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.

Aquí veremos la versión 2.5, que como principal novedad incluye la posibilidad de usar anotaciones para definir los beans necesarios, lo que elimina en gran medida el uso de XML.

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 un JAR distinto del núcleo de Spring, `spring-webmvc.jar`, que debemos incluir en nuestro proyecto. Como ocurría en Struts necesitaremos configurar el `web.xml` para que todas las peticiones HTTP con un determinado patrón se canalicen a través del mismo servlet, en este caso de la clase `DispatcherServlet` de Spring. Como mínimo necesitaremos incluir algo como esto:

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

```
<servlet-mapping>
   <servlet-name>dispatcher</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`.

Necesitaremos también un fichero de configuración XML para definir los beans de la capa web. Este fichero por defecto se supone colocado en `WEB-INF` y con el nombre *nombre_del_servlet*`-servlet.xml`. Como en nuestro ejemplo el servlet se llamaba "dispatcher", el nombre esperado sería `dispatcher-servlet.xml`. No obstante tanto la localización como el nombre del fichero se puede configurar pasándoselo al `DispatcherServlet` como parámetro bajo el nombre `contextConfigLocation`. Consultar la documentación de Spring o el formato del web.xml para más detalles.

Por tanto, la forma habitual de trabajar es usar un XML para los beans de la capa web y otro (u otros) distinto para los de la capa de negocio y DAOs. Spring establece una jerarquía de contextos de modo que en el XML de la capa web se heredan automáticamente los otros beans, lo que nos permite referenciar los objetos de negocio en nuestro código MVC.

Si vamos a usar anotaciones en la capa web y queremos que Spring examine automáticamente las clases en su búsqueda, debemos usar la etiqueta ya conocida **`component-scan` en el XML de la capa web, no basta con que esté colocada en el XML de los otros beans.**. Suponiendo que nuestros componentes web están implementados en el paquete `es.ua.jtech.spring.mvc`, en el `dispatcher-servlet.xml` aparecería:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">

  ...
       <context:component-scan
base-package="es.ua.jtech.spring.mvc"/>
  ...
</beans>
```

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 general 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`.

Lo primero es definir el controlador. Esto sería el equivalente a la acción de Struts. Con la aparición de la versión 2.5 de Spring aparece un "nuevo estilo" de controladores. Con los controladores "pre-2.5" pasa algo parecido a lo que ocurre con las acciones de Struts: heredan de una determinada clase y eso nos obliga a implementar ciertos métodos con una signatura determinada para procesar la petición. Eso sí, con la diferencia de que en Spring hay varias clases de las que podemos heredar, dependiendo del tipo de trabajo que deba hacer nuestro controlador. A partir de la generación 2.5 surge la posibilidad de usar cualquier clase con cualquier método para procesar las peticiones. Con anotaciones le indicamos a Spring qué métodos procesarán las peticiones y cómo enlazar sus parámetros con los parámetros HTTP.

Nota:

Estrictamente hablando, en los controladores "pre-2.5" en realidad no necesitamos heredar de

ninguna clase sino simplemente implementar un interfaz. No obstante, en la práctica se suele usar la herencia porque las clases base que implementa Spring aportan cierta funcionalidad útil.

4.1. Controlador totalmente basado en anotaciones

Nuestro controlador será simplemente un POJO con ciertas anotaciones. Supongamos que queremos sacar un listado de ofertas del mes. Así, el esqueleto básico de nuestro `Controller` sería:

```
import es.ua.jtech.spring.negocio.IGestorOfertas;

@Controller
@RequestMapping("/listaOfertas.do")
public class ListaOfertasController  {
   @Autowired
   private IGestorOfertas miGestor;

   ...
}
```

La anotación `@Controller` indica que la clase es un bean controlador y nos ahorra el trabajo de definir en XML el bean correspondiente. Con `@RequestMapping` asociamos una URL a este controlador.

Por supuesto, cualquier `Controller` necesitará para hacer su trabajo de la colaboración de uno o más objetos de negocio. Lo lógico es que estos objetos sean beans de Spring y que instanciemos las dependencias haciendo uso del contenedor. En nuestro caso dicho objeto es "miGestor". Supondremos que él es el que "sabe" sacar las ofertas del mes con el método `public List<Oferta> getOfertasDelMes(int mes)`. Probablemente a su vez este gestor deba ayudarse de un DAO para hacer su trabajo, pero esto no nos interesa aquí.

