Introducción a MVC en Spring

Índice

En este tema se hará una introducción a las características del *framework* modelo-vista-controlador que incorpora Spring. Veremos que tiene una completa y bien pensada arquitectura, altamente configurable, que a primera vista lo hace parecer bastante complejo, siendo aún así fácil de usar en los casos más simples.

1. Spring MVC vs. Struts

Spring MVC tiene algunos puntos en común con Struts, y también muchas diferencias. Vamos a comentar unos y otras brevemente.

En cuanto a las semejanzas:

- Tanto Spring como Struts son representantes del tipo "push" de MVC, en que primero se realiza el trabajo y se obtienen los resultados y la vista se limita a mostrarlos. Por ello el flujo de procesamiento de la petición resultará familiar hasta cierto punto a los que ya hayan trabajado con Struts. En JSF, como recordaréis, la vista es la que dispara la lógica de negocio.
- Ambos ofrecen mecanismos conceptualmente similares para encapsular los parámetros de la petición HTTP (recordemos los ActionForm de Struts) y validar los datos antes de disparar la lógica de negocio. Spring también tiene validación programada y declarativa.

No obstante, también hay muchas diferencias. La fundamental, que permea todo el *framework*, es que Spring tiene una arquitectura mejor estructurada y que resuelve mejor ciertos problemas, lo cual no es sorprendente si tenemos en cuenta que Spring es mucho más moderno que Struts y que ha podido aprovechar la experiencia ganada en el uso durante años de Struts y otros *frameworks* MVC. Vamos a ver brevemente algunas diferencias, que quedarán más claras cuando expliquemos con más detalle el funcionamiento:

- Aunque el flujo de procesamiento de la petición HTTP es similar al de Struts, es más complejo, ofreciendo muchos puntos en el mismo para que el desarrollador coloque sus propias clases que hagan tareas particulares.
- El papel de las acciones de Struts aquí lo desempeñan los denominados `Controllers`
- Aunque en Struts todas las acciones son en principio "iguales" y pueden hacer cualquier tarea, en Spring distintos tipos de `Controllers` están pensados para hacer distintas tareas. Por ejemplo, para procesar los datos de un formulario no heredaríamos del mismo `Controller` que para simplemente mostrar todos los registros de una tabla (aquí no necesitamos formulario).
- En lugar de usar Javabeans que hereden de `ActionForm` para recoger los datos de la petición HTTP, aquí se usan JavaBeans comunes, es decir, no tienen que heredar de ninguna clase especial.

2. Procesamiento de una petición en Spring MVC

A continuación se describe el flujo de procesamiento típico para una petición HTTP en Spring MVC. Este diagrama está simplificado y no tiene en cuenta ciertos elementos que luego comentaremos.

- Como prácticamente en todos los *frameworks* MVC, en Spring se canalizan todas las peticiones HTTP a través de un solo servlet, en este caso uno de la clase `DispatcherServlet` implementada por Spring.
- El servlet se ayuda de un `HandlerMapping` para averiguar, normalmente a partir de la URL, a qué `Controller` hay que llamar para servir la petición.
- Se llama al `Controller`, que ejecuta la lógica de negocio, obtiene los resultados y los devuelve al servlet, junto con el nombre lógico de la vista a mostrar, encapsulados en un objeto de la clase `ModelAndView` .
- Un `ViewResolver` se encarga de averiguar el nombre físico de la vista que se corresponde con el nombre lógico del paso anterior.
- Finalmente, el `DispatcherServlet` redirige la petición hacia la vista, que muestra los resultados de la operación realizada.

En realidad, el procesamiento es más complejo. Nos hemos saltado algunos pasos en aras de una mayor claridad. Por ejemplo, en Spring se pueden usar interceptores, que son como los filtros del API de servlets, pero adaptados a Spring MVC. Estos interceptores pueden pre y postprocesar la petición alrededor de la ejecución del `Controller`. No obstante, todas estas cuestiones deben quedar por fuerza fuera de una breve introduccion a Spring MVC como la de estas páginas.

