SPRING MVC

Índice

1 Configuración básica de Spring MVC

1.1 Ficheros de configuración

Una vez usados los asistentes para la creación del proyecto, se habrán generado dos ficheros de configuración: **applicationContext.xml** y **dispatcher-servlet.xml**, y varias líneas en web.xml:

```
<listener>
        <listener-class>
               org.springframework.web.context.ContextLoaderListener
        </listener-class>
</listener>

<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>

    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>
                org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>2</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>
```

El fichero dispatcher-servlet.xml es el fichero de configuración por defecto del controlador (siempre tiene como nombre nombre_del_servlet_controlador-servlet.xml); En este fichero se suele definir todo lo concerniente a Spring MVC. El “listener” **ContextLoaderListener** es la clase encargada de disparar WebApplicationContext, el marco de trabajo de Spring MVC. El parámetro **contextConfigLocation** permite definir varios ficheros de configuración para el marco de trabajo. Por defecto se usa **applicationContext.xml**. Se suele usar para definir el resto de beans de la aplicación, aunque se pueden usar otros ficheros.

NOTA: Los esquemas configurados por el asistente no suelen ser suficientes. En vez de añadirlos manualmente resulta más cómodo crear un fichero adicional con el asistente que englobe dichos esquemas, y a continuación copiar los espacios de nombres dentro de los ficheros de configuración.

1.2 Configuración del framework

Ya que se trata del marco MVC, lo escribiremos en dispatcher-servlet.xml. Necesitamos:

- <context:annotation-config/>. Activa la detección de anotaciones para configurar beans: @Autowired, @Resource...
- <context:component-scan base-package="paquete"/>. Activa la detección de @Component y sus tres estereotipos: @Controller, @Service, @Repository. Incluye por defecto “”annotation-driven”, por lo que en principio no hace falta volver a indicar la etiqueta anterior.
- <mvc:annotation-driven/>. Activa los controladores dirigidos por anotaciones. Configura muchos de los beans necesarios para usar Spring a través de anotaciones: Activa un controlador de mapeo (El “handler mapping” DefaultAnnotationHandlerMapping**)** para resolver las peticiones del cliente con

anotaciones, soporta la validación, el formateo y los servicios de conversión de Spring 3, admite las anotaciones Joda, soporta la conversión automática de las respuestas en formato JSON, etc.

Asimismo, necesitamos una forma de resolver las respuestas de los controladores: Qué páginas se van a dibujar. De momento usamos el más sencillo: **InternalResourceViewResolver**. Permite especificar un prefijo y un sufijo al texto devuelto por el controlador, para generar así el nombre real de la página a dibujar.

El fichero de configuración básico quedará así (no incluyo <mvc:annotation-driven/> ya que esta definida por defecto en el controlador de mapeo que estamos usando:

```
<context:component-scan base-package="org.javi.ejemplo"/>
<mvc:annotation-driven/>
<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
      p:prefix="/WEB-INF/paginas/"
      p:suffix=".jsp" />
```

1.3 Soporte i18n para idiomas

En toda aplicación necesitamos un fichero de recursos para permitir traducciones y resolver los mensajes de error. Spring proporciona (por ejemplo) el bean **ReloadableResourceBundleMessageSource**, que permite especificar qué ficheros ".properties" se van a usar.

```
<bean id="messageSource"
   class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <property name="basenames">
        <list>
            <value>classpath:/org/javi/ejemplo/vista/Textos</value>
            <value>classpath:/org/javi/ejemplo/vista/Errores</value>
        </list>
    </property>
</bean>
```

Spring dispone de varias clases para manejar el **Locale** del cliente:

- AcceptHeaderLocaleResolver. El objeto Locale se toma de la cabecera "accept-language" de la petición de cliente. Si se usa esta opción no se podrán realizar cambios de idioma. Es la que está activa por defecto.
- CookieLocaleResolver. Obtiene Locale de una cookie de cliente. Si no existe, se crea y se le envía a partir del Locale usado en ese momento (por lo general de la cabecera de petición).
- SessionLocaleResolver. Como en el caso anterior, pero se toma de la sesión.

Asimismo, Spring ofrece un **interceptor** (una clase que "intercepta" las peticiones del cliente para realizar ciertas tareas) que permite cambiar el idioma sin necesidad de escribir ningún controlador específico: **LocaleChangeInterceptor**; Permite especificar un parámetro de la petición del que se tomará el idioma y país a usar.

Para que funcione, tendremos que registrar el interceptor en los controladores de mapeo que estemos usando. Tradicionalmente eso implicaba definir explícitamente un "handler mapping" para poder definir a su vez que interceptores queremos usar. Ya que queremos controlar las peticiones a través de anotaciones necesitamos la clase **DefaultAnnotationHandlerMapping:**

```
<bean id="localeChangeInterceptor"
      class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
          <property name="paramName" value="idioma" />
</bean>

<bean id="localeResolver"
      class="org.springframework.web.servlet.i18n.SessionLocaleResolver"/>

<!—se supone que se elimina la etiqueta <mvc:annotation-driven/> -->
<bean id="urlMapping"
class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping">
```

```
    <property name="interceptors">
        <list>
            <ref local="localeChangeInterceptor"/>
        </list>
    </property>
</bean>
```

Pero esto plantea un problema: la etiqueta <mvc:annotation-driven/> ya define implícitamente ese controlador de mapeo. ¡No podemos definir dos veces el mismo controlador! Si hacemos eso, no funcionará correctamente.

Podemos eliminar la etiqueta “mvc” de nuestro fichero de configuración. La traducción funcionará, pero estropearemos toda la validación, formateo y conversión de Spring 3. La mejor solución es usar las nuevas etiquetas que proporciona esta versión de Spring, entre ellas **<mvc:interceptors/>**:

```
<bean id="localeResolver"
class="org.springframework.web.servlet.i18n.SessionLocaleResolver"/>

<mvc:interceptors>
    <bean id="localeChangeInterceptor"
    class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
        <property name="paramName" value="idioma" />
    </bean>
</mvc:interceptors>
```

Aparte de un código más limpio, ahora funcionará tanto la traducción como el resto de características de Spring 3. Cualquier petición a un controlador dirigido mediante anotaciones que tenga el parámetro “idioma” cambiará el Locale. Es habitual escribir el enlace de tal modo que la página actual siga siendo la misma:

```
<a href=”?idioma=en”>Inglés</a>
```

Si se desea usar el resolutor de mensajes desde un bean, únicamente tenemos que pedir a Spring que lo inyecte:

```
@Autowired
MessageSource mensajes;
```

Es habitual usarlo desde un controlador. Los métodos de **MessageSource** suelen necesitar el objeto Locale activo en ese momento. Para disponer de él basta con definir en el método del controlador un parámetro de tipo Locale. Spring nos pasará automáticamente dicho objeto:

```
@ResponseBody
@RequestMapping("/crear.html")
public Respuesta crear(@Valid Cliente c, BindingResult errores, Locale l) {
    ...
    String texto=this.ms.getMessage("cliente.crear.clave", null, l);
    ...
}
```

El método “getMessage()” está diseñado para usar las claves del fichero de recursos.

Si queremos hacer referencia al idioma desde la página JSP podemos usar EL:

```
${pageContext.response.locale}
```

La propiedad “locale” es el objeto Locale activo, por lo que podemos usar sus propiedades “language” o “country”:

```
<c:if test="${pageContext.response.locale.language=='es'}">
    <script src="../js/messages_es.js" type="text/javascript"></script>
</c:if>
```

1.4 Problemas con UTF-8

Por defecto, si configuramos la base de datos y las páginas Web en formato UTF-8 todo se almacenará y se mostrará correctamente. Pero no sucederá lo mismo si generamos el código con las bibliotecas de etiquetas XML de Spring para formularios. Si por ejemplo las usamos para dibujar el siguiente formulario de HTML:

```
<f:form commandName="usuario">
  ...
  <tr>
    <td><label><s:message code="entrada.et.nombre"/></label></td>
    <td><f:input path="nombre"/></td>
    <td><f:errors path="nombre"/></td>
  </tr>
  <tr>
    <td><label><s:message code="entrada.et.clave"/></label></td>
    <td><f:password path="clave"/></td>
    <td><f:errors path="clave"/></td>
  </tr>
  <tr>
  ...
</f:form>
```

Funcionará para recoger los datos o mostrarlos directamente desde el controlador, rellenando el comando “usuario” en este caso.

Sin embargo, cuando tiene que mostrar los parámetros de la petición anterior (la típica página JSP que primero presenta el formulario en blanco y después redibuja la página con el formulario relleno con lo escrito en la petición anterior) falla y lo muestra todo en ASCII.

