incluirá; en la sentencia UPDATE generada.
    - table - nombre de la tabla que contiene la columna.
    - length - longitud de la columna.
    - precision - número de dígitos decimales.
    - scale - escala decimal.
- **@Id**: Indica la clave primaria de la tabla.
- **@GeneratedValue**: Asociado con la clave primaria, indica que ésta se debe generar por ejemplo con una secuencia de la base de datos.
    - strategy – estrategia a seguir para la generación de la clave: AUTO (valor por defecto, el contenedor decide la estrategia en función de la base de datos), IDENTITY (utiliza un contador, ej: MySQL), SEQUENCE (utiliza una secuencia, ej: Oracle, PostgreSQL) y TABLE (utiliza una tabla de identificadores).
    - generator – forma en la que genera la clave.

7. Abrir y estudiar la interfaz IProductoDao, que contiene el contrato para el CRUD:
   /src/main/java/com.bolsadeideas.ejemplos.models.dao/IProductoDao.java

```
package com.bolsadeideas.ejemplos.models.dao;

import java.util.List;

import com.bolsadeideas.ejemplos.models.entity.Producto;

/**
 *
 * @author Andres Guzman F
 */
public interface IProductoDao {

    public List<Producto> findAll();

    public Producto findById(int productoId);

    public void save(Producto producto);

    public void delete(Producto producto);
}
```

8. Abrir y estudiar la clase ProductoDaoHibernateTemplate, esta clase implementa la interfaz Dao usando Hibernate con Spring:

   /src/main/java/com.bolsadeideas.ejemplos.models.dao/ProductoDaoHibernateTemplate.java

```
package com.bolsadeideas.ejemplos.models.dao;

import java.util.List;
import com.bolsadeideas.ejemplos.models.entity.Producto;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.orm.hibernate3.HibernateTemplate;
import org.springframework.stereotype.Repository;

@Repository("productoDaoHibernateTemplate")
public class ProductoDaoHibernateTemplate implements IProductoDao {

    @Autowired
    private HibernateTemplate hibernateTemplate;

    @Override
    public List<Producto> findAll() {
        return hibernateTemplate.find("from Producto");
    }

    @Override
    public Producto findById(int productoId) {
        return (Producto) hibernateTemplate.get(Producto.class, productoId);
    }

    @Override
    public void save(Producto producto) {
      hibernateTemplate.saveOrUpdate(producto);
    }

    @Override
    public void delete(Producto producto) {
      hibernateTemplate.delete(producto);
    }
}
```

- Spring nos provee la clase **HibernateTemplate** para brindarle a nuestros DAO soporte para Hibernate de forma más sencilla y directa.
- **HibernateTemplate**, en particular, contiene varios métodos útiles, tales como guardar, borrar, listar y obtener los objetos de la base de datos que simplifican el uso de Hibernate.
- Estos métodos suelen encapsular varias excepciones propias de acceso a datos de Hibernate (y SQL) dentro de una DataAccessException (que hereda de RuntimeException). Internamente manipula el SessionFactory la session de Hibernate.

9. Abrir y estudiar la interfaz IProductoService, que contiene el contrato para el CRUD del Servicio:

   /src/main/java/com/bolsadeideas/ejemplos/models/services/IProductoService.java

```
package com.bolsadeideas.ejemplos.models.services;

import java.util.List;

import com.bolsadeideas.ejemplos.models.entity.Producto;
/**
 *
 * @author Andres Guzman F
 */
public interface IProductoService {

    public List<Producto> findAll();

    public Producto findById(int productoId);

    public void save(Producto producto);

    public void delete(Producto producto);
}
```

10. Abrir y estudiar la clase ProductoServiceImpl, esta clase implementa la interfaz Service, la clase va anotada con la anotación de Spring **@Service**.
/src/main/java/com/bolsadeideas/ejemplos/models/services/ProductoServiceImpl.java

```
package com.bolsadeideas.ejemplos.models.services;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.bolsadeideas.ejemplos.models.dao.IProductoDao;
import com.bolsadeideas.ejemplos.models.entity.Producto;

@Service("productoService")
public class ProductoServiceImpl implements IProductoService {

	@Autowired
	@Qualifier("productoDaoHibernateTemplate")
	private IProductoDao productoDao;

	@Override
	@Transactional(readOnly = true)
	public List<Producto> findAll() {
		return productoDao.findAll();
	}

	@Override
	@Transactional(readOnly = true)
	public Producto findById(int productoId) {
		return productoDao.findById(productoId);
	}

	@Override
	@Transactional
	public void save(Producto producto) {
		productoDao.save(producto);
	}

	@Override
	@Transactional
	public void delete(Producto producto) {
		productoDao.delete(producto);
	}
}
```

- Se inyecta una instancia del **ProductoDao** mediante la anotación **@Autowired**.
- Notamos que maneja transacción en los métodos del servicio usando la anotación **@Trasaction**

11. **Doble clic** en la clase **ProductoController.java** La clase controladora es completamente multi-acción, mediante la anotación **@Controller** identifica que es un **Bean** o **Componente** de **Spring,** y el controlador será **Multiacción** si detecta que hay más de un método anotado con **@RequestMapping**, nos permite tener varios métodos y cada uno responde a cierta petición de usuario independiente unas de otras, cada una ejecuta/procesa/controla una petición de usuario HTTP, en el ejercicio del ejemplo tenemos un típico controlador CRUD (Crear, Obtener, Actualizar y Borrar), en este caso un mantenedor de productos, donde tenemos diferentes métodos de acción para cada tipo de operación: listar, ver detalle, crear, editar y eliminar.

/src/main/java/com.bolsadeideas.ejemplos.controllers/ProductoController.java

```
package com.bolsadeideas.ejemplos.controllers;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.bind.support.SessionStatus;
import javax.validation.Valid;
import com.bolsadeideas.ejemplos.models.services.IProductoService;
import com.bolsadeideas.ejemplos.models.entity.Producto;

@Controller
@RequestMapping("/catalogo")
@SessionAttributes("producto")
public class ProductoController {

    @Autowired
    private IProductoService productoService;

    @RequestMapping(value = "/listado.htm", method = RequestMethod.GET)
    @ModelAttribute("productos")
    public List<Producto> listado(Model modelo) throws Exception {
        modelo.addAttribute("titulo", "Listado de Productos");

        return productoService.findAll();
    }
```

```
        @RequestMapping(value = "/form.htm", method = RequestMethod.GET)
        public String setupForm(@RequestParam(value = "id", required = false,
             defaultValue = "0") int id, Model model) {
            Producto producto = null;
            if (id > 0) {
                producto = productoService.findById(id);
            } else {
                producto = new Producto();
            }

            model.addAttribute("producto", producto);
            return "form";
        }

        @RequestMapping(value = "/form.htm", method = RequestMethod.POST)
        public String processSubmit(@Valid Producto producto,
                BindingResult result, SessionStatus status) {

            new ProductoValidator().validate(producto, result);
            if (result.hasErrors()) {
                return "form";
            } else {
                productoService.save(producto);
                status.setComplete();
                return "redirect:listado.htm";
            }
        }

        @RequestMapping(value = "/eliminar.htm", method = RequestMethod.GET)
        public String eliminar(@RequestParam("id") int id) {
            Producto producto = productoDao.findById(id);

            if (null != producto) {
                productoService.delete(producto);
            }

            return "redirect:listado.htm";
        }

        @ModelAttribute("titulo")
        public String populateTitulo() {
            return "Formulario Producto";
        }
    }
```

- Observamos que el controlador es marcado con la anotación **@Controller** y además está marcado con la anotación **@RequestMapping** con un valor que especifica el URL Path, es decir el Path URL “/catalogo/” será el primer nivel en la URL.

- La sesión del objeto comando del formulario **producto** la configuramos con **@SessionAttributes("producto")** para que mantenga su estado persistente entre las peticiones y validaciones
- Notamos que el atributo IProductoService **productoService** está anotado con **@Autowired** , le informa a Spring, que deberá inyectar automáticamente la dependencia del objeto beans **productoService** en el objeto controlador, sin la necesidad de configurar en el XML, basado en el tipo definido como Autowired.
- Los métodos de acción **de la peticiones**, están anotados con **@RequestMapping** mediante el atributo **value**, dicho valor tendrá que ver con el URL al que tendremos que dirigirnos para ejecutar el método del Controller, en otras palabras son mapeada a un Path URL.
- Los métodos de acción del ejemplo reciben por argumento el objeto Model para asignar valores a la vista, también podemos asignar parámetros o atributos a la vista utilizando la anotación **@ModelAttribute("nombreParametro")**, ejemplo **@ModelAttribute("titulo")**. Entonces resumiendo, cuando usamos la anotación **@ModelAttribute** sobre un método es para pasar a la vista JSP el objeto que se retorna por dicho método anotado, es decir es una forma de pasar valores o datos a la vista, desde el controlador a la jsp usando anotaciones
- Observamos que algunos métodos de acción **setupForm , proccessSubmit y eliminar** están retornando un String, éste corresponde al nombre de la vista JSP (que se encuentra en WEB-INF/views) o bien para hacer una redirección a otro recurso web o url usando “redirect:URL”.
- El método **processSubmit**, se encarga de procesar el formulario, por lo tanto se debe especificar el método de la petición como POST **RequestMethod.POST**, para procesar los datos enviados por un formulario, recibe los datos de la petición (POST) poblados en el objeto comando **producto**, el objeto **result**, que contiene el objeto **errors** para la validación, luego se valida pasando el objeto comando y el **result**, si falla la validación retornará de vuelta al formulario de lo contrario guardara el producto a la base de datos.