3. Configuración básica

La implementación de las clases necesarias para el módulo MVC está incluída en el `spring.jar`,de modo que si estamos usando otros módulos de Spring en nuestro proyecto ya es probable que lo tengamos incluído. Además necesitaremos configurar el `web.xml` para que todas las peticiones HTTP con un determinado patrón se canalicen a través del `DispatcherServlet` de Spring. Como mínimo necesitaremos incluir algo como esto:

```
<servlet>
   <servlet-name>spring</servlet-name>
   <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
   </servlet-class>
   <load-on-startup>1</load-on-startup>
```

```
</servlet>
<servlet-mapping>
   <servlet-name>spring</servlet-name>
   <url-pattern>*.mvc</url-pattern>
</servlet-mapping>
```

Con esta configuración, todas las peticiones acabadas en `.mvc` se redirigirían al servlet principal, por ejemplo `getPedido.mvc` o `verClientes.mvc`.

De modo similar a como se hacía en Struts con el `struts-config.xml`, en Spring se usa un fichero de configuración XML llamado por defecto `spring-servlet.xml`, que se supone colocado en `WEB-INF`. Podemos cambiar la localización y/o el nombre de este fichero pasándole un parámetro llamado `contextConfigLocation` al `DispatcherServlet`. Por ejemplo:

```
<servlet>
   <servlet-name>spring</servlet-name>
   <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
   </servlet-class>
   <load-on-startup>1</load-on-startup>
   <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/classes/mvc.xml</param-value>
   </init-param>
</servlet>
<servlet-mapping>
   <servlet-name>spring</servlet-name>
   <url-pattern>*.mvc</url-pattern>
</servlet-mapping>
```

Con esta configuración, el fichero pasaría a llamarse `mvc.xml` y a ser buscado en `/WEB-INF/classes`.

4. Caso 1: petición sin procesamiento de datos de entrada

La elaborada arquitectura de Spring MVC, y las muchas posibilidades que tiene el usuario de configurar a su medida el procesamiento que hace el *framework* hacen que sea poco intuitivo hacer una descripción abstracta de Spring MVC, al menos si no se dispone del suficiente tiempo para hacerlo de manera pausada, lo que no es el caso. En su lugar, hemos preferido aquí describir cómo se implementarían un par de casos típicos en una aplicación web, indicando cómo implementar cada caso y las posibilidades adicionales que ofrece Spring MVC. El lector tendrá que consultar fuentes adicionales para ver con detalle el resto de opciones.

El primer caso sería el de una petición que no necesita interacción por parte del usuario en el sentido de proceso de datos de entrada: por ejemplo sacar un listado de clientes, mostrar los datos de un pedido, etc. La "no interacción" aquí se entiende como que no hay que procesar y validar datos de entrada. Es decir, que no hay un formulario HTML. Esto no quiere decir que no haya parámetros HTTP, pero entonces suelen estar fijos en la URL de un enlace o de modo similar, no introducidos directamente por el usuario. Estas peticiones suelen ser simplemente listados de información de "solo lectura".

Vamos a poner estos ejemplos en el contexto de una hipotética aplicación web para un hotel, en la cual se pueden ver y buscar ofertas de habitaciones, disponibles con un determinado precio hasta una fecha límite. Aquí tendríamos lo que define a una oferta:

```
 package es.ua.jtech.spring.dominio;

import java.math.BigDecimal;
import java.util.Date;

public class Oferta {
        private BigDecimal precio;
        private Date fechaLimite;
        private TipoHabitacion tipoHab;
        private int minNoches;

        //..aquí vendrían los getters y setters
}
```

TipoHabitación es un tipo enumerado que puede ser `individual` o `doble`.

4.1. El Controller

Esto sería el equivalente a la acción de Struts. Si en Struts nuestra acción debe heredar de la clase `Action` aquí ocurre algo parecido con la familia de `Controllers`, con la diferencia de que no hay una única clase, sino varias, de las que debemos escoger la más apropiada. En el caso que nos ocupa (no hay formulario HTML) la clase más indicada es `AbstractController`. Supongamos que queremos sacar un listado de ofertas del mes. Nuestro `Controller` podría comenzar así:

```
package es.ua.jtech.spring.ejemplo.mvc;

import org.springframework.web.servlet.mvc.AbstractController;

public class ListaOfertasController extends AbstractController {


}
```

Cualquier `Controller` necesitará para hacer su trabajo de la colaboración de uno o más objetos de negocio. Si estamos usando Spring, lo lógico es que estos objetos sean beans de Spring y que instanciemos las dependencias haciendo uso del contenedor IoC. En nuestro caso supongamos que nos hace falta un objeto que implemente el interfaz `GestorOfertas`. Dicho objeto es el que "sabe" sacar de la base de datos las ofertas del mes con el método `public List<Oferta> getOfertasActuales()`. Probablemente este objeto deba ayudarse de un DAO o de otros objetos de negocio para hacer su trabajo, pero esto no nos interesa aquí.