Para evitarlo es necesario incorporar un filtro (si tienes varios debe ser el primero) en del descriptor de despliegue, **web.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns...>
    <filter>
        <filter-name>encoding-filter</filter-name>
        <filter-class>
            org.springframework.web.filter.CharacterEncodingFilter
        </filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <init-param>
            <param-name>forceEncoding</param-name>
            <param-value>true</param-value>
        </init-param>
    </filter>

    <filter-mapping>
        <filter-name>encoding-filter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    ...
</web-app>
```

Esta solución la he copiado de https://stackoverflow.com/questions/5928046/spring-mvc-utf-8-encoding. En ese enlace encontrarás otras posibles maneras de corregirlo, dependiendo del tipo de proyecto y de cómo has configurado Spring.

1.5 Excepciones declarativas

Algo habitual es que por defecto las excepciones que lancemos desde el controlador se resuelvan a partir de la configuración del fichero de XML. Usaremos la clase de Spring **SimpleMappingExceptionResolver**, que asocia un nombre de clase con una página.

```
<bean
class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
        <map>
            <entry key="DatosException" value="errordatos"/>
            <entry key="SeguridadException" value="errorseguridad"/>
        </map>
    </property>
    <property name="defaultErrorView" value="errorgeneral"/>
</bean>
```

En este caso las excepciones DatosException y SeguridadException tienen página propia, mientras que el resto de excepciones se visualizarán con “errorgeneral”. Por supuesto, podemos hacer nuestro propio gestor de excepciones implementado ciertas interfaces o extendiendo la clase SimpleMappingExceptionResolver.

Los mensajes de error (el **getMessage()** de la excepción) se pueden recuperar siguiendo los estándares de Servlets:

- **exception** representa la excepción: ${exception.message} es el mensaje de error.
- **“javax.servlet.error.exception_type”** es el nombre cualificado de la excepción. Se recuperaría con ${requestScope[“javax.servlet.error.exception_type”]}
- **“javax.servlet.error.status_code”** es el código de error.

1.6 Soporte para Tiles 2

Es aconsejable añadir el soporte para Tiles a Spring, ya que mejora notablemente el desarrollo de las páginas JSP. Funciona como un resolutor de vistas más.

Tiles no está soportado directamente por Spring, por lo que habrá que bajar los “.jar” adecuados del proyecto apache. Es recomendable bajar tanto las clases (tiles-2.2.2-bin.zip) como la documentación (tiles-2.2.2-docs.zip), para poder usarlo posteriormente de manera cómoda. Los ficheros jar que hay que añadir a la aplicación (o a una biblioteca):

- Clases para Tiles:/tiles-api-2.2.2.jar, /tiles-core-2.2.2.jar, /tiles-jsp-2.2.2.jar, /tiles-servlet-2.2.2.jar, /tiles-servlet-wildcard-2.2.2.jar, /tiles-template-2.2.2.jar, /tiles-el-2.2.2.jar.
- Ficheros adicionales: /lib/commons-beanutils-1.8.0.jar, /lib/commons-digester-2.0.jar, /lib/jcl-over-slf4j-1.5.8.jar, /lib/slf4j-api-1.5.8.jar, /lib/optional/slf4j-jdk14-1.5.8.jar.
- Documentación: /apidocs.

Habrá que crear asimismo un fichero de XML para las definiciones de plantillas, con la DTD adecuada. Por ejemplo podemos crear el fichero “/WEB-INF/definiciones-tiles.xml”:

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE tiles-definitions PUBLIC
    "-//Apache Software Foundation//DTD Tiles Configuration 2.0//EN"
    "http://tiles.apache.org/dtds/tiles-config_2_0.dtd">

  <tiles-definitions>
```

```
<definition name=".plantilla" template="/WEB-INF/paginas/varios/plantilla.jsp">
    <put-attribute name="menu" value="/WEB-INF/paginas/varios/barramenu.jsp"/>
    <put-attribute name="pie" value="/WEB-INF/paginas/varios/pie.jsp"/>
    <put-attribute name="encabezado" value="/WEB-
INF/paginas/varios/encabezado.jsp"/>
    <put-attribute name="contenido" value="/WEB-
INF/paginas/varios/vacia.jsp"/>
    <put-attribute name="titulo" value="general.titulo"/>
</definition>

<definition name="entrada" extends=".plantilla">
    <put-attribute name="titulo" value="entrada.titulo"/>
    <put-attribute name="contenido" value="/WEB-
INF/paginas/entrada/entrada.jsp"/>
</definition>

</tiles-definitions>
```

La página “plantilla.jsp” tendrá las etiquetas Tiles necesarias para definir una plantilla básica:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles"%>
<%@taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<!DOCTYPE html>
<html>

<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>
        <tiles:useAttribute name="titulo" id="clave"/>
        <spring:message code="${clave}"/>
    </title>
    <link href="/ejemplospring/estilos/limpiar.css" rel="stylesheet"/>
    <link href="/ejemplospring/estilos/menu.css" rel="stylesheet"/>
    <link href="/ejemplospring/estilos/estilo.css" rel="stylesheet"/>
</head>

<body>
    <div id="contenedor">
        <div id="menu">
            <tiles:insertAttribute name="menu"/>
        </div>
        <div id="encabezado">
            <tiles:insertAttribute name="encabezado"/>
        </div>
        <div id="contenido">
            <tiles:insertAttribute name="contenido"/>
        </div>
        <div id="pie">
            <tiles:insertAttribute name="pie"/>
        </div>
    </div>
</body>

</html>
```

Hay que configurar a Spring para que resuelva nombres de Tiles en vez de páginas JSP. Si se desea, se puede escribir de tal modo que en primer lugar trate de resolver el nombre de plantilla, y sólo si éste no existe busque un nombre de página JSP. Sólo hay que usar la propiedad **order**.

```
<bean id="viewResolver"

class="org.springframework.web.servlet.view.InternalResourceViewResolver"
      p:prefix="/WEB-INF/paginas/"
      p:suffix=".jsp"
```

```
        p:order="1"/>

    <bean id="tilesViewResolver"
        class="org.springframework.web.servlet.view.tiles2.TilesViewResolver"
        p:order="0"/>
```

Se ha añadido un nuevo resolutor de vistas, **TilesViewResolver**, que tiene prioridad sobre el anterior.

Por último, sólo nos queda indicar a Tiles qué ficheros contienen las definiciones de las plantillas. No se hace a través de propiedades del resolutor, sino con un nuevo bean de clase **TilesConfigurer**:

```
    <bean id="tilesConfigurer"
    class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
            <property name="definitions">
                <list>
                    <value>/WEB-INF/definiciones-tiles.xml</value>
                </list>
            </property>
        </bean>
```

Por supuesto, se pueden usar otros resolutores, como ResourceBundleViewResolver, al que habría que asociarle la vista **TilesView** en el fichero de propiedades.

1.7 Soporte para Tiles 3

Es similar al anterior, aunque se han producido algunos cambios. Los archivos jar necesarios son:

- Clases para Tiles: tiles-api-3.0.8.jar, tiles-core-3.0.8.jar, tiles-el-3.0.8.jar, tiles-jsp-3.0.8.jar, tiles-servlet-3.0.8.jar, tiles-template-3.0.8.jar, tiles-extras-3.0.8.jar, tiles-request-servlet-1.0.7.jar, tiles-request-servlet-wildcard-1.0.7.jar, tiles-request-api-1.0.7.jar, tiles-request-jsp-1.0.7.jar, tiles-autotag-core-runtime-1.2.jar.
- Ficheros adicionales: lib/commons-beanutils-1.8.0.jar, lib/commons-digester-2.0.jar, lib/jcl-over-slf4j-1.7.6.jar, lib/slf4j-api-1.7.6.jar.

La cabecera del fichero de definiciones también cambia para hacer referencia a la nueva versión:

```
    <?xml version="1.0" encoding="UTF-8" ?>

    <!DOCTYPE tiles-definitions PUBLIC
        "-//Apache Software Foundation//DTD Tiles Configuration 3.0//EN"
        "http://tiles.apache.org/dtds/tiles-config_3_0.dtd">
```

Las etiquetas XML que usamos en las plantillas se han dividido en dos bibliotecas distintas, por lo que tendremos que modificarlas:

```
    <%@page contentType="text/html" pageEncoding="UTF-8"%>
    <%@taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles"%>
    <%@taglib uri="http://tiles.apache.org/tags-tiles-extras" prefix="tilesx"%>
    ...
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>
            <tilesx:useAttribute name="titulo" id="clave"/>
            <spring:message code="${clave}"/>
    ...
    <body>
        <div id="contenedor">
            <div id="menu">
                <tiles:insertAttribute name="menu"/>
    ...
```

Y por supuesto, el cambio de versión también hay que reflejarlo en el fichero de configuración de Spring:

```
<bean id="tilesViewResolver"
      class="org.springframework.web.servlet.view.tiles3.TilesViewResolver"
      p:order="0"/>

<bean id="tilesConfigurer"
class="org.springframework.web.servlet.view.tiles3.TilesConfigurer">
        <property name="definitions">
            <list>
                <value>/WEB-INF/definiciones-tiles.xml</value>
            </list>
        </property>
</bean>
```

1.8 Objetos JSON

En capítulos posteriores veremos cómo se deben crear los métodos del controlador para que envíen al cliente objetos JSON en respuesta a una petición AJAX.

Hasta la versión 3 de Spring no había que hacer nada especial para convertir un objeto Java a un texto con aspecto de objeto de JavaScript. Sin embargo en las últimas versiones del framework la configuración por defecto ha cambiado. Ahora, si no se dice lo contrario, Spring "deduce" qué tipo de respuesta acepta el cliente por la extensión del fichero que ha pedido, en vez de usar la cabecera de petición "accept".

Lógicamente debemos decirle que deje de hacer eso, y que por tanto vuelva a leer las cabeceras de petición del cliente para saber el tipo de información que acepta ("application/json" en el caso de una petición AJAX escrita de la manera adecuada):

```
<mvc:annotation-driven content-negotiation-manager="contentNegotiationManager"/>

<bean id="contentNegotiationManager"
   class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean">
   <property name="favorPathExtension" value="false" />
</bean>
```

Configuramos un bean con la clase que usa Srping para decidir los contenidos aceptados por el cliente, y lo configuramos para que deje de usar la extensión del fichero pedido.

Para enganchar este nuevo bean usamos el atributo **content-negotiation-manager** de la anotación mvc:annotation-driven.

1.9 El fichero de configuración completo

Aplicando todos los pasos, el fichero de configuración quedará de este modo:

```
<context:component-scan base-package="org.javi.ejemplo"/>

<mvc:annotation-driven content-negotiation-manager="contentNegotiationManager"/>

<bean id="contentNegotiationManager"
   class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean">
   <property name="favorPathExtension" value="false" />
</bean>

<bean id="viewResolver"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver"
      p:prefix="/WEB-INF/paginas/"
      p:suffix=".jsp"
      p:order="1"/>

<bean id="tilesViewResolver"
      class="org.springframework.web.servlet.view.tiles2.TilesViewResolver"
      p:order="0"/>
```

```
<bean id="tilesConfigurer"
      class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
    <property name="definitions">
        <list>
            <value>/WEB-INF/definiciones-tiles.xml</value>
        </list>
    </property>
</bean>

<bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <property name="basenames">
        <list>
            <value>classpath:/org/javi/ejemplo/vista/Textos</value>
            <value>classpath:/org/javi/ejemplo/vista/Errores</value>
        </list>
    </property>
</bean>

<bean
class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
        <map>
            <entry key="DatosException" value="error"/>
            <entry key="SeguridadException" value="error"/>
        </map>
    </property>
</bean>

<bean id="localeResolver"
      class="org.springframework.web.servlet.i18n.SessionLocaleResolver"/>

<mvc:interceptors>
    <bean id="localeChangeInterceptor"
          class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
        <property name="paramName" value="idioma" />
    </bean>
</mvc:interceptors>
```

2 El código de Java

Vamos a suponer una aplicación muy sencilla. Unos “jefes” quieren ver datos de sus “empleados”. La pantalla inicial preguntará al jefe su login y clave. Si la escribe correctamente, accederá al menú principal de la aplicación. Sólo habrá tres páginas JSP: la de errores (ya está configurada con excepciones declarativas), la de bienvenida y la de acceso al sistema.

Lógicamente, una vez que un jefe ha entrado no tendremos que volver a pedirle la clave, por lo que recordaremos en la sesión que ya ha entrado, así como el resto de datos que nos parezca oportuno.

De momento, para simplificar el ejemplo, vamos a obviar la parte de acceso a datos.

2.1 El Controlador

Sólo necesitamos un controlador, que pregunte la clave cuando sea necesario:

```
package org.javi.ejemplo.controlador;

@Controller
public class ControladorEntrada {

@Autowired
private GestorJefe gestor;

@Autowired
private DatosDeSesion ds;

    @RequestMapping("/entrada.html")
    public String entrada(@Valid Usuario usuario,Errors resul,Map modelo)
    throws DatosException, NoEncontradoException  {
        if (ds.getHaEntrado()) return "bienvenida";
        if (usuario.isVacio() || resul.hasErrors()) return "entrada";

        try {
            Jefe jefe=gestor.getJefe(usuario.getLogin(),usuario.getClave());
            ds.setJefe(jefe);
            return "bienvenida";
        } catch (NoEncontradoException ex) {
            modelo.put("fallo", true);
            return "entrada";
        }
    }
}
```

Un controlador es cualquier clase anotada con **@Controller**. Atenderá a todas las peticiones que coincidan con lo declarado en **@RequestMapping**. Hay decenas de combinaciones de posibles firmas para el método anotado, así como de formas de completar la anotación.

Por supuesto, la anotación @ Controller funciona por que se supone que hemos configurado Spring de la forma adecuada. Consulta en el capítulo anterior la etiqueta de XML **<mvc:annotation-driven/>**.

En el capítulo siguiente veremos con detalle cómo crear un controlador.

En este ejemplo, el método se lanzará cuando el usuario pregunte por la página “/entrada.html”. Se supone que en dicha página hay un formulario definido con etiquetas de Spring que rellenará el javabean “usuario”. Como veremos más adelante, es automático. Sólo hace falta que coincidan el nombre de las propiedades de Usuario con el nombre de los campos de formulario.

El parámetro está anotado con **@Valid**, por lo que se validará siguiendo las reglas definidas en “algún sitio”. El resultado de la validación lo podemos ver (y modificar) con el objeto **Errors** o **BindingResult**. En el capítulo dedicado a la validación mediante anotaciones se verá la anotación de Spring **@Validated**, que mejora en ciertos aspectos a la anotación estándar.

El controlador necesita dos beans para trabajar: Uno que le permita acceder al modelo y otro para acceder a la sesión. Ambos son inyectados automáticamente por Spring al estar anotados con **@Autowired**.

2.2 La Sesión

No hace falta capturar los eventos de sesión ni nada parecido para acceder a la misma. Basta con anotar una clase cualquiera de esta forma:

```
package org.javi.ejemplo.sesion;

@Component
@Scope(proxyMode=ScopedProxyMode.INTERFACES,value="session")
public class DatosDeSesionImpl implements DatosDeSesion {
private Jefe jefe;

    public DatosDeSesionImpl() {
    }

    @Override
    public void setJefe(Jefe jefe) {
        this.jefe = jefe;
    }

    @Override
    public int getCodigo() {
        if (jefe==null) return -1;
        else return jefe.getCodigo();
    }

    @Override
    public String getNombre() {
        if (jefe==null) return "";
        else
            return jefe.getNombre() + " " + jefe.getApellidoUno();
    }

    @Override
    public boolean getHaEntrado() {
        return jefe!=null;
    }

}
```

la anotación **@Component** define a esta clase como un bean de Spring. Se ha modificado el alcance de la misma a través de la anotación **@Scope**.

Esta anotación tiene un atributo “extraño”: **proxyMode=ScopedProxyMode**. Spring sólo creará una instancia del controlador para todas las peticiones de todos los clientes, pero obviamente no hará esto con la sesión. ¿Cómo es posible que pueda inyectar en un objeto que sólo crea una vez otro que es creado para cada nuevo cliente?. Spring usa AOP para ello. Por eso tenemos que definir el tipo de “objeto proxy” (ver AOP en el manual de referencia) que creará.

Por lo general es preferible el uso de interfaces para esa tarea. Además, siempre es más elegante inyectar dependencias a través de interfaces que a través de clases. Por eso la sesión implementa la interfaz **DatosDeSesion**. Es algo que es recomendable hacer para todos los objetos que se inyectan, pero especialmente en aquellos que van a ser tratados mediante AOP: beans con “scope” extraños, gestores de datos (transacciones), etc.

Si se desea, también se puede definir el bean de sesión en el fichero de configuración:

```
bean id="datosSesion"
     class="org.javi.ejemplo.sesion.DatosDeSesionImpl" scope="session">
            <aop:scoped-proxy/>
</bean>
```

Se puede hacer referencia a la sesión desde una página JSP usando EL, aunque la sintaxis es un poco incómoda:

```
${sessionScope['scopedTarget.nombreBeanSesion'].propiedad}
```

Para que la sesión mediante AOP funcione es necesario hacer algunos cambios en las bibliotecas del proyecto. En la versión tres de Spring hay que añadir un fichero jar al proyecto y **reemplazar** otro de la biblioteca suministrada con Netbeans. El fichero a añadir es **aopaliance-1.0.jar**. El fichero a reemplazar de la bilioteca es **cglib-2.2.jar**, y se debe cambiar por **cglib-nodep-2.2.2.jar**. No deberías tener problemas para encontrarlos en Internet.

En la versión cuatro de Spring sólo es necesario añadir **aopaliance-1.0.jar** para que todo funcione.

2.3 La Vista

La clase Usuario es un javabean sin nada especial. Sólo usa la validación mediante anotaciones por comodidad:

```
package org.javi.ejemplo.vista;

public class Usuario implements Serializable{
    @Size(min=3,max=40)
    private String login;
    @Size(min=8, max=40)
    private String clave;

    public Usuario() {
    }

    public String getLogin() {
        return login;
    }

    public void setLogin(String nombre) {
        this.login = nombre;
    }

    public String getClave() {
        return clave;
    }

    public void setClave(String clave) {
        this.clave = clave;
    }

    public String toString() {
        return login + ":" + clave;
    }
    public boolean isVacio() {
        return (login==null & clave==null);
    }
}
```

El truco para engancharlo al controlador no está en esta clase, sino en las etiquetas empleadas en la página JSP:

```
<%@taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<h1><spring:message code="entrada.mensaje"/></h1>

<form:form commandName="usuario" action="entrada.html">
    <table class="lista">
        <tr>
            <td><label><spring:message
code="entrada.et.login"/></label></td>
            <td><form:input path="login"/></td>
            <td class="mal"><form:errors path="login"/></td>
        </tr>
        <tr>
            <td><label><spring:message
code="entrada.et.clave"/></label></td>
            <td><form:input path="clave"/></td>
            <td class="mal"><form:errors path="clave"/></td>
        </tr>
        <tr>
            <td colspan=3>
                <input type="submit"
                 value="<spring:message code="entrada.et.boton"/>"/>
            </td>
        </tr>
    </table>
</form:form>

<c:if test="${fallo}">
    <p class="mal"><spring:message code="entrada.mal"/></p>
</c:if>
```

El formulario definido con las etiquetas “form” de Spring hace referencia a un **comando** (un javabean asociado a un formulario) llamado **usuario**. Ese nombre es el de la clase que usamos en el controlador:

```
@RequestMapping("/entrada.html")
    public String entrada(@Valid Usuario usuario,Errors resul,Map modelo)
    throws DatosException, NoEncontradoException  {
...
}
```

Si no se indica lo contrario (más anotaciones), los datos del formulario rellenarán un parámetro de clase Usuario en el controlador. Por supuesto, se supone que el nombre de las propiedades coinciden con los “path” del formulario.

Si el controlador decide volver a la página, es posible que se hayan producido errores. Las etiquetas **<form:errors/>** se encargarán de presentarlos. Por supuesto, es necesario que se hayan definido una serie de claves en el fichero de recursos:

```
entrada.mal=No hay un usuario con ese login o clave
Size=El tamaño de "{0}" debe estar entre {2} y {4}
```

Ver “validación mediante anotaciones” para comprender las claves empleadas.

2.4 Configuración del DataSource para JDBC

Por lo general la conexión a los datos se hará a partir de un DataSource. En nuestro caso estamos en el entorno de un contenedor Web capaz de configurar por sí mismo el DataSource y el Pool de conexiones, por lo que nos limitaremos a pedirlo a través de JNDI. En el fichero ApplicationContext.xml:

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/empresa"/>
```

Podemos gestionar nosotros mismos las conexiones y cierres con el DataSource, por ejemplo a través de objetos de la sesión, pero no merece la pena. Es mucho más cómodo usar **JdbcTemplate** o alguno de sus

derivados, que automáticamente abrirán o cerrarán el Pool de conexiones. Lo habitual es hacerlo cuando se inyecta el DataSource a los objetos de la “capa de servicio”:

```
@Service
public class GestorJefeImpl implements GestorJefe {

private SimpleJdbcTemplate plantilla;

    @Resource(name="dataSource")
     public void setDataSource(DataSource ds) {
          this.plantilla=new SimpleJbcTemplate(ds);
    }

     ...
}
```

JdbcTemplate es “**thread-safe**”, es decir, está programado de tal modo que un mismo objeto puede ser ejecutado a la vez en diferentes hilos. Sin embargo no se suele hacer un único objeto para toda la aplicación, sino que cada clase de servicio crea el suyo propio.

Si no se desea usar los recursos del servidor o se está en un entorno “standalone” tambiéns e puede configurar manualmente el DataSource:

```
<bean name="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
          <property name="driverClassName"
value="sun.jdbc.odbc.JdbcOdbcDriver"/>
          <property name="url" value="jdbc:odbc:empresaODBC"/>
          <property name="username" value=""/>
          <property name="password" value=""/>
</bean>
```

Por lo general, los valores de configuración se encuentran en un fichero de propiedades. Spring proporciona la clase **PropertyPlaceholderConfigurer** para leer automáticamente de un fichero de propiedades y usar sus entradas donde convenga. Si tenemos el fichero “/WEB-INF/jdbc.properties”:

```
jdbc.clase=sun.jdbc.odbc.JdbcOdbcDriver
jdbc.cadena=jdbc:odbc:empresaODBC
jdbc.usuario=""
jdbc.clave=""
```

Los beans necesarios para que Spring genere el DataSource serán:

```
    <bean id="propertyConfigurer"

    class="org.springframework.beans.factory.config.PropertyPlaceholderConfi
    gurer"
        p:location="/WEB-INF/jdbc.properties" />

    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource"
        p:driverClassName="${jdbc.clase}"
        p:url="${jdbc.cadena}"
        p:username="${jdbc.usuario}"
        p:password="${jdbc.clave}" />
```

La clase **PropertyPlaceholderConfigurer** presenta los contenidos del fichero de propiedades como **variables de entorno**. La sintaxis para acceder a una de esta variables dentro del fichero de configuración de Spring es **${variable}**.

2.5 Configuración de JPA

Podemos configurar JPA de múltiples maneras, pero lo más cómodo será permitir a Spring que gestione tanto la factoría de EntityManager como las transacciones: Haremos una instalación gestionada por contenedor, pero no por el contenedor Glassfish, sino por Spring.

El primer paso es crear el DataSource y pool de conexiones (ver apartado anterior). Más adelante podremos hacer referencia al DataSource desde el fichero de configuración de Spring o bien desde la unidad de persistencia.

A continuación escribiremos la unidad de persistencia. Aunque el asistente de Netbeans cree el fichero **persistence.xml** en la carpeta de "configuración", lo moveremos al classpath de nuestro proyecto, por ejemplo dentro del modelo. Si la dejamos en su ubicación por defecto Glassfish la detectará automáticamente y nos obligará a usar sus transacciones gestionadas por contenedor (JTA). Como quiero usar las de Spring y no me apetece volverme loco con la configuración, lo más sencillo es cambiar de sitio el fichero.

Si usamos el asistente de NetBeans para crear el fichero, deberemos dejar sin marcar la casilla "Use Java Transaction APIs", y añadiremos manualmente las entidades que queremos gestionar. El fichero quedará:

```
<persistence version="2.0" ... >
  <persistence-unit name="empresaPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <non-jta-data-source>jdbc/biblioteca</non-jta-data-source>
    <class>com.javi.empresa.modelo.entidad.Jefe</class>
    <class>com.javi.empresa.modelo.entidad.Empleado</class>
    <class>com.javi.empresa.modelo.entidad.Persona</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    <properties>
      <property name="eclipselink.ddl-generation" value="create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

La gestión del contexto de persistencia lo va a realizar Spring. Desde el punto de vista del servidor, va a ser nuestras propias clases quienes se van a encargar de esa tarea: No las hemos escrito nosotros, pero Glassfish no distingue entre clases compiladas en nuestro ordenador o bajadas de Internet (Spring). Por lo que a él respecta, la gestión será manual.

Ahora tenemos que indicar a Spring que queremos que sea él quien gestione la factoría, y que queremos que nos suministre EntityManager gestionadas por contenedor: Se trata sencillamente de configurar el bean adecuado en **dispatcher-servlet.xml**:

```
<bean id="entityManagerFactory"

class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="persistenceXmlLocation"

value="classpath:/com/javi/empresa/modelo/persistence.xml"/>
    <property name="persistenceUnitName" value="empresaPU"/>
    <property name="jpaPropertyMap">
        <props>
            <prop key="eclipselink.weaving">false</prop>
        </props>
    </property>
</bean>
```

La clase de Spring que vamos a usar es **LocalContainerEntityManagerFactoryBean**. Proporciona EntityManager gestionada por contenedor, y nos permite manejar las transacciones a través de Spring (lo veremos a continuación). Tenemos que especificar dónde está el fichero persistence.xml, la unidad de persistencia que queremos usar ("empresaPU" en este caso) y un extraño atributo propio de la biblioteca que estamos usando para implementar JPA.

Es complejo de explicar (ver AOP para más información), pero si dejamos "**eclipselink.weaving"** a true (su valor por defecto), tendremos dos cargadores de clases gestionando las entidades, lo cual puede provocar un error de casting muy extraño: Por ejemplo, podría producirse un error del tipo "no puedo convertir objetos de la clase Jefe a objetos de la clase Jefe". Al tener dos cargadores distintos, habría dos objetos Class para Jefe, por lo que de vez en cuando nos diría que no puede convertir objetos de clase Jefe (del primer cargador) a objetos de clase Jefe (del segundo cargador).

No es necesario indicar la fuente de datos en la unidad de persistencia; Si hemos configurado un pool de conexiones en el contenedor podemos usarlo en Spring y por tanto en el bean que estamos configurando:

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/biblioteca"/>
```

La propiedad de la factoría:

```
<bean id="entityManagerFactory"
...
    <property name="dataSource" ref="dataSource"/>
...
</bean>
```

A continuación tenemos que configurar las transacciones que usaremos con JPA:

```
<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>

<tx:annotation-driven transaction-manager="transactionManager"/>
```

Spring proporciona diferentes gestores de transacciones; Utilizaremos uno u otro en función de la plataforma que estemos usando. En este caso, ya que queremos gestionar los datos con JPA necesitamos la clase **JpaTransactionManager**. Cuando definimos el bean tenemos que indicarle asimismo la factoría empleada.

La etiqueta **annotation-driven** nos permite trabajar con las transacciones mediante anotaciones. Por supuesto, esta no es la única manera de hacerlo. Ver el manual de referencia para más información.

Una vez definido el fichero de configuración, escribir el modelo es sencillo. A continuación se muestra un ejemplo en el que se ha dividido el código en una clase "Repository" y otra (u otras) de "Service". Por supuesto, es sólo un ejemplo: No tiene por qué existir un repositorio, ni hay que agrupar los métodos de esta manera.

```
@Repository
public class ModeloGeneralImpl implements ModeloGeneral {

@PersistenceContext
private EntityManager em;

    @Override
    public void guardar(Object objeto) throws DatosException {
        try {
            em.persist(objeto);
        } catch (Exception ex) {
            throw new DatosException("Error al Crear: " + ex.getMessage());
        }
    }

    @Override
    public Object modificar(Object objeto) throws DatosException {
        try {
            return em.merge(objeto);
        } catch (Exception ex) {
            throw new DatosException("Error al Modificar: "
                                              + ex.getLocalizedMessage());
        }
    }
}
```

La anotación **@Repository** declara un bean. En vez de escribir la definición en el fichero de configuración, Spring permite hacer lo mismo anotando el código de Java. Como se ha explicado anteriormente, existen cuatro anotaciones para definir beans. Todas salvo @Controller hacen exactamente lo mismo. La diferencia es "estilística": Queda más elegante indicar qué tipo de código estamos escribiendo. Y es posible que futuras versiones de Spring sí traten de forma distinta estas anotaciones.

- @Component. Es la anotación base. Es genérica, no indica qué tarea realiza el bean.
- @Repository. Almacén de datos, "repositorio". Son las clases que trabajan directamente con el gestor de entidades o con las sentencias de SQL.
- @Service. Clases del modelo que internamente utilizan los "repositorios", y que a su vez son usadas por el controlador u otras clases externas al modelo.
- @Controller. Los beans marcados con esta anotación se comportan de manera muy distinta a los anteriores. Ver la sección del manual dedicada a los controladores para más información.

La anotación **@PersistenceContext** es estándar de JPA. En un entrono gestionado por contenedor, inyecta el EntityManager que usaremos en la clase. Se supone que el contenedor es quien administra la factoría y los gestores de entidades creados por ésta, por lo que no tenemos que preocuparnos de cómo se abren o cierran ni de cuándo lo hacen.

Este código no funcionaría, faltan las transacciones. Ya que el repositorio es común para todo el modelo, es preferible definirlas en la capa de servicio y hacer el programa más genérico:

```
@Service
public class GestionarJefeImpl implements GestionarJefe {

@Autowired
private ModeloGeneral general;

    @Override
    @Transactional(readOnly=true)
    public Jefe getJefe(String login, String clave) throws

DatosException,NoHayDatosException {
            return (Jefe)
this.general.getResultadoSimple("Jefe.porLoginClave",
                                                Usuario.class,
                                                new Campo("login",login),
                                                new Campo("clave",clave));
    }

    @Override
    @Transactional
    public void crearSocio(Socio socio) throws DatosException {
        this.general.guardar(socio);
    }
...
}
```

La anotación **@Transactional** se usa a nivel de método. Todo lo que se ejecute en dicho método se hará dentro de la misma transacción. Consultar el manual de referencia para más información sobre los valores con los que se puede configurar la anotación.

3 El Controlador

Las posibilidades de configuración y programación son muy amplias. Podemos definir diferentes tipos de controladores, y en el caso de que, siguiendo el ejemplo visto en capítulos anteriores, definamos un controlador dirigido por anotaciones podemos escribirlo de decenas de formas distintas.

3.1 Definir un controlador

Como ya hemos visto, usaremos la etiqueta de XML **<mvc:annotation-driven/>** en el fichero de configuración de Spring para definir los controladores mediante la anotación **@Controller**:

```
@Controller
public class ControladorPedidos {
    ...
}
```

Lógicamente podemos crear varias clases controladoras, agrupando las peticiones a las que responderán según nos convenga.

Otra manera de crear un controlador es extender la clase **AbstractController** o **MultiActionController**:

```
public class ControladorDos extends AbstractController {
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest req,
                                HttpServletResponse res) throws Exception {
        ModelAndView mv=new ModelAndView("dos");
        mv.addObject("mensaje", "controlador dos");
        return mv;
    }
}
```

No suelo usarlas, ya que desde mi punto de vista se han quedado muy anticuadas.

3.2 Atender una petición

Spring proporciona varios beans para decidir qué vista o clase va a atender una petición del usuario. Generalmente utilizaré la anotación @RequestMapping dentro del propio controlador, pero no es la única forma. A continuación veremos algunas de ellas.

3.2.1 SimpleUrlHandlerMapping

Spring proporciona la clase **SimpleUrlHandlerMapping**:

```
<bean id="urlMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="index.html">indexController</prop>
            <prop key="uno.html">controladorUno</prop>
            <prop key="dos.html">controladorDos</prop>
        </props>
    </property>
</bean>

<bean name="controladorUno" class="com.javi.es.ejUno.ControladorUno"/>
<bean name="controladorDos" class="com.javi.es.ejUno.ControladorDos"/>
```

```
<bean name="indexController"
class="org.springframework.web.servlet.mvc.ParameterizableViewController"
      p:viewName="index" />

<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
      p:prefix="/WEB-INF/jsp/"
      p:suffix=".jsp" />
```

A partir de la petición del usuario se describe qué controlador va a atender la petición. En la propiedad **mappings** se indica la petición del usuario y el nombre del bean controlador a la que se redirige. Admite muchas más propiedades, como **pathMatcher**, que permite usar caracteres comodín.

En el ejemplo he usado la clase **ParameterizableViewController**. Es un controlador proporcionado por Spring para cuando lo único que queremos es redirigir la petición directamente a una vista. En este caso, queríamos que la petición "index.html" lanzara la página "/WEB-INF/jsp/index.jsp". Para una tarea tan sencilla no es necesario escribir un controlador propio.