Por tanto, hemos usado `@Autowired` para resolver dependencias por tipo, suponiendo para simplificar que no hay ambigüedad (solo existe una clase que implemente `IGestorOfertas`)

Para que Spring sepa qué método del controlador debe procesar la petición HTTP se puede usar de nuevo la anotación `@RequestMapping`. Podemos especificar qué método HTTP (GET, POST,...) asociaremos al método java.

```
@Controller
@RequestMapping("/listaOfertas.do")
public class ListaOfertasController {
   @Autowired
   private GestorOfertas miGestor;


   @RequestMapping(method=RequestMethod.GET)
```

```
   public String procesar(HttpServletRequest req) {
      int mes = Integer.parseInt(req.getParameter("mes"));
      ...
   }
}
```

Aquí estamos asociando una llamada a "listaOfertas.do" de tipo GET con el método java `procesar`. Una llamada a la misma URL con POST produciría un error HTTP de tipo 404 ya que no habría nada asociado a dicha petición. Si no necesitamos tanto control sobre el método HTTP asociado podemos poner la anotación `@RequestMapping("/listaOfertas.do")` directamente en el método `procesar` en lugar de en la clase.

La primera tarea que debe hacer un método de procesamiento de una petición es **leer el valor de los parámetros HTTP**. Spring automáticamente detectará que `procesar` tiene un argumento de tipo `HttpServletRequest` y lo asociará a la petición actual. Esta "magia" funciona también con otros tipos de datos: por ejemplo, un parámetro de tipo `Writer` se asociará automáticamente a la respuesta HTTP y uno de tipo `Locale` al *locale* de la petición. Consultar la documentación de Spring para ver más detalladamente las asociaciones que se realizan automáticamente.

Otra posibilidad es asociar explícitamente parámetros HTTP con parámetros java mediante la anotación `@RequestParam`. Usando esta anotación, el ejemplo quedaría:

```
@Controller
@RequestMapping("/listaOfertas.do")
public class ListaOfertasController {
   @Autowired
   private GestorOfertas miGestor;


   @RequestMapping(method=RequestMethod.GET)
   public String procesar(@RequestParam("mes") int mes) {
      ...
   }
}
```

De este modo Spring hace por nosotros el trabajo de extraer el parámetro y convertir el tipo de String a int. Hay que tener en cuenta que esta forma de trabajar puede generar múltiples excepciones. Si el parámetro no está presente en la petición HTTP se producirá una excepción de tipo `MissingServletRequestParameterException`. Si está presente y no se puede convertir se generará una `TypeMismatchException`. Para especificar que el parámetro no es obligatorio, podemos hacer:

```
public String procesar(@RequestParam(value="mes",required=false)
int mes) {
    ...
}
```

En este caso, se recomienda usar un Integer en lugar de un int para que si no existe el

parámetro Spring nos pueda pasar el valor null (si no existe Spring no lo interpreta como 0).

Una vez leídos los parámetros, lo siguiente suele ser **disparar la lógica de negocio y colocar los resultados en algún ámbito al que pueda acceder la vista**. Spring ofrece algunas clases auxiliares para almacenar de manera conveniente los datos obtenidos. En realidad son simplemente `Map` gracias a los que se puede almacenar un conjunto de objetos dispares, cada uno con su nombre. Spring hace accesible el modelo a la vista como un atributo de la petición y al controlador si le pasamos un parámetro de tipo `ModelMap`. Aquí tenemos la lógica de `procesar`:

```
@RequestMapping(method=RequestMethod.GET)
public String procesar(@RequestParam("mes") int mes,
                       ModelMap modelo) {
   modelo.addAttribute("ofertas", miGestor.getOfertasDelMes(mes));
   return "listaOfertas";
}
```

Finalmente, nos queda **especificar qué vista hemos de mostrar**. Hemos estado viendo en todos los ejemplos que el método `procesar` devuelve un String. Dicho String se interpreta como el nombre "lógico" de la vista a mostrar tras ejecutar el controlador. Consultar el siguiente apartado para ver cómo asociar un nombre lógico de vista a un nombre físico (de página JSP, por ejemplo). En el JSP será sencillo mostrar los datos ya que, como hemos dicho, se guardan en un atributo de la petición:

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core"prefix="c"%>
...
<c:forEach items="${ofertas}" var="o">
        Habitación ${o.tipoHab} un mínimo de
        ${o.minNoches} noches por solo ${o.precio}eur./noche
</c:forEach>
...
```

4.2. Controlador "al estilo clásico"

En realidad, vamos a usar aquí un modelo "mixto" en el que empleamos anotaciones para definir el propio controlador y el mapeo con la URL pero el método de procesamiento de la petición no va a ser arbitrario. Estos controladores normalmente heredan de una cierta clase propia de Spring, y por tanto deben sobreescribir ciertos métodos con una determinada signatura. En el caso que nos ocupa (no hay procesamiento de datos de formulario) la clase más indicada es `AbstractController`.