12. Abrir y estudiar el archivo applicationContext-hibernate.xml.
    /src/main/webapp/WEB-INF/spring/root-context.xml.
    - Recordemos que el archivo XML **root-context.xml** es para configurar el **dataSources** o las **conexiones a la base de datos**, configuraciones de **ORM** (**Session Factory de Hibernate**) y los **objetos relacionados al modelo** y negocio como objetos de dominio, **DAO (objeto de acceso a datos**.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- Escanea o busca en el package base de la aplicación clases beans anotados
        con @Components, @Repository, @Service -->
    <context:component-scan base-package="com.bolsadeideas.ejemplos.models" />

    <!-- Activa las anotaciones para ser detectadas en las clases bean de Spring:
        @Required y @Autowired, @Resource. -->
    <context:annotation-config />

    <!-- Root Context: defines shared resources visible to all other web components -->
    <import resource="appServlet/applicationContext-hibernate.xml" />
</beans>
```

- La declaración XML **context:component-scan** se encargará de escanear las clases del modelo o accesos a datos que contengan las anotaciones (@Service, @Repository, etc) indicando el packages (paquetes) que se le indique, en ese packages deberían estar todas nuestras clases del modelo o lógica de negocio/accesos a datos.

13. Abrir y estudiar el archivo applicationContext-hibernate.xml.
/src/main/webapp/WEB-INF/spring/appServlet/applicationContext-hibernate.xml.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context" xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans.xsd
                        http://www.springframework.org/schema/context
                        http://www.springframework.org/schema/context/spring-context.xsd
                        http://www.springframework.org/schema/tx
                        http://www.springframework.org/schema/tx/spring-tx.xsd">
    <!-- ======================== DEFINICION HIBERNATE ======================== -->
    <!-- importamos las definicion del dataSource -->
    <import resource="applicationContext-dataSource.xml" />

    <!-- Configuracion que reemplaza los placeholders ${...} placeholders con
        los valores del archivo properties -->
    <!-- (En este caso, los datos relacionados a la conexion JDBC para el DataSource) -->
    <context:property-placeholder location="classpath:jdbc.properties" />

    <!-- Hibernate SessionFactory, para anotaciones en las clases entity usamos AnnotationSessionFactoryBean -->
    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="annotatedClasses">
            <list>
                <value>com.bolsadeideas.ejemplos.models.entity.Producto</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">${hibernate.dialect}</prop>
                <prop key="hibernate.show_sql">${hibernate.show_sql}</prop>
                <prop key="hibernate.generate_statistics">${hibernate.generate_statistics}</prop>
            </props>
        </property>
    </bean>

    <!-- HibernateTemplate -->
    <bean id="hibernateTemplate" class="org.springframework.orm.hibernate4.HibernateTemplate">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <!-- Transaction manager para un solo SessionFactory (es una alternativa
        a JTA) -->
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate4.HibernateTransactionManager"
        p:sessionFactory-ref="sessionFactory" />

    <!-- Instrucción de Spring para activar y manejar el transaction management de forma automática
        sobre las clases anotadas con @Repository. -->
    <tx:annotation-driven />

</beans>
```

- El **bean** **productoDao** se define y registra en el contexto de Spring mediante la anotación @Repository, por lo que no es necesario regístralo en el XML, la clase **com.formacionbdi.spring.webmvc.catalogo.models.dao.ProductoDao** tiene implementado las operaciones básicas de la base de datos usando **HibernateTemplate de Spring** (Integración Hibernate), cumpliendo con el contrato de implementación de la interface ***IProductoDao***.
- Tenemos el bean **propertyConfigurer** que nos permite configurar en el archivo properties "**jdbc.properties**" los datos de conexión a la base de datos con JDBC.
- El archivo **jdbc.properties**, guardado bajo el directorio **src/main/webapp/WEB-INF/classes/jdbc.properties** conteniendo la configuración del driver (en nuestro caso HSQL DB), el string de conexión o url y las credenciales de acceso usuario y clave de la base de datos:

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:mem:estudiante
jdbc.username=sa
jdbc.password=
```

- Después tenemos el bean **dataSource**, configurado en el archivo **applicationContext-dataSource.xml**, mapeado a la clase de spring ***org.apache.commons.dbcp2.BasicDataSource***, que será inyectada en el bean **sessionFactory de Hibernate** para la conexión.
- **SessionFactory** mapeada a la clase ***org.springframework.orm.hibernate4.LocalSessionFactoryBean***, este objeto de Hibernate maneja las sesiones que ejecutan las distintas transacciones a la base de datos, contiene los archivos mapping en el atributo **annotatedClasses** y además se le inyecta el objeto dataSource.
- Finalmente tenemos el bean que maneja las transacciones automáticas de spring, necesario para trabajar con la integración hibernate mediante **HibernateTemplate**.
- Más adelante el DAO será inyectado, como veremos mediante la anotación **@Autowired**.

14. Abrir y estudiar jdbc.properties. Archivo de configuración con los datos de conexión - Es referenciado e incluido en el archivo applicationContext-dataSource.xml.
/src/main/webapp/WEB-INF/classes/jdbc.properties

```
# Properties file with JDBC and JPA settings.
#
# Applied by <context:property-placeholder location="jdbc.properties"/> from
# various application context XML files (e.g., "applicationContext-*.xml").
# Targeted at system administrators, to avoid touching the context XML files.

#-------------------------------------------------------------------------------
# HSQL Settings

jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:mem:estudiante
jdbc.username=sa
jdbc.password=

# Properties that control the population of schema and data for a new data source
jdbc.initLocation=classpath:db/hsqldb/initDB.txt
jdbc.dataLocation=classpath:db/hsqldb/populateDB.txt

#-------------------------------------------------------------------------------
# MySQL Settings

#jdbc.driverClassName=com.mysql.jdbc.Driver
#jdbc.url=jdbc:mysql://localhost:3306/test
#jdbc.username=root
#jdbc.password=

# Properties that control the population of schema and data for a new data source
#jdbc.initLocation=classpath:db/mysql/initDB.txt
#jdbc.dataLocation=classpath:db/mysql/populateDB.txt
```

15. Abrir y estudiar el script SQL initDB.txt. Es referenciado e incluido en el archivo jdbc.properties.
/src/main/webapp/WEB-INF/classes/db/hsqldb/initDB.txt

```
/* Create tables */
CREATE TABLE productos(
id INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
nombre VARCHAR(200) NOT NULL,
precio INTEGER NOT NULL,
cantidad INTEGER NOT NULL
);
```

16. Abrir y estudiar el script SQL populateDB.txt. Es referenciado e incluido en el archivo jdbc.properties.
/src/main/webapp/WEB-INF/classes/db/hsqldb/populateDB.txt

```
/* Populate tables */
INSERT INTO productos (id, nombre, precio, cantidad) VALUES(1, 'Panasonic Pantalla LCD', 259990, 20);
INSERT INTO productos (id, nombre, precio, cantidad) VALUES(2, 'Sony Cámara digital DSC-W320B', 123490,
12);
INSERT INTO productos (id, nombre, precio, cantidad) VALUES(3, 'Apple iPod shuffle', 1499990, 25);
INSERT INTO productos (id, nombre, precio, cantidad) VALUES(4, 'Sony Notebook Z110', 37990, 10);
INSERT INTO productos (id, nombre, precio, cantidad) VALUES(5, 'Hewlett Packard Multifuncional F2280',
69990, 7);
INSERT INTO productos (id, nombre, precio, cantidad) VALUES(6, 'Bianchi Bicicleta Aro 26', 69990, 5);
INSERT INTO productos (id, nombre, precio, cantidad) VALUES(7, 'Mica Cómoda 5 Cajones', 299990, 20);
```

AUTOR: Andrés Guzmán Fontecilla.
Email: andresguzf@gmail.com
WEB: http://www.bolsadeideas.com

17. Abrir y estudiar la vista **listado.jsp**.
    src/main/webapp/WEB-INF/views/catalogo/listado.jsp

```
<%@page contentType="text/html;charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

...Etc...

<table style="width: 700px;"
    class="table table-striped table-hover table-bordered">
    <thead>
        <tr>
            <th>#</th>
            <th>Nombre</th>
            <th>Precio</th>
            <th>Cantidad</th>
            <th>Editar</th>
            <th>Eliminar</th>
        </tr>
    </thead>
    <tbody>
        <c:forEach items="${productos}" var="producto">
            <tr>
                <td><c:out value="${producto.id}" /></td>
                <td><c:out value="${producto.nombre}" /></td>
                <td><c:out value="${producto.precio}" /></td>
                <td><c:out value="${producto.cantidad}" /></td>
                <td><a class="btn-xs btn-primary"
                    href="<c:url value="/catalogo/form.htm?id=${producto.id}"/>">
                       editar</a></td>
                <td><a class="btn-xs btn-danger" onclick="return confirm('Esta seguro?');"
                    href="<c:url value="/catalogo/eliminar.htm?id=${producto.id}"/>">
                   eliminar</a></td>
            </tr>
        </c:forEach>
    </tbody>
</table>
```

- Observamos que tenemos links hacia el formulario "**form.htm**" tanto para crear y para editar cada producto pasando el parámetro id.
- Observamos que en el elemento iterador **forEach** accedemos al listado mediante **items="${productos}"**, internamente hará una llamada al listado de productos guardado en el objeto Model, el que fue asignado a la vista desde el controlador en el método **listado(Model modelo)** mediante la anotación **@ModelAttribute("productos")**.

AUTOR: Andrés Guzmán Fontecilla.
Email: andresguzf@gmail.com
WEB: http://www.bolsadeideas.com

- Ahora bien, si queremos ser capaces de visualizar, actualizar y eliminar registros de producto pre-existentes en la base de datos, necesitamos pasar el identificador (o id) por URL al controlador. Para hacer esto usamos parámetros GET URL **?id=${producto.id}**. Ahora con esto, contamos con nuestros links para crear nuevo producto, para editar a cada uno y eliminar.
- En el link Editar pasamos como parámetro de la URL el id del producto con el nombre: **id** y valor **${producto.id}**.
- La vista del formulario **form.jsp** se re-utiliza tanto para crear y modificar, y es lo que veremos a continuación, en la siguiente página.

18. Abrir y estudiar la vista **form.jsp**.
    src/main/webapp/WEB-INF/views/catalogo/form.jsp

```
<%@page contentType="text/html;charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
...Etc...