3.2.2 BeanNameUrlHandlerMapping

Es el bean configurado por defecto en Spring, y por tanto el que se usará si no se configura ningún controlador de mapeos. El **nombre del bean** debe coincidir con la petición del usuario. El ejemplo anterior quedaría:

```
<bean name="/uno.html" class="com.javi.es.ejUno.ControladorUno"/>
<bean name="/dos.html" class="com.javi.es.ejUno.ControladorDos"/>

<bean name="/index.html"
class="org.springframework.web.servlet.mvc.ParameterizableViewController"
      p:viewName="index" />

<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
      p:prefix="/WEB-INF/jsp/"
      p:suffix=".jsp" />
```

3.2.3 ControllerClassNameHandlerMapping

Este controlador de mapeo permite redirigir la petición en función del nombre de ésta y del **nombre de la clase** del controlador. Siguiendo con el ejemplo, si queremos que las peticiones "uno.html" y "dos.html" sean atendidas por los controladores anteriores, deberemos cambiar el nombre de los mismos a "UnoController" y " DosController" respectivamente:

```
<bean class="org.springframework.web.servlet.mvc.support.
                                        ControllerClassNameHandlerMapping"/>

<bean class="com.javi.es.ejUno.UnoController"/>
<bean class="com.javi.es.ejUno.DosController"/>
```

Las peticiones **"/uno*"** y "**/dos***" se redirigirán al controlador correspondiente

Como siempre, admite múltiples propiedades: Caracteres comodín, prefijos, distinguir entre minúsculas y mayúsculas...

```
<bean class="org.springframework.web.servlet.mvc.support.
                                        ControllerClassNameHandlerMapping">
    <property name="caseSensitive" value="true"/>
    <property name="pathPrefix" value="/datos"/>
</bean>

<bean class="com.javi.es.ejUno.UnoController"/>
<bean class="com.javi.es.ejUno.DosController"/>
```

En este caso, las peticiones que se redirigirán a los controladores serán **"/datos/uno*"** y **"/datos/dos*"**.

3.2.4 Anotación @RequestMapping

Esta forma de redirigir las peticiones es la más versátil y usada. Dentro del controlador anotaremos los métodos que nos interesen, y de ese modo las peticiones del usuario serán respondidas por dichos métodos.

Para hacerlo debemos usar el controlador de mapeo **DefaultAnnotationHandlerMapping**. No se suele indicar directamente, ya que esta configuración está implícita en la etiqueta **<mvc:annotation-driven/>**. Como voy a definir los controladores mediante anotaciones (en vez de definirlos en el fichero de XML con la etiqueta bean) necesito asimismo incluir **<context:component-scan/>**:

```
<mvc:annotation-driven/>
<context:component-scan base-package="com.javi.es.ejUno"/>

<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
      p:prefix="/WEB-INF/jsp/"
      p:suffix=".jsp" />
```

Sólo necesito definir un controlador para atender las diferentes peticiones:

```
@Controller
public class Ejemplo {

    @RequestMapping("/uno.html")
    public String métodoUno() {
        return "uno";
    }

    @RequestMapping("/dos.html")
    public String métodoDos() {
        return "uno";
    }

   @RequestMapping("/index.html")
    public String entrada() {
        return "index";
    }
}
```

En el ejemplo, los métodos anotados devuelven la vista a dibujar. En apartados posteriores veremos varios de los múltiples parámetros y valores de retorno que pueden adoptar estos métodos.

La anotación es muy configurable, admitiendo todo tipo de filtros para seleccionar el método que atenderá a la petición:

- **value** (propiedad por defecto). Lo mostrado en el ejemplo anterior. Admite nombres de página, carpetas, caracteres comodín...

  ```
  @RequestMapping(value="/")
  @RequestMapping(value="/ejemplo/página.html")
  @RequestMapping(value="/index.html")
  @RequestMapping(value="/datos/*/index.html")
  @RequestMapping(value="/pagina*.html")
  ```

  También admite el mapeo de partes del "path" a un parámetro del método anotado:

  ```
  @RequestMapping("/prueba/{valor}/uno.html")
  public String método(@PathVariable String valor) {
      ...
  }
  ```

  Se usa junto a la anotación **@PathVariable** para asignar el valor al parámetro del método que se desee. La anotación permite indicar el nombre del "path" capturado. Incluso admite el uso de expresiones regulares:

  ```
  @RequestMapping("/prueba/{valor:[0-9]+}/uno.html")
  public String método(@PathVariable String valor) {
  ```

```
        ...
    }
```

- **consumes**. Tipo de dato que se acepta:

```
@RequestMapping(value=..., consumes="application/json")
@RequestMapping(value=..., consumes="!text/plain")
```

  También acepta el uso de **produces**, para indicar qué tipo de dato se devolverá, pero es preferible indicarlo de otras formas, como con la anotación @ResponseBody

- **method**. Tipo de petición. Admite GET, POST, DELETE... todos los tipos HTTP:

```
@RequestMapping(value ..., method=RequestMethod.GET)
@RequestMapping(value ..., method=RequestMethod.POST)
@RequestMapping(value ..., method={RequestMethod.GET,
                                   RequestMethod.POST})
```

- **params**. Parámetros que deben estar (o no) en la petición.

```
RequestMapping(params={"nombre","apellido", "edad"})
RequestMapping(params={"nombre","!dni"})
RequestMapping(params="id=34")
```

- **headers**. Igual que el anterior, pero para cabeceras de la petición.

Por último, esta anotación también se puede emplear a nivel de clase. Lo definido de esta forma se sumará a lo indicado en cada uno de los métodos de la clase:

```
Controller
@RequestMapping("/empleado/")
public class Ejemplo {
    ...
    @RequestMapping("ver.html")
    public String ver() {
        ...
    }

    @RequestMapping("borrar.html")
    public String borrar() {
       ...
    }

    @RequestMapping("crear.html")
    public String crear() {
        ...
    }
}
```

En el ejemplo, los métodos atenderán a las peticiones "/empleado/ver.html", "/empleado/borrar.html" y "/empleado/crear.html".

3.3 Argumentos de los métodos anotados

La anotación @RequestMapping no sólo proporciona una gran variedad de formas de redirigir la petición del usuario, sino que además permite definir los métodos anotados con una gran variedad de parámetros. Sólo voy a mostrar las más usadas, o aquellas que considero más útiles. Para una lista exhaustiva, consulta el manual de referencia:

- **java.util.Locale**. El objeto Locale activo en la petición, necesario para formateo de datos y traducciones en el código del controlador.
- **org.springframework.http.HttpMethod**. El tipo de método (GET, POST...) usado en la petición.
- **java.security.Principal**. El usuario autentificado que está realizando la petición. Lógicamente sólo funcionará si usamos la seguridad implementada por el contenedor en vez de escribirla nosotros mismos.

- **java.util.Map**. Uno de los parámetros que más utilizo. Spring proporciona un mapa vacío (depende de cómo esté escrito el resto del controlador) que rellenaremos con las propiedades que nos interesen. Dichas propiedades están disponibles automáticamente en la vista, usando EL:

```
@RequestMapping("/dos.html")
public String métodoEjemplo(Map mapa) {
    mapa.put("mensaje","Método de Ejemplo");
    return "dos";
}
```

  El código de la página JSP:

```
<p>He llegado desde ${mensaje}</p>
```

  El manual de referencia de Spring usa el término **modelo** para referirse al conjunto de objetos que el controlador deja disponibles para la vista.

- **org.springframework.ui.Model**. Como el anterior, pero con un objeto de Spring. Personalmente, prefiero usar un mapa estándar.

- **Comando**. Como los mapas, uno de los más usados. Un **comando** es un objeto de java definido por nosotros mismos. Spring creará automáticamente el objeto (es necesario que tenga un constructor sin parámetros), y si los nombres de los parámetros de la petición coinciden con los nombres de los métodos "setXxx" del mismo, el contenedor los llamará automáticamente.

  A este proceso se le llama **binding**, y es uno de los fundamentos de Spring MVC. Se verá con detalle en los capítulos posteriores, dedicados a validación y conversión de datos.

  Por ejemplo, supongamos que hemos creado la clase Persona:

```
public class Persona {
      privare Integer id;
      private String nombre;
      private String apellidos;
      private Integer edad;

      ...
      public void setNombre(String valor) {
            this.nombre= valor;
      }
      public void setApellidos(String valor) {
            this.apellidos = valor;
      }
      public void setEdad(Integer valor) {
            this.edad = valor;
      }
      public void setId(Integer id) {
            this.id = valor;
      }
}
```

  En la petición del usuario existen los parámetros "nombre", "apellidos" y "edad", y queremos leerlos de forma cómoda y que Spring valide y convierta los datos que nos envíen:

```
@RequestMapping(value="/crear.html",params={"nombre","apellidos","edad"})
public String crearPersona(Persona persona) {
    ...
}
```

  Spring creará un objeto de clase Persona y llamará automáticamente a los métodos setNombre(), setApellidos() y setEdad(), convirtiendo si es necesario (y si puede hacerlo) los textos enviados por el cliente a los objetos Java adecuados. En el ejemplo convertiría el parámetro "edad" a un Integer.