Necesitamos por tanto en el código una referencia al "gestor de ofertas", y un *setter* para que Spring pueda "inyectar" dicho objeto en el controlador(en negrita el código añadido):

```
package es.ua.jtech.spring.mvc;

import es.ua.jtech.spring.negocio.GestorOfertas;
import org.springframework.web.servlet.mvc.AbstractController;

public class ListaOfertasController extends AbstractController {

   private GestorOfertas miGestor;

   public void setMiGestor(GestorOfertas miGestor) {
      this.miGestor = miGestor;
   }

}
```

Para que Spring resuelva automáticamente nuestra dependencia del `GestorOfertas` y lo instancie adecuadamente, es necesario que nuestro *controller* también sea un bean de Spring. Por tanto debemos definirlo en algún archivo de configuración de beans, que en el caso de la capa MVC es el `spring-servlet.xml` (aunque como ya hemos visto se le puede cambiar el nombre por defecto)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       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-2.0.xsd">

  <bean name="/verOfertas.mvc"
class="es.ua.jtech.spring.mvc.ListaOfertasController">
          <property name="miGestor" ref="miGestorOfertas"/>
  </bean>
  <bean id="miGestorOfertas"
class="es.ua.jtech.spring.negocio.GestorOfertasDummy">
  </bean>
```

```
</beans>
```

La razón de que el bean *controller* se haya definido con name="/verOfertas.mvc" se verá en el siguiente apartado, y como puede suponerse está relacionada con la URL mediante la que se accederá al bean.

Como puede verse, hemos enlazado la propiedad `miGestor` con un bean de Spring que configuramos también en el mismo XML.

El procesamiento de la petición se hace en el método `handleRequestInternal`, que se sobreescribe de la clase base `AbstractController`. Dicho método tiene dos parámetros: la petición y la respuesta HTTP.

`handleRequestInternal` debe devolver un objeto de la clase `ModelAndView`, en el que se encapsule el nombre lógico de la vista y el modelo con los resultados de la operación realizada, en nuestro caso la lista de ofertas. El modelo es simplemente un `Map` en el que podemos ir añadiendo objetos para luego acceder a ellos por el mismo nombre en la vista. Aquí tenemos una posible implementación para dicho método:

```
//... resto de la clase ListaOfertasController
protected ModelAndView handleRequestInternal(HttpServletRequest arg0,
                                             HttpServletResponse arg1)
throws Exception {
   //Creamos un nuevo ModelAndView que por ahora solo tiene el nombre
lógico de la vista
   ModelAndView mav = new ModelAndView("ofertas");

   //Llamamos al objeto de negocio y obtenemos el resultado
   List<Oferta> ofertas = miGestor.getOfertasActuales();

   //Guardamos el resultado en el ModelAndView, con el nombre "ofertas"
   mav.addObject("ofertas",ofertas);

   //Devolvemos el ModelAndView
   return mav;
}
//... resto de la clase ListaOfertasController
```

Ya tenemos por fin el código completo del Controller

```
package es.ua.jtech.spring.mvc;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;
```

```
import java.util.List;

import es.ua.jtech.spring.negocio.GestorOfertas;
import es.ua.jtech.spring.dominio.Oferta;

public class ListaOfertasController extends AbstractController {

        private GestorOfertas miGestor;

        public void setMiGestor(GestorOfertas miGestor) {
                this.miGestor = miGestor;
        }

        @Override
        protected ModelAndView handleRequestInternal(HttpServletRequest
arg0,
                        HttpServletResponse arg1) throws Exception {
                ModelAndView mav = new ModelAndView("ofertas");
                List<Oferta> ofertas = miGestor.getOfertasActuales();
                mav.addObject("ofertas",ofertas);
                return mav;