<form:form modelAttribute="producto" method="post"
    cssClass="form-horizontal" role="form">
    <div class="form-group">
        <form:label for="nombre" path="nombre"
            cssClass="col-sm-2 control-label">Nombre</form:label>
        <div class="col-sm-10">
            <form:input path="nombre" style="width: 300px;" cssClass="form-control"
                cssErrorClass="form-control alert-danger" />
            <form:errors path="nombre" />
        </div>
    </div>
    <div class="form-group">
        <form:label for="cantidad" path="cantidad"
            cssClass="col-sm-2 control-label">Cantidad</form:label>
        <div class="col-sm-10">
            <form:input path="cantidad" style="width: 300px;" cssClass="form-control"
                cssErrorClass="form-control alert-danger" />
            <form:errors path="cantidad" />
        </div>
    </div>
    <div class="form-group">
        <form:label for="precio" path="precio"
            cssClass="col-sm-2 control-label">Precio</form:label>
        <div class="col-sm-10">
            <form:input path="precio" style="width: 300px;" cssClass="form-control"
                cssErrorClass="form-control alert-danger" />
            <form:errors path="precio" />
        </div>
    </div>
    <div class="form-group">
        <div class="col-sm-offset-2 col-sm-10">
          <input type="submit" value="Crear Producto" class="btn btn-primary" role="button" />
        </div>
    </div>
</form:form>
```

- Accedemos al objeto **producto** gracias al objeto Model, que fue asignado a la vista en el controlador, dentro del método **setupForm()** mediante el objeto Model: **model.addAttribute("producto", producto)**.

Librerías y dependencias en el pom.xml de Hibernate

Las librerías que necesitamos agregar en el pom.xml para un proyecto Spring Web MVC con **Hibernate** son las relacionadas con Hibernate (ORM bases de datos), y serían las siguientes:

```
Etc…
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <!-- Hibernate -->
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>4.3.11.Final</version>
        </dependency>
        <!-- JDBC drivers -->
        <dependency>
            <groupId>org.hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>2.3.3</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.36</version>
        </dependency>
        <!-- DBCP2 -->
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-dbcp2</artifactId>
            <version>2.1</version>
        </dependency>
etc ......
```

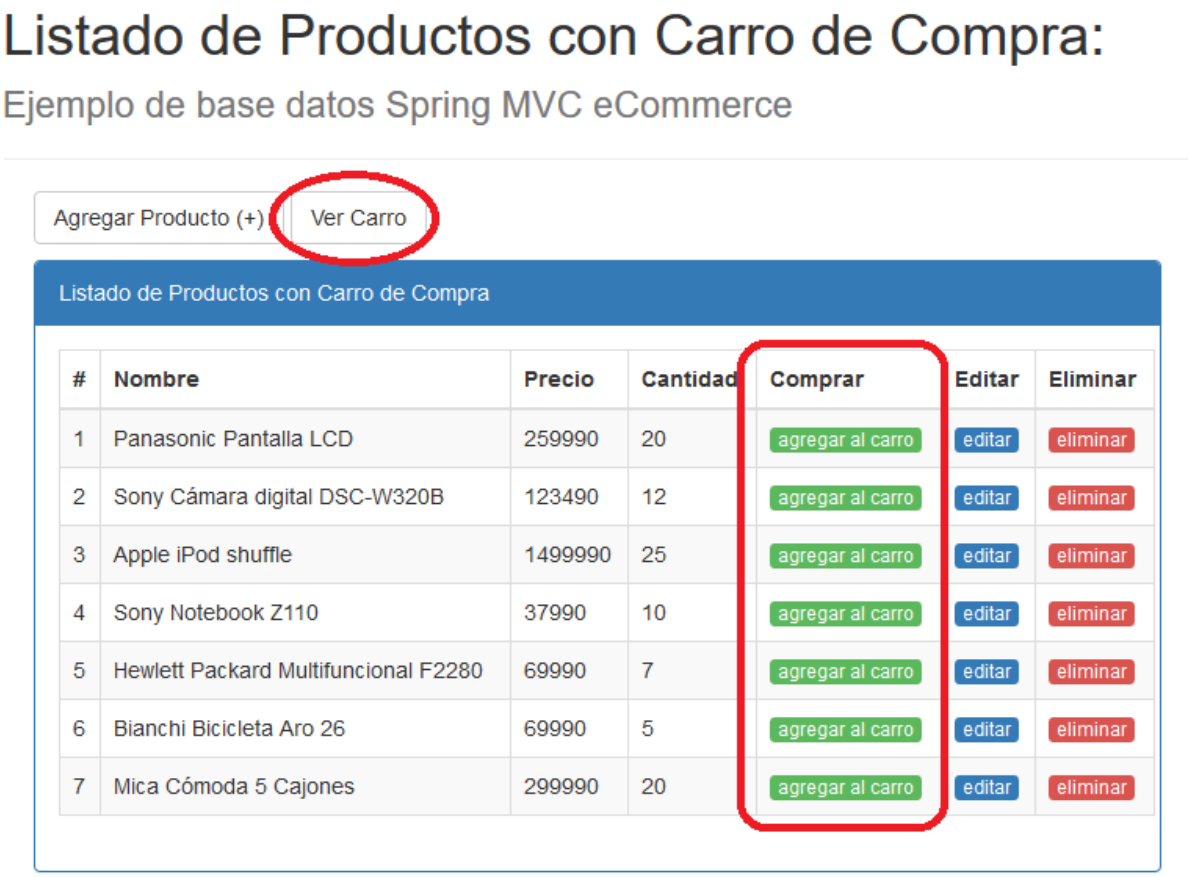
Todas las que están marcadas en rojo son propias de Hibernate, el resto son dependencias de Spring para la integración con Hibernate y librerías externas o de terceros requeridas por Hibernate, por ejemplo hsqldb que es el motor de base de datos, si usáramos mysql se cambia por la librería de mysql. Otra dependencia importante es el commons-dbcp para manejar un data source con pool de conexiones en hibernate.

Ejercicio 2: Generar y ejecutar el ejemplo "Carro de compras con Spring MVC"

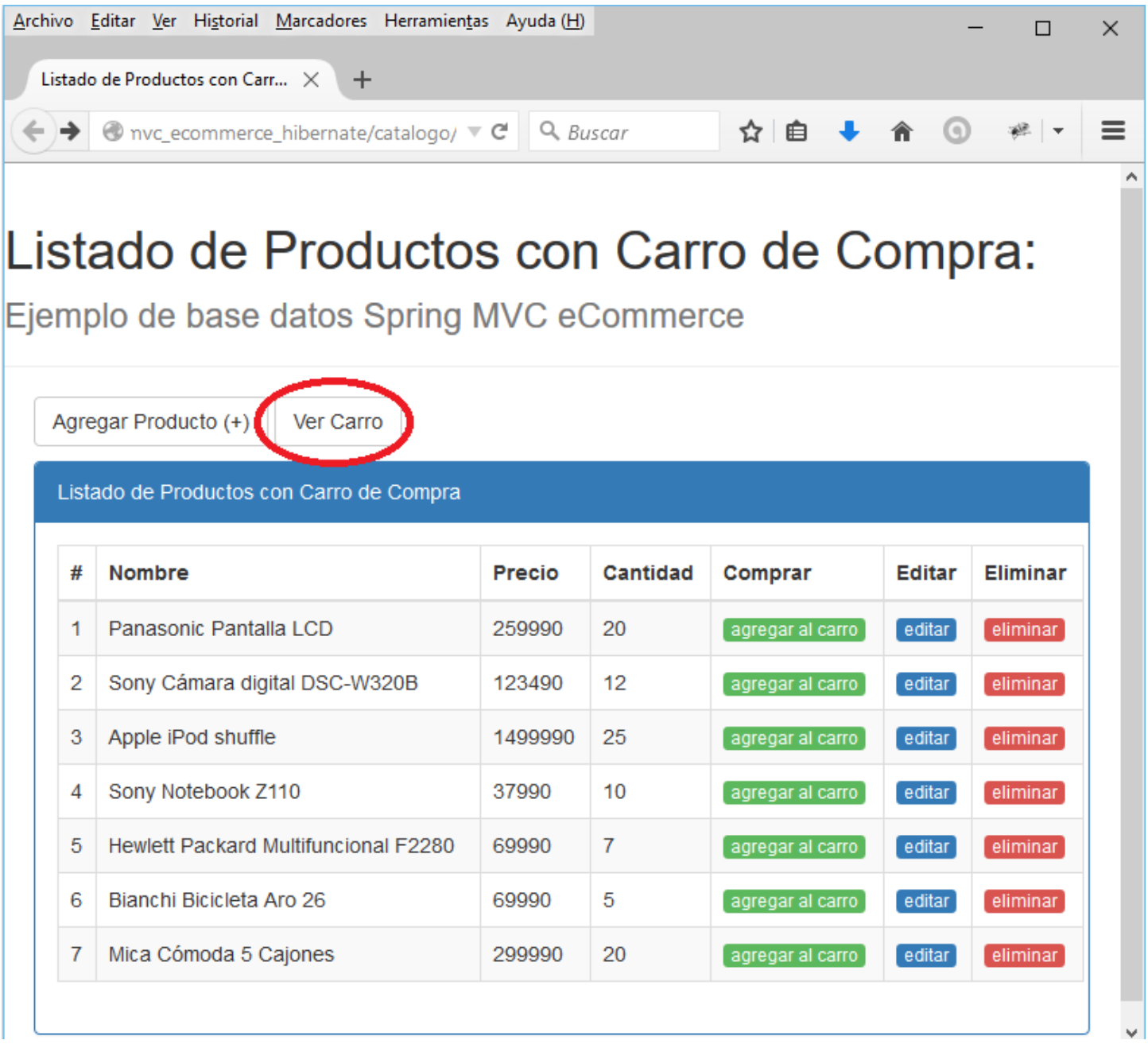
En este ejemplo no pretendemos desarrollar una gran ingeniería del software, pero sí desmenuzar de una forma directa los pasos a seguir en el desarrollo de nuestra aplicación.

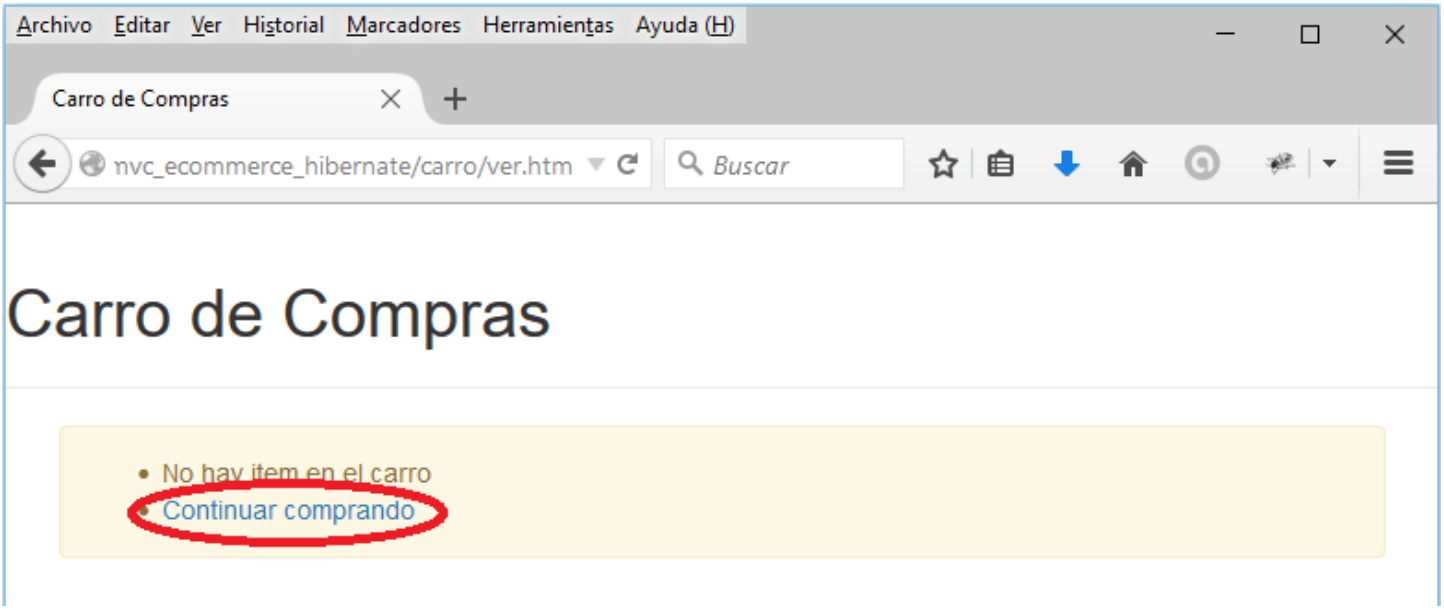
Gracias a la arquitectura MVC, podríamos desarrollar por un lado la interfaz del cliente Web, y delegar en otro equipo de desarrollo la parte de la lógica de negocio. En nuestro caso primero desarrollaremos la parte de negocio, para terminar implementando la capa de presentación y control (más que nada, por claridad en el desglose de los conceptos y su aplicación práctica).

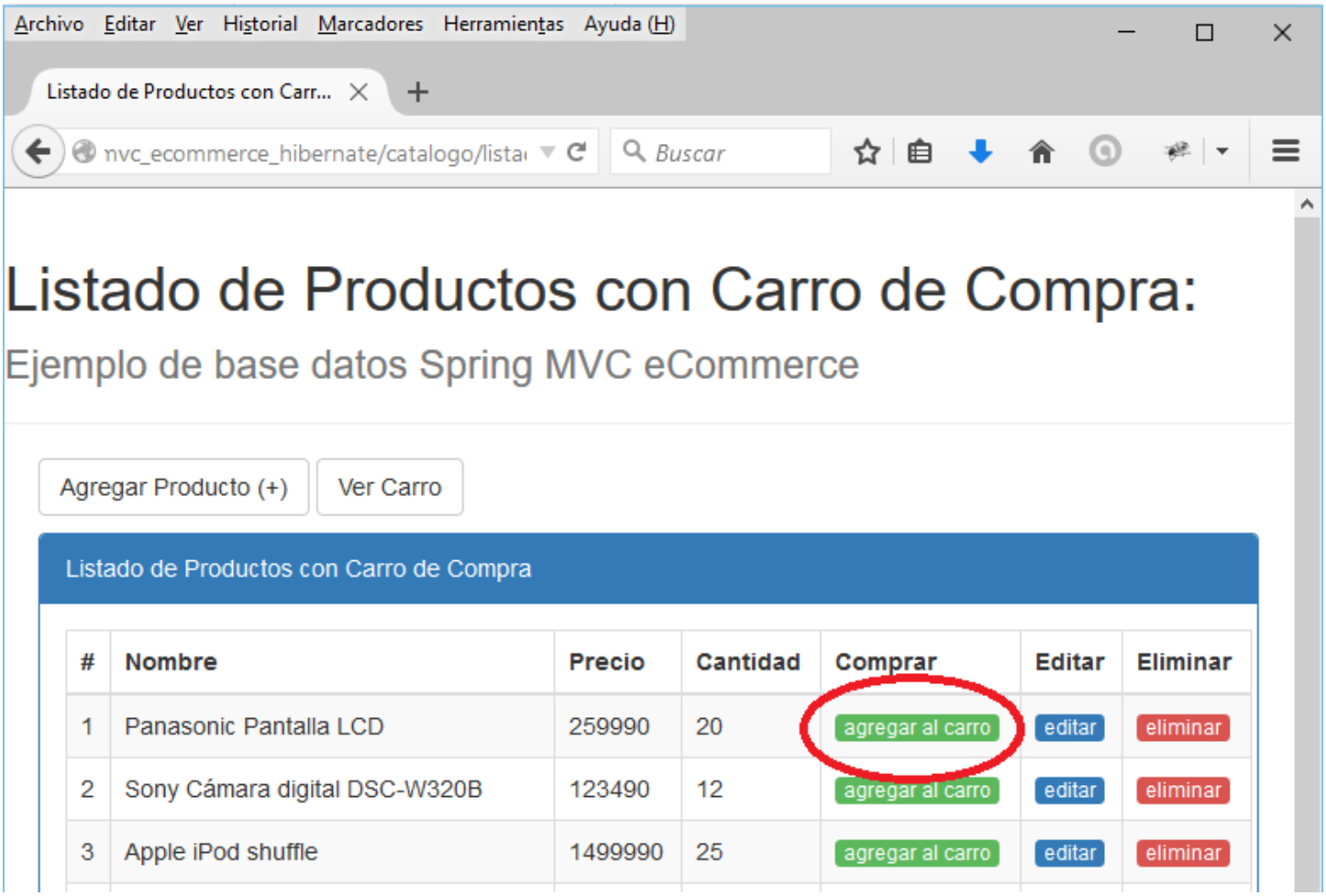
1. Clic derecho sobre el proyecto **Run As->Maven Clean** y **Run As->Maven Install**.
2. Clic derecho sobre el proyecto y **Maven->Update Project…**
3. Clic derecho sobre **basedatos_springmvc_ecommerce_hibernate-> Run As on Server**
4. Observe el resultado en el navegador

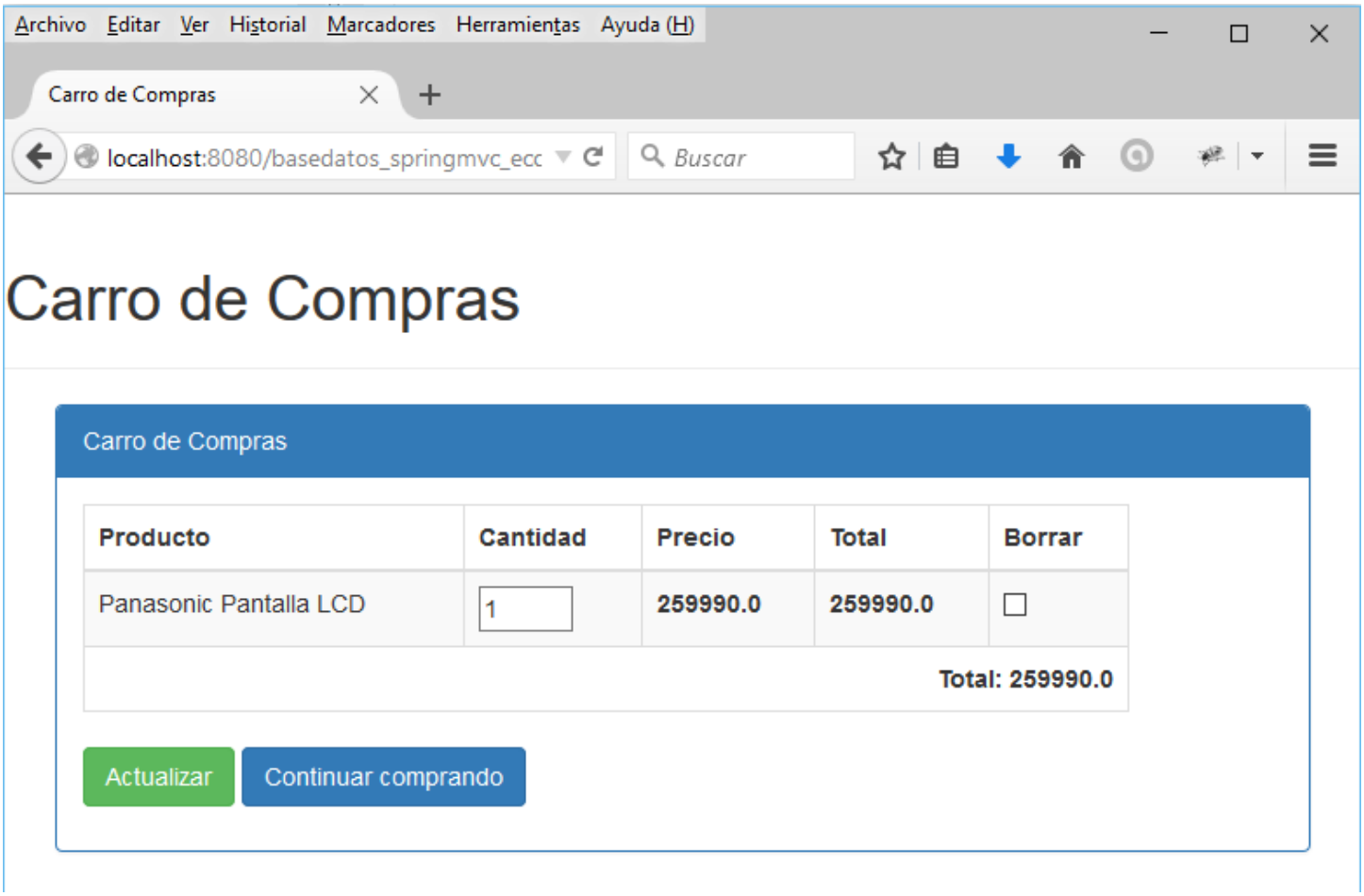


AUTOR: Andrés Guzmán Fontecilla.
Email: andresguzf@gmail.com
WEB: http://www.bolsadeideas.com


Archivo Editar Ver Historial Marcadores Herramientas Ayuda (H)
Listado de Productos con Carr...
nvc_ecommerce_hibernate/catalogo/
Buscar
Listado de Productos con Carro de Compra:
Ejemplo de base datos Spring MVC eCommerce
Agregar Producto (+)
Ver Carro
Listado de Productos con Carro de Compra
Nombre Precio Cantidad Comprar Editar Eliminar
1 Panasonic Pantalla LCD 259990 20 agregar al carro editar eliminar
2 Sony Cámara digital DSC-W320B 123490 12 agregar al carro editar eliminar
3 Apple iPod shuffle 1499990 25 agregar al carro editar eliminar
4 Sony Notebook Z110 37990 10 agregar al carro editar eliminar
5 Hewlett Packard Multifuncional F2280 69990 7 agregar al carro editar eliminar
6 Bianchi Bicicleta Aro 26 69990 5 agregar al carro editar eliminar
7 Mica Cómoda 5 Cajones 299990 20 agregar al carro editar eliminar


Archivo Editar Ver Historial Marcadores Herramientas Ayuda (H)
Carro de Compras
nvc_ecommerce_hibernate/carro/ver.htm
Buscar
Carro de Compras
No hay item en el carro
Continuar comprando


Archivo Editar Ver Historial Marcadores Herramientas Ayuda (H)
Listado de Productos con Carr...
nvc_ecommerce_hibernate/catalogo/lista
Buscar
Listado de Productos con Carro de Compra:
Ejemplo de base datos Spring MVC eCommerce
Agregar Producto (+)
Ver Carro
Listado de Productos con Carro de Compra
Nombre Precio Cantidad Comprar Editar Eliminar
1 Panasonic Pantalla LCD 259990 20 agregar al carro editar eliminar
2 Sony Cámara digital DSC-W320B 123490 12 agregar al carro editar eliminar
3 Apple iPod shuffle 1499990 25 agregar al carro editar eliminar


Archivo Editar Ver Historial Marcadores Herramientas Ayuda (H)
Carro de Compras
localhost:8080/basedatos_springmvc_ecc
Buscar
Carro de Compras
Carro de Compras
Producto
Cantidad
Precio
Total
Borrar
Panasonic Pantalla LCD
1
259990.0
259990.0
Total: 259990.0
Actualizar
Continuar comprando

5. Abrir y estudiar el archivo applicationContext-hibernate.xml. **/src/main/webapp/WEB-INF/spring/root-context.xml.**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
    http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- Escanea o busca en el package base de la aplicación clases beans anotados
        con @Components, @Repository, @Service -->
    <context:component-scan base-package="com.bolsadeideas.ejemplos.models" />

    <!-- Activa las anotaciones para ser detectadas en las clases bean de Spring:
        @Required y @Autowired, @Resource. -->
    <context:annotation-config />

    <!-- Root Context: defines shared resources visible to all other web components -->
    <import resource="appServlet/applicationContext-hibernate.xml" />

    <bean id="carro" class="com.bolsadeideas.ejemplos.dominio.carro.Carro"
        scope="session">
        <aop:scoped-proxy />
    </bean>

</beans>
```