```
...
@Controller
@RequestMapping("/listaOfertas.do")
public class ListaOfertasController extends AbstractController {
```

```
        @Autowired
        private GestorOfertas miGestor;

        //Falta el método para procesar la petición...
}
```

En un `AbstractController` la lógica de procesamiento del controlador se debe implementar en el método `handleRequestInternal`, que se sobreescribe de la clase base `AbstractController`. Dicho método tiene dos parámetros: la petición y la respuesta HTTP.

El método `handleRequestInternal` debe devolver un objeto de la clase `ModelAndView`, en el que se encapsula tanto el nombre lógico de la vista como 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:

```
...
protected ModelAndView handleRequestInternal(HttpServletRequest
arg0,
                                    HttpServletResponse arg1) throws
Exception {
   int mes = Integer.parseInt(argo.getParameter("mes"));
   //Creamos un nuevo ModelAndView que por ahora
   //solo tiene el nombre lógico de la vista
   ModelAndView mav = new ModelAndView("listaOfertas");
   //Llamamos al objeto de negocio y obtenemos el resultado
   List<Oferta> ofertas = miGestor.getOfertasDelMes(mes);
   //Guardamos el resultado en el ModelAndView, con el nombre
"ofertas"
   mav.addObject("ofertas",ofertas);
   //Devolvemos el ModelAndView
   return mav;
}
...
```

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`. Debemos definir un bean en el XML 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: `prefix` 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="prefix" 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`.

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 la práctica habitual es mostrar el formulario y procesar los datos dentro del mismo controlador. 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 es.ua.jtech.spring.dominio.TipoHabitacion;

public class BusquedaOfertas {
        private int 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 supuesto, tendremos que implementar la validación de datos.

Igual que antes, vamos a ver dos formas de hacerlo: con una clase propia con métodos arbitrarios, gracias al uso de anotaciones, y heredando de las clases base de Spring.

5.1. Controlador totalmente basado en anotaciones

Mediante la anotación `@RequestMapping` mapearemos la petición que muestra el formulario a un método java (será con GET) y la que lo procesa a otro distinto (POST):

```
...
@Controller
@RequestMapping("/busquedaOfertas.do")
public class BusquedaOfertasController {
    @Autowired
    private GestorOfertas miGestor;

    @RequestMapping(method=RequestMethod.GET)
    public String preparaForm(ModelMap modelo) {
       ...
    }

    @RequestMapping(method=RequestMethod.POST)
    public String procesaForm(@ModelAttribute("bo") BusquedaOfertas
bo,
                              BindingResult result, ModelMap
modelo) {
       ...
    }
}
```

Como ya hemos dicho, los datos del formulario se van a almacenar en un objeto de tipo `BusquedaOferta`. El método `preparaForm`, que se ejecutará *antes* de mostrar el formulario, debe crear un `BusquedaOferta` con los valores por defecto y colocarlo en el modelo para que puedan salir en el formulario. Finalmente, devolverá el nombre lógico de la vista que contiene el formulario. Aquí tenemos el código:

```
@RequestMapping(method=RequestMethod.GET)
public String preparaForm(ModelMap modelo) {
    modelo.addAttribute("bo", new BusquedaOfertas());
    return "busquedaOfertas";
}
```

En la página `busquedaOfertas.jsp` colocaremos un formulario usando las *taglibs* de Spring. Estas etiquetas nos permiten vincular el modelo al formulario de forma sencilla y además mostrar los errores de validación como veremos posteriormente.

```
<%@ taglib prefix="form"
uri="http://www.springframework.org/tags/form" %>
<html>
    <head>
        <title>Esto es busquedaOfertas.jsp</title>
    </head>
    <body>
        <form:form modelAttribute="bo">
            <form:input path="precioMax"/> <br/>
            <form:select path="tipoHab">
                <form:option value="individual"/>
                <form:option value="doble"/>
            </form:select>
            <input type="submit" value="buscar"/>
        </form:form>
    </body>
</html>
```

Obsérvense varios puntos interesantes:

- La vinculación entre el formulario y el objeto `BusquedaOfertas` del modelo se hace con el atributo `modelAttribute` de la etiqueta "form". Como es lógico, se le ha dado el mismo nombre ("bo") que en el código java de `preparaForm`
- Las etiquetas de Spring para formularios son muy similares a su contrapartida HTML. El atributo `path` de cada una indica la propiedad de `BusquedaOfertas` a la que están asociadas.
- El botón de enviar del formulario no necesita usar ninguna etiqueta propia de Spring, es HTML convencional
- El formulario no tiene "action" ya que llamará a la misma página. Implícitamente Spring lo convierte en un formulario con action vacío y método POST.

Llegados a este punto, el usuario rellena el formulario, lo envía y nosotros debemos procesarlo. Esto lo hace el controlador en el método `procesaForm`. Veamos su código:

```
@RequestMapping(method=RequestMethod.POST)
public String procesaForm(@ModelAttribute("bo") BusquedaOfertas bo,
                          BindingResult result, ModelMap modelo) {
   //validamos los datos
   if (bo.getPrecioMax()<0)
       result.rejectValue("precioMax", "intPositivo");
   //si Spring o nosotros hemos detectado error, volvemos al
formulario
   if (result.hasErrors()) {
       return "busquedaOfertas";
   }
   //si no, realizamos la operación
   modelo.addAttribute("ofertas", miGestor.BuscaOfertas(bo));
   //y saltamos a la vista que muestra los resultados
   return "listaOfertas";
 }
```

Antes de ejecutar este método Spring habrá tomado los datos del formulario y creado un

`BusquedaOferta` con ellos. Lo hacemos accesible al método a través del parámetro `bo`, enlazándolo con el atributo "bo" del modelo gracias a la anotación `@ModelAttribute("bo")`.

Antes de disparar la lógica de negocio hay que **validar los datos**. Spring habrá hecho una pre-validación comprobando que los datos son del tipo adecuado para encajar en un `BusquedaOfertas`: por ejemplo, que el precio es convertible a `int`. El resultado de esta "pre-validación" es accesible a través del segundo parámetro. "Semi-mágicamente" cuando tenemos un parámetro de tipo `BindingResult` que viene justo después de un atributo del modelo Spring lo asocia al resultado de su validación. La clase `BindingResult` tiene métodos para averiguar qué errores se han producido, y como ahora veremos, añadir nuevos.

Nuestra lógica de negocio puede tener requerimientos adicionales a la mera conversión de tipos. Por ejemplo, en nuestro caso está claro que un precio no puede ser un valor negativo. Por tanto, lo comprobamos y si es negativo usamos el método `rejectValue` para informar de que hay un nuevo error. Este método tiene dos parámetros: el nombre de la propiedad asociada al error y la clave del mensaje de error. El mensaje estará guardado en un fichero properties bajo esa clave. Si hay errores retornamos a la vista del formulario. Si no, disparamos la lógica de negocio, obtenemos los resultados, los añadimos al modelo y saltamos a la vista que los muestra. Estas últimas operaciones son equivalentes a las que hacíamos en el ejemplo de MVC sin procesamiento de datos de entrada.

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 `dispatcher-servlet.xml` definimos un bean que representa al archivo de mensajes, simplemente necesitamos un bean de la clase `ResourceBundleMessageSource`cuyo `id` sea `messageSource`

```
<bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
     <property name="basename"
value="es/ua/jtech/spring/mvc/mensajesWeb"/>
</bean>
```

Y aquí tenemos el archivo `mensajesWeb.properties` (que, como indicaba la propiedad "basename", está colocado dentro del paquete `es.ua.jtech.spring.mvc`:

```
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.

Para mostrar los errores se usa la etiqueta `errors`. Dicha etiqueta tiene un atributo `path` con el mismo significado que en el resto. De este modo, para mostrar los mensajes de error asociados al campo "tipoMax" pondríamos:

```
<form:form modelAttribute="bo">
   <form:input path="precioMax"/> <form:errors path="precioMax"/>
<br/>
   ...
```

5.2. Controlador "clásico"

La clase de la que vamos a heredar es `SimpleFormController`, ya que nos parece el más sencillo de usar, aunque en Spring hay otras variantes 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 otro lado, el procesamiento de los datos, una vez se ha rellenado el formulario, se hace en el método `onSubmit`. Este método recibe como parámetro un `Command`, del que tomaremos los datos. 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.

```
...
@Controller
@RequestMapping("/buscarOfertas.do")
public class BuscarOfertasController extends SimpleFormController {
        @Autowired
        private GestorOfertas miGestorOfertas;

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

```
        }


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

 }
```

Para validar los datos 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).