        }
}
```

4.2. El mapeo entre la URL y el bean controller

El encargado de asociar la petición con un determinado *controller* es el `HandlerMapping`. Spring ofrece varias implementaciones distintas de este interfaz, y siempre podemos escribir la nuestra propia. No obstante, por defecto usa una clase llamada `BeanNameUrlHandlerMapping`, que, de modo parecido a como se haría en un `struts-config.xml` de Struts, asocia una URL comenzando por "/" con una clase de un *controller*. Nótese que en el caso de usar patrones del estilo de *.mvc o *.do para los *controller* aquí no podemos "ahorrarnos" el ".mvc", ".do" o lo que sea, al contrario de lo que ocurría en el `struts-config.xml`.

4.3. Resolver el nombre lógico de la vista

La última tarea que queda es resolver el nombre lógico de la vista, asociándolo a una vista física. Para ello necesitamos un `ViewResolver`. Al contrario que en el caso del `HandlerMapping`, Spring no nos proporciona ninguno por defecto, así que debemos definir un bean con el `id=viewResolver` y la clase que nos interese. De las que proporciona Spring una de las más sencillas de usar es `InternalResourceViewResolver`. Esta clase usa dos parámetros básicos: `preffix` y `suffix`, que puestos respectivamente delante y detrás del nombre lógico de la vista nos

dan el nombre físico. Por ejemplo:

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
         <property name="preffix" value="/jsp/"/>
         <property name="suffix" value=".jsp"/>
</bean>
```

Así, si el nombre lógico de la vista de nuestro ejemplo era `ofertas`, se acabaría buscando el recurso físico `/jsp/ofertas.jsp`.

En este caso, la vista no tiene nada particular de Spring. Solo necesitamos código java o mejor etiquetas JSTL que muestren los resultados de la operación. Recuérdese que en el `controller` habíamos metido las ofertas en el `ModelAndView` con el nombre "ofertas", por el que ahora serán accesibles como un bean normal de JSP:

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core"prefix="c"%>
<html>
<head>
        <title>Ejemplo de vista</title>
</head>
<body>
<h1>Superofertas del mes</h1>
<c:forEach items="${ofertas}" var="o">
        Habitación ${o.tipoHab} un mínimo de ${o.minNoches} noches por solo
${o.precio} &euro; la noche<br/>
</c:forEach>
</body>
</html>
```

5. Caso 2: procesamiento de un formulario

Este caso es más complejo ya que implica varios pasos:

- El usuario introduce los datos, normalmente a través de un formulario HTML
- Los datos se validan, y en caso de no ser correctos se vuelve a mostrar el formulario para que el usuario pueda corregirlos.
- En caso de pasar la validación, los datos se "empaquetan" en un objeto Java para que el `controller` pueda acceder a ellos de modo más sencillo que a través de la petición HTTP.
- El `controller` se ejecuta, toma los datos, realiza la tarea y cede el control para que se muestre la vista.

Esto en Struts lo haríamos normalmente con dos acciones, una de ellas para mostrar inicialmente el formulario y otra para procesar los datos introducidos. En Spring hay una familia de `controllers` pensados para tanto mostrar el formulario como procesar los datos, de los cuales probablemente el más sencillo de usar es el `SimpleFormController`. Por otro lado en Struts se usaría un *actionform* para empaquetar y validar los datos. En Spring se usa un objeto similar (aunque aquí se le llama `Command`), con la diferencia de que la clase que lo implementa no es necesario que herede de ninguna clase en especial, únicamente debe ser un JavaBean. Recordemos que en Struts un *actionform* debe ser un javabean y *además* heredar de `ActionForm` o de `DynaActionForm`.

Por ejemplo, este podría ser un `Command` apropiado para buscar ofertas.Solo contiene los campos estrictamente necesarios para la búsqueda, no todos los datos que puede contener una oferta:

```
package es.ua.jtech.spring.mvc;

import java.math.BigDecimal;

import es.ua.jtech.spring.dominio.TipoHabitacion;

public class BusquedaOfertas {
        private BigDecimal precioMax;
        private TipoHabitacion tipoHab;