- Notamos que declaramos el bean de session de spring **Carro**, el cual su contexto o scope es de session HTTP y además de declara un proxy el cual interceptará mediante AOP (Programación Orientada a Aspectos) al bean Carro para que sea persistente en una SessionHttp

6. **Doble clic** en la clase **CarroController.java**. La clase controladora es completamente multi-acción, mediante la anotación **@Controller** identifica que es un **Bean** o **Componente** de **Spring,** implementamos el carro de compras controller multi-acción.
    - Observamos cómo se inyecta el **bean carro de session** con **@Autowired**

/src/main/java/com.bolsadeideas.ejemplos.controllers/CarroController.java

```
package com.bolsadeideas.ejemplos.controllers;

import java.util.ArrayList;
import java.util.Enumeration;
import java.util.List;

import javax.servlet.http.HttpServletRequest;

import com.bolsadeideas.ejemplos.models.services.IProductoService;
import com.bolsadeideas.ejemplos.models.entity.Producto;
import com.bolsadeideas.ejemplos.dominio.carro.ICarro;
import com.bolsadeideas.ejemplos.dominio.carro.ItemCarro;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
@RequestMapping("/carro")
public class CarroController {

  @Autowired
  private IProductoService productoService;

  @Autowired
  private ICarro carro;

  @RequestMapping(value = "/ver.htm", method = RequestMethod.GET)
  public String verCarro(ModelMap model) {
    model.addAttribute("carro", carro);
    model.addAttribute("titulo", "Carro de Compras");
    return "carro/ver";
  }

  @RequestMapping(value = "/agregar.htm", method = RequestMethod.GET)
  public String addCarro(
      @RequestParam("id") int id,
      @RequestParam(value = "cantidad", required = false, defaultValue = "1") int cantidad,
      ModelMap model) {
    // Llamamos al servicio y le pedimos un Producto Entity para el producto elegido.
    Producto producto = productoService.findById(id);

    // Añadimos el producto al carrito.
    carro.addProducto(new ItemCarro(producto, cantidad));
    return "redirect:ver.htm";
  }
```

```
    @RequestMapping(value = "/actualizar.htm", method = RequestMethod.POST)
    public String updateCarro(HttpServletRequest request, ModelMap model) {

      updateProductos(request);
      updateCantidades(request);

      return "redirect:ver.htm";
    }

    private void updateProductos(HttpServletRequest request) {
      String[] deleteIds = request.getParameterValues("deleteProductos");

      if (deleteIds != null && deleteIds.length > 0) {
        int size = deleteIds.length;
        List<String> productoIds = new ArrayList<String>();

        for (int i = 0; i < size; i++) {
          productoIds.add(deleteIds[i]);
        }

        // Obtenemos el carrito del UserContainer y boramos los productos.
        carro.removeProductos(productoIds);
      }

    }

    private void updateCantidades(HttpServletRequest request) {

      Enumeration enumer = request.getParameterNames();

      // Iteramos a traves de los parÃ¡metros y buscamos los que empiezan con
      // "cant_". El campo cant en la vista fueron nombrados "cant_" +
      // productoId.
      // Obtenemos el id de cada producto y su correspondiente cantidad ;-).
      while (enumer.hasMoreElements()) {
        String paramName = (String) enumer.nextElement();
        if (paramName.startsWith("cant_")) {
          String id = paramName.substring(5, paramName.length());
          String qtyStr = request.getParameter(paramName);
          if (id != null && qtyStr != null) {
            carro.updateCantidad(id, Integer.parseInt(qtyStr));
          }
        }
      }
    }
  }
```

7. **Doble clic** en la vista JSP **ver.jsp** (**ver carro**)
   basedatos_springmvc_ecommerce_hibernate/src/main/webapp/WEB-INF/views/carro/ver.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

<!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=ISO-8859-1">
        <title><c:out value="${titulo}" /></title>
    </head>
    <body>
        <h3><c:out value="${titulo}" /></h3>

        <c:choose>
            <c:when test="${carro.size > 0}">
                <form name="formcarro" action="actualizar.htm" method="post">
                    <table cellspacing="0" cellpadding="0" border="0" style="width: 50%">
                        <thead>
                            <tr>
                                <th>Producto</th>
                                <th>Cantidad</th>
                                <th>Precio</th>
                                <th>Total</th>
                                <th>Borrar</th>
                            </tr>
                        </thead>
                        <tbody>
                            <c:forEach items="${carro.items}" var="item">
                                <tr>
                                    <td><c:out value="${item.nombre}" /></td>
                                    <td><input type="text" maxlength="4" size="4" name="cant_<c:out
value="${item.id}" />" value="<c:out value="${item.cantidad}" />" /></td>
                                    <td><strong><c:out value="${item.basePrecio}" /></strong></td>
                                    <td><strong><c:out value="${item.importe}" /></strong></td>
                                    <td><input type="checkbox" value="<c:out value="${item.id}" />"
name="deleteProductos" /></td>
                                </tr>
                            </c:forEach>
                            <tr>
                                <td colspan="5" style="text-align: right;"><strong>Total: <c:out
value="${carro.totalPrecio}" /></strong></td>
                            </tr>
                        </tbody>
                    </table>
                    <p>
                        <a href="javascript:document.formcarro.submit();">Actualizar</a>    
                        <a href="<c:url value="/catalogo/listado.htm"/>">Continuar comprando</a>
                    </p>
                </form>
            </c:when>
            <c:otherwise>
                <p>No hay item en el carro</p>
                <p><a href="<c:url value="/catalogo/listado.htm"/>">Continuar comprando</a></p>
            </c:otherwise>
        </c:choose>
    </body>
</html>
```

8. Estudiar la interface **ICarro**
    - Es la interfaz que nos dice que debe hacer nuestras clases que la implementan, nos entregan un contrato de implementación y las clases son las que implementan dicho contrato y nos dicen quiénes son y como lo hacen. En otras palabras **la clase interfaz definen una colección de métodos abstractos sin implementar los** cuales deben de ser implementados por las clases que la implementa, en nuestro caso la clase concreta **Carro** (carro de compras) debe de definir/implementar los métodos.

```
package com.bolsadeideas.ejemplos.dominio.carro;

import java.util.List;

/**
 *
 * @author Andres Guzman
 */
public interface ICarro {

    public void addProducto(ItemCarro newProducto);

    public void setItems(List<ItemCarro> otrosProductos);

    public void setSize(int size);

    public int getSize();

    public void empty();

    public double getTotalPrecio();

    public void removeProducto(String productoId);

    public void removeProductos(List<String> productoIds);

    public void updateCantidad(String productoId, int cantidad);

    public List<ItemCarro> getItems();
}
```

9. Estudiar la clase **Carro.** Implementa la interfaz ICarro
    - Es nuestra clase concreta de carro de compras, es quien define e implementa los métodos del contrato de la interfaz mencionada más arriba. Se ubica dentro del package dominio. Además hace uso (Composición) de un objeto de **colección de items** que veremos más abajo.

```
package com.bolsadeideas.ejemplos.dominio.carro;

import java.io.Serializable;
import java.util.LinkedList;
import java.util.List;

/**
 *
 * @author Andres Guzman
 */
public class Carro implements ICarro, Serializable {

    private List<ItemCarro> items = new LinkedList<ItemCarro>();

    /**
     * Constructor por defecto.
     */
    public Carro() {
    }

    /**
     * Miramos si este producto ya esta en el carrito.
     * Si ya esta simplemente incrementamos su cantidad.
     * Si es un producto que no estaba; lo añadimos al carrito.
     */
    @Override
    public void addProducto(ItemCarro newProducto) {

        ItemCarro carroProducto = findProducto(Integer.toString(newProducto.getId()));

        if (carroProducto != null) {
            carroProducto.setCantidad(carroProducto.getCantidad()
                    + newProducto.getCantidad());
        } else {
            items.add(newProducto);
        }
    }

    /**
     * Añadimos una lista de products.
     */
    @Override
    public void setItems(List<ItemCarro> otrosProductos) {
        items.addAll(otrosProductos);
    }

    /**
     * El tamaño es el de la lista enlazada.
     * Pero hay que implementar este metodo al tratarse de un JavaBean.
     */
    @Override
    public void setSize(int size) {
    }
```

```
@Override
public int getSize() {
    return items.size();
}

@Override
public void empty() {
    items.clear();
}

@Override
public double getTotalPrecio() {
    double total = 0.0;
    int size = items.size();

    for (int i = 0; i < size; i++) {
        total += ((ItemCarro) items.get(i)).getImporte();
    }

    return total;
}

@Override
public void removeProducto(String productoId) {
    ItemCarro producto = findProducto(productoId);

    if (producto != null) {
        items.remove(producto);
    }
}

@Override
public void removeProductos(List<String> productoIds) {
    if (productoIds != null) {
        int size = productoIds.size();

        for (int i = 0; i < size; i++) {
            removeProducto((String) productoIds.get(i));
        }
    }
}

@Override
public void updateCantidad(String productoId, int cantidad) {
    ItemCarro producto = findProducto(productoId);

    if (producto != null) {
        producto.setCantidad(cantidad);
    }
}

@Override
public List<ItemCarro> getItems() {
    return items;
}

private ItemCarro findProducto(String productoId) {
    ItemCarro producto = null;
    int size = getSize();

    for (int i = 0; i < size; i++) {
        ItemCarro carroProducto = (ItemCarro) items.get(i);

        if (productoId.equals(Integer.toString(carroProducto.getId()))) {
            producto = carroProducto;
```

```
                    break;
                }
            }

            return producto;
        }
    }
```

Ahora vamos a explicar brevemente algunos de los principales métodos:

- **addProducto(item):** Agrega items a nuestro carro de compras, lo agrega al objeto de colección, primero valida si el item/producto ya existe. Si existe actualiza su cantidad, de lo contrario lo agrega como nuevo item.
- **updateCantidad(productoId, cantidad):** Busca el objeto ítem en la colección mediante su id y cuando lo encuentra actualiza su cantidad.
- **removeProducto(productId):** Busca el objeto ítem en la colección mediante su id y cuando lo encuentra lo elimina.
- **getTotalPrecio():** Recorre la colección y suma el total (importe p*q), peso y impuesto y los guarda como atributo del objeto Carro.
- **findProducto(productId):** Busca a un ítem/producto en la colección por su id.
- **getSize():** Obtiene el numero de items guardados en la colección.
- **getItems():** Obtenemos la colección de items/productos.
- El resto de los métodos es Coser y Cantar.

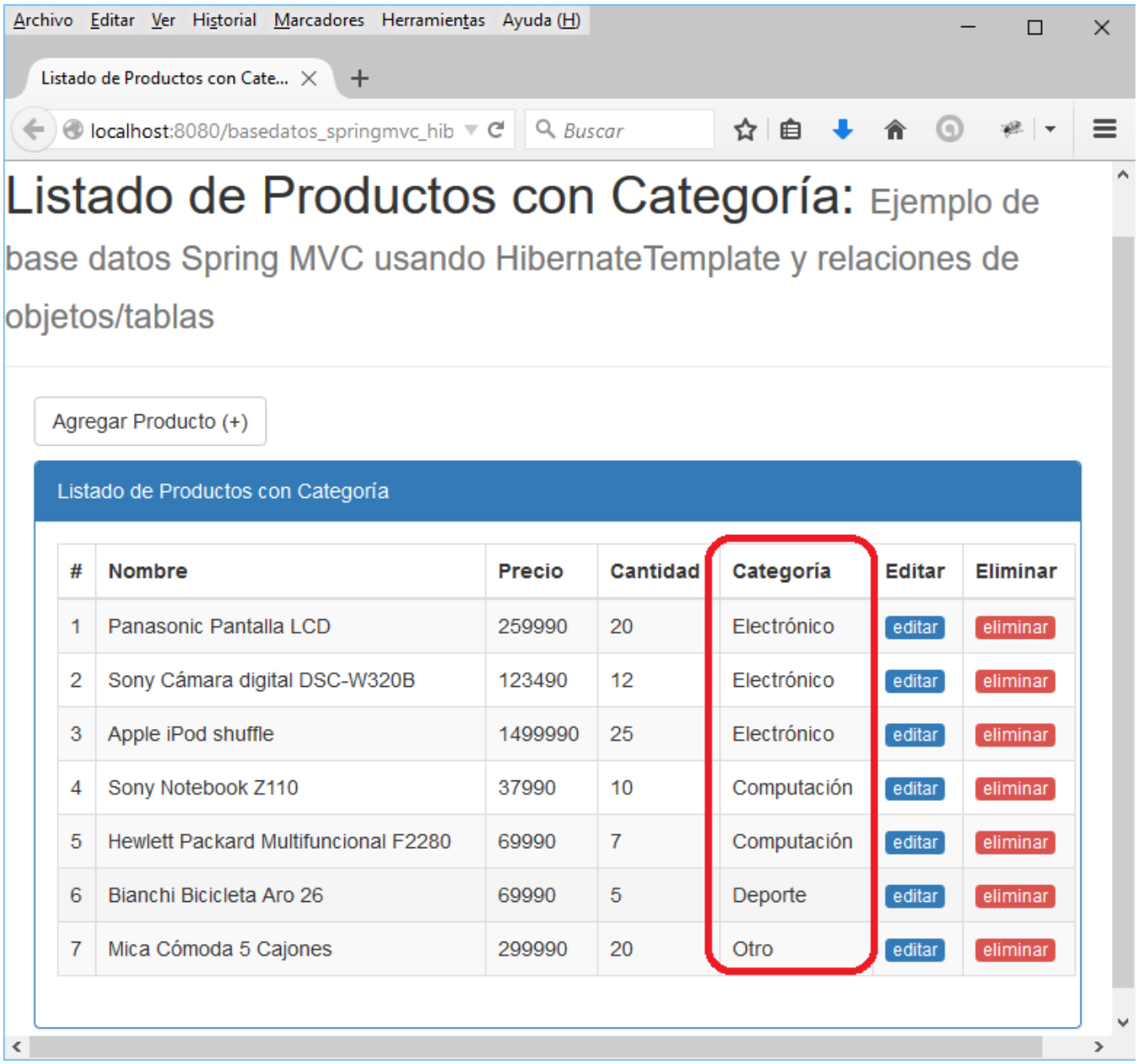
10. Estudiar la clase **ItemCarro,** el detalle de la compra**.**
    - Representa a cada item que ingresamos al carro de compras, es decir serán las **líneas de detalle de nuestro carro de compras** (Clase Carro). Nuestro objeto Item va a contener al objeto Producto (producto) y cantidad del tipo entero que nos indicará el número de cada producto que tenemos almacenado en el carro y nos calculará el importe (P*Q) de cada uno.