        //..ahora vendrían los getters y setters
}
```

Desde el punto de vista de lo que tenemos que implementar, este caso solo se diferenciará del caso 1 (sin procesamiento de datos de entrada) en el `controller` y en que para la vista podemos usar *tags* de Spring, del mismo modo que en Struts usábamos las suyas propias, para que se conserve el valor de los campos y el usuario no tenga que volver a escribirlo todo si hay un error de validación. La asociación entre la URL y el controlador y entre la vista lógica y el recurso físico serán igual que antes. Además, por supuestoo, tendremos que implementar la validación de datos.

5.1. El controller

En el ejemplo, vamos a describir cómo usar el `SimpleFormController`, ya que nos parece el más sencillo de usar, aunque por supuesto en Spring hay varias implementaciones adicionales de `controllers` para trabajar con formularios, más sofisticadas.

En el constructor del `SimpleFormController` se suele dar valor a las propiedades que controlan su funcionamiento, en concreto

- Al nombre lógico del `Command`,el equivalente al *actionform* de Struts (propiedad `CommandName`)
- A la clase que implementa este `Command` (propiedad `CommandClass`)
- A la vista que muestra el formulario para introducir datos (propiedad `FormView`), y a la que se volverá si hay un error de validación.
- A la vista que muestra los resultados de la operación (propiedad `SuccessView`)

Estas propiedades se asignan simplemente con *setters*, por ejemplo:

```
package es.ua.jtech.spring.mvc;

import org.springframework.web.servlet.mvc.SimpleFormController;

public class BuscarOfertasController extends SimpleFormController {
        private GestorOfertas miGestorOfertas;

        public BuscarOfertasController() {
                setCommandName("busquedaOfertas");
                setCommandClass(BusquedaOfertas.class);
                setFormView("buscarOfertas");
                setSuccessView("resultBuscarOfertas");
        }
        //.. resto de la clase
}
```

El controller realiza su trabajo en el método `onSubmit` (recordemos que cuando usábamos el `AbstractController` el método equivalente era `handleRequestInternal`). Este método recibe como parámetro un `Command`, del que tomaremos los datos. Recordar que al igual que en Struts, **si hemos llegado a este punto es que ya se ha hecho la validación y ha tenido éxito**. No obstante aquí la validación la tratamos después por no complicar por el momento la discusión.

Por supuesto, nuestro controller necesitará de la colaboración de algún objeto de negocio para hacer su trabajo, y lo habitual es que el contenedor IoC nos pase la referencia a este objeto llamando a un *setter*. Ya podemos escribir el *controller* completo (en negrita el código nuevo):

```
 package es.ua.jtech.spring.mvc;

import java.util.List;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.SimpleFormController;
import es.ua.jtech.spring.negocio.GestorOfertas;
import es.ua.jtech.spring.dominio.Oferta;

public class BuscarOfertasController extends SimpleFormController {
        private GestorOfertas miGestorOfertas;
```

```
        public BuscarOfertasController() {
                setCommandName("busquedaOfertas");
                setCommandClass(BusquedaOfertas.class);
                setFormView("buscarOfertas");
                setSuccessView("resultBuscarOfertas");
        }

        public void setMiGestorOfertas(GestorOfertas miGestorOfertas) {
                this.miGestorOfertas = miGestorOfertas;
        }

        @Override
        protected ModelAndView onSubmit(Object command) throws Exception {
                BusquedaOfertas bo = (BusquedaOfertas) command;
                List<Oferta> encontradas =
miGestorOfertas.buscarOfertas(bo.getPrecioMax(), bo.getTipoHab());
                ModelAndView mav = new ModelAndView(getSuccessView());
                mav.addObject("ofertas", encontradas);
                return mav;
        }

}
```

5.2. La validación de datos

En Spring, al igual que en Struts, se puede realizar la validación de datos por programa al igual que de manera declarativa. De hecho, la versión declarativa se hace usando el commons validator de Jakarta, el mismo componente que se usa en Struts. No obstante, no trataremos aquí la validación declarativa por cuestiones de espacio, sino que nos limitaremos a dar un ejemplo de la programada.

Para validar un `Command` de manera programada, necesitamos una clase que implemente el interfaz `org.springframework.validation.Validator` . Supongamos que queremos rechazar la oferta buscada si el precio está vacío o bien no es un número positivo (para simplificar vamos a obviar la validación del tipo de habitación). El código sería:

```
 package es.ua.jtech.spring.mvc;

import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;


public class OfertaValidator implements Validator {

        public boolean supports(Class arg0) {
                return arg0.isAssignableFrom(BusquedaOfertas.class);
```

```
        }

        public void validate(Object obj, Errors errors) {
                ValidationUtils.rejectIfEmpty(errors, "precioMax",
"precioVacio");
                BusquedaOfertas bo = (BusquedaOfertas) obj;
                //comprobar que el precio no esté vacío (para que no haya
null pointer más abajo)
                if (bo.getPrecioMax()==null)
                        return;
                //comprobar que el número sea positivo
                if (bo.getPrecioMax().floatValue()<0)
                        errors.rejectValue("precioMax", "precNoVal");
        }

}
```

Como vemos, un `Validator` debe implementar al menos dos métodos:

- supports: indica de qué clase debe ser el `Command` creado para que se considere aceptable. En nuestro caso debe ser de la clase `BusquedaOfertas`
- validate: es donde se efectúa la validación. El primer parámetro es el `Command`, que se pasa como un `Object` genérico (lógico, ya que Spring no nos obliga a implementar ningún interfaz ni heredar de ninguna clase determinada). El segundo es una especie de lista de errores. Como vemos, hay métodos para rechazar un campo si es vacío o bien por código podemos generar errores a medida (en este caso, si el precio es un número negativo).

  Mención aparte merecen los mensajes de error. Al igual que en Struts, son claves en un fichero `.properties`, asociadas con nombres de propiedades del Command. Como se ve, en nuestro caso la propiedad a la que se asocian los errores es precioMax, como es lógico. En el archivo de configuración de beans debemos definir el nombre del archivo de mensajes, simplemente necesitamos un bean cuyo `id` sea `messageSource`

  ```
  <bean id="messageSource"
  class="org.springframework.context.support.ResourceBundleMessageSource">
          <property name="basename"><value>mensajes</value></property>
  </bean>
  ```

  Y aquí tenemos el archivo `mensajes.properties`:

  ```
  precioVacio = el precio está vacío
  precNoVal = precio no válido
  typeMismatch.precioMax = el precio no es un número
  ```

  La clave `typeMismatch.precioMax` la referencia automáticamente Spring cuando

se introduce en el campo un valor no compatible, en este caso uno que no sea un número. Si no definimos un `typeMismatch.XXXX` Spring muestra un mensaje de error por defecto.

Todavía nos falta configurar el controller en el XML correspondiente. Fijaos en que tiene una propiedad con id="validator" que es el validator que hemos definido:

```
 <bean name="/buscarOfertas.mvc"
class="es.ua.jtech.spring.mvc.BuscarOfertasController">
        <property name="miGestorOfertas" ref="miGestorOfertas"/>
        <property name="validator">
                <bean class="es.ua.jtech.spring.mvc.OfertaValidator"/>
        </property>
</bean>
```

5.3. La vista con las taglibs de Spring

Finalmente,nos queda definir el formulario usando las taglibs de Spring para mostrar errores de validación y guardar los datos para que el usuario no tenga que teclearlos de nuevo. Por supuesto Spring tiene multitud de etiquetas para crear formularios HTML, de las que solo vamos a ver el mínimo necesario para que funcione nuestro ejemplo:

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head></head>
<body>
<h1>Búsqueda de ofertas</h1>
<form action="" method="post">
<spring:bind path="busquedaOfertas.precioMax">
  Precio máximo:
  <input type="text" name="precioMax" value="${status.value}"/>
  ${status.errorMessage} <br/>
</spring:bind>
        Tipo de habitación:
        <select name="tipoHab">
                <option>individual</option>
                <option>doble</option>
        </select> <br/>
        <input type="submit" value="Buscar"/>
</form>
</body>
</html>
```

La etiqueta `<spring:bind>` rodea a los campos que queremos asociar con alguna propiedad de algún bean de Spring. En nuestro caso es la propiedad `precioMax` del `command` `busquedaOfertas`, de ahí el `busquedaOfertas.precioMax`. Dentro de una etiqueta `<spring:bind>` la variable `status.value` contiene el valor de la propiedad y `status.errorMessage` el mensaje de error asociado, si lo hay.

Finalmente, obsérvese que el action del formulario está vacío, ya que es el mismo controller el que se ocupa tanto de mostrar la página como de procesarla.