```
<
package com.bolsadeideas.ejemplos.dominio.carro;

import java.io.Serializable;
import com.bolsadeideas.ejemplos.models.entity.Producto;

/**
 *
 * @author Andres Guzman
 */
public class ItemCarro implements Serializable {

    /**
     * Importe es el precio de unidad * cantidad.
     */
    private double importe;

    private Producto producto;

    /**
     * Cantidad por defecto a 1.
     */
    private int cantidad = 1;

    /**
     * Constructor por defecto.
     */
    public ItemCarro() {
    }

    /**
     * Constructor de linea a partir de un producto y su cantidad.
     */
    public ItemCarro(Producto producto, int cant) {
        this.producto = producto;
        this.cantidad = cant;
        calcularImporte();
    }

    public void setProducto(Producto newproducto) {
        producto = newproducto;
        calcularImporte();
    }

    public void setImporte(double newPrecio) {
        importe = newPrecio;
    }

    public int getId() {
        return producto.getId();
    }
```

```
    public String getNombre() {
        return producto.getNombre();
    }

    public int getCantidad() {
        return cantidad;
    }

    public void setCantidad(int newCantidad) {
        cantidad = newCantidad;
        calcularImporte();
    }

    public Producto getproducto() {
        return producto;
    }

    public double getBasePrecio() {
        return producto.getPrecio();
    }

    public double getImporte() {
        return importe;
    }

    private void calcularImporte() {
        if (producto.getPrecio() != 0) {
            importe = producto.getPrecio() * getCantidad();
        }
    }
}
```

Ejercicio 3: Ejemplo "Catálogo con Productos y Categorías"

1. Clic derecho sobre el proyecto **Run As->Maven Clean** y **Run As->Maven Install**.
2. Clic derecho sobre el proyecto y **Maven->Update Project…**
3. Clic derecho sobre **basedatos_springmvc_hibernate_categorias**
   - **-> Run As on Server**
4. Observe el resultado en el navegador



5. Abrir y estudiar el script SQL initDB.txt. Es referenciado e incluido en el archivo jdbc.properties.
   /src/main/webapp/WEB-INF/classes/db/hsqldb/initDB.txt
   - Debemos tener la tabla categorías en nuestra base de datos y la relación con la tabla productos

```
/* Create tables */
CREATE TABLE productos(
id INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
categoria_id INTEGER NOT NULL,
nombre VARCHAR(200) NOT NULL,
precio INTEGER NOT NULL,
cantidad INTEGER NOT NULL
);

CREATE TABLE categorias(
id INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
nombre VARCHAR(20)     NOT NULL
);
```

6. Abrir y estudiar el script SQL populateDB.txt. Es referenciado e incluido en el archivo jdbc.properties.
   /src/main/webapp/WEB-INF/classes/db/hsqldb/populateDB.txt

```
/* Populate tables */
INSERT INTO categorias (id, nombre) VALUES (1, 'Electrónico');
INSERT INTO categorias (id, nombre) VALUES (2, 'Libro');
INSERT INTO categorias (id, nombre) VALUES (3, 'Ropa');
INSERT INTO categorias (id, nombre) VALUES (4, 'Deporte');
INSERT INTO categorias (id, nombre) VALUES (5, 'Computación');
INSERT INTO categorias (id, nombre) VALUES (6, 'Otro');

INSERT INTO productos (id, categoria_id, nombre, precio, cantidad) VALUES(1, 1, 'Panasonic Pantalla LCD', 259990, 20);
INSERT INTO productos (id, categoria_id, nombre, precio, cantidad) VALUES(2, 1, 'Sony Cámara digital DSC-W320B', 123490,
12);
INSERT INTO productos (id, categoria_id, nombre, precio, cantidad) VALUES(3, 1, 'Apple iPod shuffle', 1499990, 25);
INSERT INTO productos (id, categoria_id, nombre, precio, cantidad) VALUES(4, 5, 'Sony Notebook Z110', 37990, 10);
INSERT INTO productos (id, categoria_id, nombre, precio, cantidad) VALUES(5, 5, 'Hewlett Packard Multifuncional F2280',
69990, 7);
INSERT INTO productos (id, categoria_id, nombre, precio, cantidad) VALUES(6, 4, 'Bianchi Bicicleta Aro 26', 69990, 5);
INSERT INTO productos (id, categoria_id, nombre, precio, cantidad) VALUES(7, 6, 'Mica Cómoda 5 Cajones', 299990, 20);
```

7. Abrir y estudiar la clase **Categoria.java**.
    - Expandir **basedatos_springmvc_hibernate_categorias/src/main/java**.
    - Expandir **com.bolsadeideas.ejemplos.models.entity**
    - Doble clic en **Categoria.java**.

```
package com.bolsadeideas.ejemplos.models.entity;

import java.io.Serializable;
import javax.persistence.*;

@Entity
@Table(name = "categorias")
public class Categoria implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String nombre;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    @Override
    public String toString() {
        return nombre;
    }
}
```

- Nuestra clase **Categoria** para nuestros productos, cada Producto va a tener asociada una Categoria, luego veremos en la clase Producto cómo se relacionan mediante la anotación **@ManyToOne**.
- Implementamos el **método toString()** con el nombre o el id (campos que coincidan con el **itemLabel** o **itemValue** del elemento select del formulario).
- Está mapeada a la tabla categorías con la anotación: **@Table(name = "categorias")**

8. Abrir y estudiar la clase **Producto.java**.
    - Expandir **basedatos_springmvc_hibernate_categorias/src/main/java**.
    - Expandir **com.bolsadeideas.ejemplos.models.entity**
    - Doble clic en **Producto.java**.

```
package com.bolsadeideas.ejemplos.models.entity;
import java.io.Serializable;
import javax.persistence.*;

@Entity
@Table(name = "productos")
public class Producto implements Serializable {

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

    @NotEmpty private String nombre;

    @NotNull private Integer precio;

    @NotNull private Integer cantidad;

    @NotNull
    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "categoria_id")
    private Categoria categoria;

    public Categoria getCategoria() {
        return categoria;
    }

    public void setCategoria(Categoria categoria) {
        this.categoria = categoria;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
    ...ETC... LOS OTROS MÉTODOS SETTER
}
```

- Observamos la relación con Categoría usando la anotación **@ManyToOne** y métodos sette y getter

9. Abrir y estudiar la intefaz DAO **ICategoriaDao.java**.
    - Expandir **basedatos_springmvc_hibernate_categorias/src/main/java**.
    - Expandir **com.bolsadeideas.ejemplos.models.dao**
    - Doble clic en **ICategoriaDao.java**.

```
package com.bolsadeideas.ejemplos.models.dao;

import java.util.List;
import com.bolsadeideas.ejemplos.models.entity.Categoria;

public interface ICategoriaDao {
    public List<Categoria> findAll();
    public Categoria findById(int id);
}
```

- La interfaz Dao con las operaciones y contrato que tendrá la clase Dao que accede a nuestros datos, agregamos dos contratos para las categorías.

10. Abrir y estudiar la clase Dao **CategoriaDao.java**.
    - Expandir **basedatos_springmvc_hibernate_categorias/src/main/java**.
    - Expandir **com.bolsadeideas.ejemplos.models.dao**
    - Doble clic en **CategoriaDao.java**.

```
package com.bolsadeideas.ejemplos.models.dao;

import java.util.List;

import com.bolsadeideas.ejemplos.models.entity.Categoria;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.orm.hibernate4.HibernateTemplate;
import org.springframework.stereotype.Repository;

@Repository("categoriaDao")
public class CategoriaDao implements ICategoriaDao {

    @Autowired
    private HibernateTemplate hibernateTemplate;

    @SuppressWarnings("unchecked")
    @Override
    public List<Categoria> findAll() {
        return (List<Categoria>) hibernateTemplate.find("from Categoria");
    }

    @Override
    public Categoria findById(int id) {
        return hibernateTemplate.get(Categoria.class, id);
    }
}
```

11. Abrir y estudiar la clase Service **ProductoServiceImpl.java**.
    - Expandir **basedatos_springmvc_hibernate_categorias/src/main/java**.
    - Expandir **com.bolsadeideas.ejemplos.models.services**
    - Doble clic en **ProductoServiceImpl.java**.

```
package com.bolsadeideas.ejemplos.models.services;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.bolsadeideas.ejemplos.models.dao.ICategoriaDao;
import com.bolsadeideas.ejemplos.models.dao.IProductoDao;
import com.bolsadeideas.ejemplos.models.entity.Categoria;
import com.bolsadeideas.ejemplos.models.entity.Producto;

@Service("productoService")
public class ProductoServiceImpl implements IProductoService {

    @Autowired
    private IProductoDao productoDao;

    @Autowired
     private ICategoriaDao categoriaDao;

    @Override
    @Transactional(readOnly = true)
    public List<Producto> findAll() {
        return productoDao.findAll();
    }

    @Override
    @Transactional(readOnly = true)
    public Producto findById(int productoId) {
        return productoDao.findById(productoId);
    }

    @Override
    @Transactional
    public void save(Producto producto) {
        productoDao.save(producto);
    }

    @Override
    @Transactional
    public void delete(Producto producto) {
        productoDao.delete(producto);
    }
```

```
    @Override
    @Transactional(readOnly = true)
    public List<Categoria> findAllCategorias() {
        return categoriaDao.findAll();
    }

    @Override
    @Transactional(readOnly = true)
    public Categoria findCategoriaById(int id) {
        return categoriaDao.findById(id);
    }
}
```

12. Abrir y estudiar la intefaz Service **IProductoService.java**.
    - Expandir **basedatos_springmvc_hibernate_categorias/src/main/java**.
    - Expandir **com.bolsadeideas.ejemplos.models.services**
    - Doble clic en **IProductoService.java**.

```
package com.bolsadeideas.ejemplos.models.dao;

import java.util.List;
import com.bolsadeideas.ejemplos.models.entity.Categoria;
import com.bolsadeideas.ejemplos.models.entity.Producto;

public interface IProductoService {
    public List<Producto> findAll();
    public Producto findById(int productoId);
    public void save(Producto producto);
    public void delete(Producto producto);
     public List<Categoria> findAllCategorias();
     public Categoria findCategoriaById(int id);
}
```

13. Abrir y estudiar la clase controladora **ProductoController.java**.
    - Expandir **basedatos_springmvc_hibernate_categorias/src/main/java**.
    - Expandir **com.bolsadeideas.ejemplos.controllers**
    - Doble clic en **ProductoController.java**.

```
package com.bolsadeideas.ejemplos.controllers;

import ...ETC...

@Controller
@RequestMapping("/catalogo")
@SessionAttributes("producto")
public class ProductoController {

    @Autowired
    private IProductoService productoService;

    @InitBinder
    protected void initBinder(ServletRequestDataBinder binder) {
        binder.registerCustomEditor(Categoria.class,
                new CategoriaPropertyEditor(productoDao));
    }

    @RequestMapping(value = "/listado.htm", method = RequestMethod.GET)
    @ModelAttribute("productos")
    public List<Producto> listado(Model modelo) throws Exception {
        modelo.addAttribute("titulo", "Listado de Productos");
        return  productoService.findAll();
    }

    @RequestMapping(value = "/form.htm", method = RequestMethod.GET)
    public String setupForm(@RequestParam(value = "id", required = false,
                                    defaultValue = "0") int id, Model model) {
        Producto producto = null;
        if (id > 0) {
            producto =  productoService.findById(id);
        } else {
            producto = new Producto();
        }

        model.addAttribute("categorias",  productoService.findAllCategorias());
        model.addAttribute("producto", producto);
        return " catalogo/form";
    }
```

```
        @RequestMapping(value = "/form.htm", method = RequestMethod.POST)
        public String processSubmit(Model model, @ModelAttribute("producto")
                    Producto producto, BindingResult result, SessionStatus status) {

            new ProductoValidator().validate(producto, result);
            if (result.hasErrors()) {
                model.addAttribute("categorias",  productoService.findAllCategorias());
                return " catalogo/form";
            } else {
                 productoService.save(producto);
                status.setComplete();
                return "redirect:listado.htm";
            }
        }

        @RequestMapping(value = "/eliminar.htm", method = RequestMethod.GET)
        public String eliminar(@RequestParam("id") int id) {
            Producto producto = productoDao.findById(id);

            if (null != producto) {
                 productoService.delete(producto);
            }

            return "redirect:listado.htm";
        }

        @ModelAttribute("titulo")
        public String populateTitulo() {
            return "Formulario Producto";
        }
    }
```

- En el método **setupForm** pasamos las **categorías** a la vista para poblarlas en la lista desplegable **form:select** en el formulario.
- Además la clase ProveedorPropertyEditor se tiene que registrar en el controlador en el método initBinder anotado con **@InitBinder**
- La clase **ProveedorPropertyEditor** se **encarga de trasformar** el valor del string seleccionado de la lista desplegable **form:select** en un objeto categoría para que pueda ser persistido en la base de datos.

14. Abrir y estudiar la clase Property Editor **CategoriaPropertyEditor.java**.
    - Expandir **basedatos_springmvc_hibernate_categorias/src/main/java**.
    - Expandir **com.bolsadeideas.ejemplos.controllers**
    - Doble clic en **CategoriaPropertyEditor.java**.

```
package com.bolsadeideas.ejemplos.controllers;

import java.beans.PropertyEditorSupport;
import com.formacionbdi.spring.webmvc.catalogo.models.dao.IProductoDao;

public class CategoriaPropertyEditor extends PropertyEditorSupport {

    private IProductoService productoService;

    public CategoriaPropertyEditor(IProductoService productoService) {
        this.productoService = productoService;
    }

    @Override
    public void setAsText(String idStr) throws IllegalArgumentException {
        setValue(productoService.findCategoriaById(Integer.parseInt(idStr)));
    }
}
```

- Un **Property Editor** se utiliza para convertir una cadena del tipo String en un objeto que pueda ser persistido en la base de datos.
- Debe heredar de la clase **java.beans.PropertyEditorSupport** y sobrescribir el método **setAsText()**, necesario para obtener la categoría como instancia u objeto a partir del **String idStr** (id de la categoría) recibido como argumento en el método **setAsText(String idStr)**.
- Además se debe pasar mediante el constructor una referencia del productoService, para obtener la **categoría** por su **id** y de esta forma asignarla usando el método **setValue**.

15. Abrir y estudiar la vista JSP **listado.jsp**.
    - Expandir **basedatos_springmvc_hibernate_categorias/src/main/webapp/WEB-INF/views/catalogo**.
    - Doble clic en **listado.jsp**.

```
ETC...
<table style="width: 700px;"
    class="table table-striped table-hover table-bordered">
    <thead>
        <tr>
            <th>#</th>
            <th>Nombre</th>
            <th>Precio</th>
            <th>Cantidad</th>
             <th>Categoría</th>
            <th>Editar</th>
            <th>Eliminar</th>
        </tr>
    </thead>
    <tbody>
        <c:forEach items="${productos}" var="producto">
            <tr>
                <td><c:out value="${producto.id}" /></td>
                <td><c:out value="${producto.nombre}" /></td>
                <td><c:out value="${producto.precio}" /></td>
                <td><c:out value="${producto.cantidad}" /></td>
                <td><c:out value="${producto.categoria.nombre}"/></td>
                <td><a class="btn-xs btn-primary"
                    href="<c:url
value="/catalogo/form.htm?id=${producto.id}"/>">editar</a></td>
                <td><a class="btn-xs btn-danger" onclick="return confirm('Esta seguro?');"
                    href="<c:url
value="/catalogo/eliminar.htm?id=${producto.id}"/>">eliminar</a></td>
            </tr>
        </c:forEach>
    </tbody>
</table>
ETC...
```

- Observamos que ahora se muestran las categorías en el listado de productos, las cuales se accede mediante el objeto producto por la relación **@ManyToOne**.

16. Abrir y estudiar la vista JSP **form.jsp**.
    - Expandir **basedatos_springmvc_hibernate_categorias/src/main/webapp/WEB-INF/views/catalogo**.
    - Doble clic en **form.jsp**.

```
/* etc ... /*

<form:form modelAttribute="producto" method="post"
    cssClass="form-horizontal" role="form">

    <div class="form-group">
        <form:label for="categoria" path="categoria"
            cssClass="col-sm-2 control-label">Categorías</form:label>
        <div class="col-sm-10">
            <form:select path="categoria" style="width: 300px;" cssClass="form-control"
                cssErrorClass="form-control alert-danger">
                <form:option value="0" label="seleccione una categoría -->" />
                <form:options items="${categorias}" itemLabel="nombre"
                    itemValue="id" />
            </form:select>
            <form:errors path="categoria" />
        </div>
    </div>

    <div class="form-group">
        <form:label for="nombre" path="nombre"
            cssClass="col-sm-2 control-label">Nombre</form:label>
        <div class="col-sm-10">
            <form:input path="nombre" style="width: 300px;"
                cssClass="form-control"
                cssErrorClass="form-control alert-danger" />
            <form:errors path="nombre" />
        </div>
       </div>
   /* etc ... /*
```

- En el elemento del formulario **<form:select path="categoria">** mostramos las categorías en una lista desplegable, asignando **itemLabel** con el "**nombre**" y **itemValue** con el "**id**".
- Las categorias fueron pasadas a la vista desde el controlador en el método **setupForm()** mediante el objeto Model: **model.addAttribute("categorias", productoService.findAllCategorias())**.

17. Como **tarea** y ejercicio estudiar todos los demás proyectos, sus clases, vistas y archivos de configuraciones:
    - **basedatos_springmvc_hibernate_HibernateDaoSupport** (En vez de usar directamente HibernateTemplate, se usa y hereda de HibernateDaoSupport)
    - **basedatos_springmvc_hibernate_paginador** (Ejemplo para paginar resultados con Hibernate)
    - **basedatos_springmvc_jpa2** (Ejemplo JPA2)
18. Cualquier duda lo revisamos en el foro.

Resumen

En este workshop, hemos visto las funciones avanzadas de Spring Framework Web MVC usando anotaciones, utilizando dos completos ejemplos, de los cuales uno es un avanzado ejercicio complementado con Hibernate para crear un CRUD.

Además hemos implementado un sistema de compras online utilizando Spring Framework, con carrito de compras y manejo de sesiones HTTP manejadas con Spring.

Envía tus consultas a los foros!

Aquí es cuando debes sacarte todas las dudas haciendo consultas en los foros correspondientes

AUTOR: Andrés Guzmán Fontecilla.
Email: andresguzf@gmail.com
WEB: http://www.bolsadeideas.com