  Todos los comandos pasan a ser parte del modelo de forma automática. El nombre del parámetro en el modelo no es el nombre de la variable, sino **el de la clase** con la primera inicial en minúsculas, aunque se puede cambiar con la anotación **@ModelAttribute** usada a nivel de parámetro.

- **org.springframework.validation.Errors** / **org.springframework.validation.BindingResult**. Cuando se ejecuta un binding se pueden producir errores de conversión o de validación. Este parámetro informa de los errores producidos, junto con varios métodos útiles para realizar el control de los mismos. El método anterior quedaría:

```
@RequestMapping(value="/crear.html",params={"nombre","apellidos","edad"})
public String crearPersona(Persona persona, BindingResult errores) {
    if (errores.hasErrors()) {
        ...
    }
    ...
}
```

  Se deben definir **obligatoriamente a continuación del comando** que se desee inspeccionar; Pueden definirse varios comandos, y el sistema necesita saber a qué comando afecta. BindingResult implementa a la interfaz Errors, por lo que se puede usar cualquiera de las dos definiciones.

  En la práctica siempre que se define un comando también hay que declarar un “Errors” o “BindingResult” junto al mismo. Aunque no usemos validaciones personalizadas Spring siempre realizará un “binding”, por lo que se pueden producir errores de conversión de datos (“type mismatch”). Si no hay definido un parámetro para recoger el error se producirá una excepción.

- **@RequestParam.** Anotación a nivel de parámetro. Indica qué parámetro de la petición se copiará en uno de los argumentos de la función:

```
@RequestMapping(value="/dos.html", params="nombre")
public String métodoEjemplo(@RequestParam("nombre") String nombre) {
    ...
}
```

  Si existe, el argumento "nombre" se copiará en la variable indicada. Es responsabilidad del programador validarla de la forma adecuada.

- **@RequestHeader**. Como la anterior, pero actúa sobre las cabeceras de la petición.

```
@RequestMapping(value="/dos.html")
public String métodoEjemplo(@RequestHeader("accept") String tipo) {
    ...
}
```

- **@RequestBody**. Similar a las anteriores, pero guarda el un parámetro toda la petición. Consulta el manual de referencia "Mapping the request body" para aprender a usar esta anotación
- **HttpSession**. Se pude tener acceso a la sesión directamente, aunque como ya hemos visto en capítulos anteriores Spring dispone de maneras mucho más eficaces para tratar con los objetos de sesión.
- **HttpServeltRequest**. También podemos trabajar con los parámetros de la manera tradicional. Lógicamente, se ha quedado desfasado.

3.4 Tipos de retorno de los métodos anotados

Como en el punto anterior, sólo describiré aquellos más usados o que considero útiles.

- **ModelAndView**. Es un objeto de Spring que guarda la clave a partir de la cual se resolverá la vista y un mapa que será añadido al modelo.

```
@RequestMapping(value="/ver.html")
 public ModelAndView getJefes() {
    ModelAndView mv=new ModelAndView();
    mv.setViewName("paginaJSP");
    mv.addObject("mensaje","mensaje de ejemplo");
    return mv;
 }
```

- **View**. Básicamente, un String con la clave de la vista.
- **String**. Es el que suelo utilizar. Devuelve la clave de la vista que quiero que se dibuje:

```
@RequestMapping("/index.html")
```

```
public String entrada() {
    return "index";
}
```

- **Objeto de java**. El objeto se añadirá automáticamente al modelo. El nombre del objeto será el de su clase en minúsculas.
- **@ResponseBody**. Anotación a nivel de método. Si todo está bien configurado (ver primer capítulo), el objeto devuelto por el método será convertido a un texto en formato JSON y enviado al cliente. Para que funcione la cabecera de petición "accept" debe contener "application/json"; Esto sucede automáticamente si realizamos un petición AJAX con JQuery y la configuramos para que use JSON.

  Un ejemplo:

```
@ResponseBody
@RequestMapping(value="/verTodos.html")
public List<Jefe> getJefes() {
    ...
    return listaJefes;
}
```

Validación, conversión y formateo de datos

Spring proporciona clases, interfaces y anotaciones para conversión y validación. Pueden realizar tres tareas diferenciadas:

- Validación. Comprueban los datos que se **van a escribir** en las **propiedades** de un objeto, generando mensajes de error si es necesario.
- Conversión. Convierte un tipo de datos en otro; casi todas transforman de String al tipo de datos que nos interese.
- Binding. “une, ata” automáticamente los datos escritos por el usuario (por ejemplo los parámetros de la petición que llegan al controlador en MVC) con las propiedades del javabean que nos interese. Binding siempre realiza cierta validación; Si no puede convertir generará errores.

Los mensajes de error suelen ser claves de un fichero de recursos, usando la clase Errors o BindingResult.

La tabla siguiente presenta un resumen de las diferentes técnicas que pueden usarse en Spring para este cometido:

	@Anot.	**Valida**	**Convierte**	**Binding**	**Descripción**
Interfaz Validator		**X**			Validación tradicional
PropertyEditor			**X**	**X**	String → Propiedad
API JSR–303	**X**	**X**			Sólo Java 6
Formatter	**X**	**X**	**X**	**X**	String → Propiedad / Propiedad → String
Converter			**X**		Objeto → Objeto

En los siguientes capítulos veremos en detalle cada uno de los apartados..

4 Interfaz Validator

Tenemos que implementar la interfaz **Validator** pare crear nuestro propio validador. Su uso es muy sencillo. Supongamos que queremos validar el javabean Usuario:

```
public class Usuario implements Serializable {
    private String login;
    private String clave;
    ... (típicos getXXX y setXXX)
}
```

Supongamos que queremos permitir que los valores puedan ser null (no queremos mensajes de error nada más empezar), pero queremos que si hay algo escrito, tenga un máximo de 20 caracteres (evitemos la sobrecarga de pila) y un mínimo de 3 y 8 para login y clave respectivamente. Tenemos que escribir una clase que implemente la interfaz Validator:

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

public class ValidarUsuario implements Validator{

    @Override
    public boolean supports(Class<?> clase) {
        return clase.equals(Usuario.class);
    }

    @Override
    public void validate(Object target, Errors errores) {
        Usuario usuario=(Usuario) target;
        String login=usuario.getLogin();
        String clave=usuario.getClave();
        if (login!=null && (login.length()<3 || login.length()>20))
            errores.rejectValue("login", "error.tam.campo",
                                new Object[]{"login",3,20}, null);

        if (clave!=null && (clave.length()<3 || clave.length()>20))
            errores.rejectValue("clave", "error.tam.campo",
                                new Object[]{"clave",8,20}, null);
    }
}
```

La interfaz nos obliga a definir dos métodos:

- **supports()** indica si una clase está soportada por este validador.
- **validate()** es el método que realiza el trabajo. Devolverá el objeto “Errors” que le haya pasado Spring con los nuevos mensajes que le hayamos añadido. Por supuesto, nosotros decidimos si usamos claves del fichero de recursos, si tiene parámetros, si es un texto estático, etc.

4.1 Creación de errores

Para crear los errores usamos la interface **Errors**. Representa una colección de errores a la que podemos añadir elementos desde el validador e incluso desde el controlador.

Es importante recordar que no sólo los validadores añaden errores a esta lista. Los editores de propiedades y en general cualquier clase que convierta de String a “propiedad del javabean” puede generar un error de conversión de tipos.

La interfaz proporciona varios métodos útiles (ver la documentación de la API para una lista exhaustiva):

- List<ObjectError> getAllErrors(). Proporciona una lista con todos los errores registrados hasta el momento.
- boolean hasErrors(). Indica si hay errores.
- void rejectValue(String field, String errorCode, Object[] errorArgs, String defaultMessage). Este método tiene varias firmas: Sin argumentos, sin mensaje por defecto... En este caso, hay que indicarle el nombre de la propiedad que genera el error (importante para presentarlo después en la página JSP), la clave del error en el fichero de recursos, qué parámetros queremos para ese mensaje y un mensaje por defecto que puede usarse si no existe la clave.

Asimismo, Spring proporciona la clase **ValidationUtils** con un par de métodos estáticos para ayudarnos a generar los errores más habituales:

- static void rejectIfEmpty(Errors errors, String field, String errorCode, Object[] errorArgs, String defaultMessage). Como en el caso anterior, el método admite varias firmas. Como es un método estático pide la lista de errores a modificar. Genera un error si la propiedad del objeto en cuestión (Spring lo comprueba posteriori) es null o contiene “”.
- RejectIfEmptyOrWhitespace(...) es como el anterior, pero también genera el error si sólo contiene espacios en blanco.

4.2 Registrar el validador

Aunque se puede utilizar de manera global (ver <mvc:annotation-driven validator=”xxx”/>), se suele registrar únicamente en los controladores que van a usar el javabean como un comando. Siguiendo el ejemplo:

```
@Controller
public class ControladorEntrada {

    @InitBinder
    public void prepararControlador(WebDataBinder bind) {
        bind.setValidator(new ValidarUsuario());
        ...
    }

    @RequestMapping("/entrada.html")
    public String entrada(@Valid Usuario usuario, BindingResult resul) {
        if (resul.hasErrors())...
        ...
    }
   ...
}
```

El método anotado con **@InitBinder** se lanzará antes de que el controlador sea usado por vez primera. Usa el parámetro **WebDataBinder** para configurar los conversores, los editores de propiedades y los validadores que necesita el controlador.

Cuando a uno de los métodos anotados con @RequestMapping le llegue una petición, si ese método tiene un comando asociado, Spring comprobará si existe algún validador registrado para ese comando (método supports() del validador). Si es así se lanzará el método validate(), que a su vez rellenará (o no) la lista de posibles errores.

En los métodos del controlador que atienden a peticiones se suele definir a continuación del comando un objeto de clase Errors o BindingResult (extiende a Errors). De este modo el controlador sabe qué errores se han producido.

Desde la versión 4.x de Spring es necesario decorar el comando con la anotación **@Valid** o **@Validated**, o de lo contrario no se aplicará ninguna validación personalizada: únicamente detectaríamos los errores de conversión de datos producidos al realizar el binding. Estas anotaciones están explicadas en el capítulo 8, “Validación estándar de Java: JSR-303”.

5 Editores de Propiedades

Los editores de propiedades (PorpertyEditors) son el mecanismo que tradicionalmente ha usado Spring para convertir un texto a un objeto de Java. No sólo se usan para la conversión de la entrada del usuario; También se usan por ejemplo en la conversión de los “property” de los ficheros de configuración de Spring.

Por defecto el marco de trabajo viene con editores de propiedades ya configurados para las conversiones típicas: de String a número, a fecha, colecciones, mapas, enumeraciones... hasta objetos de clase Color. Pero de vez en cuando tenemos que definir uno propio.

NOTA: Si se va a usar para convertir y enlazar datos del usuario, es recomendable usar Formatter en vez de PropertyEditor, ya que seguramente querremos formatear **también** la salida. Además permite hacerlo mediante anotaciones.

Supongamos que tenemos un bean “Persona” con las siguientes propiedades:

```
public class Persona implements Serializable {

    private String nombre;
    private String apellidoUno;
    private String apellidoDos;
    private Integer edad;
    private DNI dni;

    ... (típicos getXxx y SetXxx)

    @Override
    public String toString() {
        if (numero==null) return "";
        else return numero + " " + letra;
    }
}
```

Los datos se van a leer desde un formulario de HTML. Para casi todas las propiedades no habrá ningún problema: Hay editores de propiedades para todos los datos comunes; Pero no hay ninguno definido para la clase DNI:

```
public class DNI implements Serializable{

    private String numero;
    private char letra;

    ... (típicos getXxx y SetXxx)
}
```

Para crear nuestro propio editor de propiedades podemos implementar la interfaz **PropertyEditor**. Pero para casos sencillos (casi todos) es más sencillo extender la clase **PropertyEditorSupport**:

```
import java.beans.PropertyEditorSupport;

public class DNIEditor extends PropertyEditorSupport {

    @Override
    public void setAsText(String texto) {
        if (texto==null) {
            super.setValue(null);
        }
        else {
            texto=texto.trim().toUpperCase();
            if (!texto.matches("^[0-9]{8}[\\s-]*[A-Z]$"))
```

```
                throw new IllegalArgumentException("DNI incorrecto.");
            DNI dni=new DNI();
            dni.setNumero(texto.substring(0, 8));
            dni.setLetra(texto.charAt(texto.length()-1));
            //no compruebo si el numero y la letra se corresponden.
            super.setValue(dni);
        }
    }
}
```

Sólo necesitamos sobreescribir el método **setAsText()**, al que Spring le proporciona a texto a convertir. A través del método **setValue()**, tiene que proporcionar el objeto destino. Si por algún motivo el editor falla, éste debe lanzar un **IllegalArgumentException**. La clave del fichero de recursos será como es habitual “typeMismatch” o “typeMismatch.claseBean.propiedad”.

5.1 Registrar el editor de propiedades

Se puede registrar de manera local en un controlador concreto o bien de manera global para toda la aplicación. Esta es una opción aconsejable para un PorpertyEditor.

Registrarlo en el controlador es similar a registrar un validador:

```
@InitBinder
    public void prepararControlador(WebDataBinder bind) {
        bind.registerCustomEditor(DNI.class, new DNIEditor());
        ...
    }
```

El método anotado con **@InitBinder** se lanzará antes del primer uso del controlador. Al método de **WebDataBinder registerCustomEditor()** hay que pasarle el class del objeto a editar y una instancia de la clase que realizará el trabajo.

Si se quiere registrar de manera global hay que hacerlo a través de los ficheros de configuración de XML. Se puede usar la clase **CustomEditorConfigurer** para ello:

```
<bean
class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
        <map>
            <entry key=" ejemplo.DNI" value="ejemplo.DNIEditor"/>
        </map>
    </property>
</bean>
```

Se pueden registrar tantos editores como se desee. A partir de ahora se usará nuestra clase “DNIEditor” para convertir de String a “DNI”, incluidos los ficheros de configuración.

Si se está realizando la conversión de String a DNI desde un campo de formulario de Spring, el texto se “formateará” en la salida usando el método “toString()” de DNI.

6 Conversión de Clases

Hay varias interfaces que podemos implementar para escribir nuestro propio conversor. La más sencilla es la interfaz **Converter**:

```
package org.springframework.core.convert.converter;

public interface Converter<S, T> {
     T convert(S source);
}
```

Como se ve en la definición, sólo tenemos que indicar los tipos “Source” y “Target” que queremos que intervengan en la conversión. Supongamos que tenemos las siguientes clases, y queremos convertir de una a otra:

```
package ejemplo;

public class DatosEntrada {
    private String codigo;
    private String nombre;
    private String apellidos;
    private String edad;
    ... (típicos métodos getXxx y SetXxx)
    ...
}


package ejemplo;

public class Trabajador {
    private int codigo;
    private String nombre;
    private String apellidoUno;
    private String apellidoDos;
    private int edad;
    ... (típicos métodos getXxx y SetXxx)
    ...
}
```

Para crear el conversor que cree un objeto de clase Trabajador a partir de otro de tipo DatosEntrada sólo tengo que implementar la interfaz Converter:

```
package ejemplo;

import org.springframework.core.convert.converter.Converter;

public class ConvertirEntradaTrabajador implements Converter<DatosEntrada,
Trabajador> {

    @Override
    public Trabajador convert(DatosEntrada origen) {
        try {
            Trabajador p = new Trabajador();
            p.setCodigo(Integer.parseInt(origen.getCodigo()));
            p.setNombre(origen.getNombre());
            if (origen.getApellidos() == null ||
                origen.getApellidos().trim().equals("")) {
                p.setApellidoUno(null);
                p.setApellidoDos(null);
            } else {
```

```
                //Sólo es un ejemplo...
                String apellidos=origen.getApellidos().replace(" +", " ");
                int posición = apellidos.indexOf(' ');
                if (posición == -1) {
                    p.setApellidoUno(apellidos);
                    p.setApellidoDos(null);
                } else {
                    p.setApellidoUno(apellidos.substring(0, posición));
                    p.setApellidoDos(apellidos.substring(posición+1));
                }
            }
            p.setEdad(Integer.parseInt(origen.getEdad()));
            return p;
        } catch (NumberFormatException ex) {
            throw new IllegalArgumentException(ex.getLocalizedMessage());
        }
    }
}
```

En el ejemplo me limito a convertir de String a Int, y divido “apellidos” en “apellidoUno” y “apellidoDos”, si puedo.

6.1 Registrar conversores

Por último sólo queda registrar el conversor. Generalmente querremos hacerlo de forma global a toda la aplicación, en los ficheros de configuración de Spring:

```
<mvc:annotation-driven conversion-service="conversionService"/>

<bean id="conversionService"

class="org.springframework.format.support.FormattingConversionServiceFac
toryBean">
    <property name="converters">
        <list>
            <bean class="ejemplo.ConvertirEntradaTrabajador"/>
        </list>
    </property>
</bean>
```

La anotación <mvc:annotation-driven/> incorpora un servicio de conversión para las transformaciones habituales. Si se desea modificar o ampliar las conversiones posibles, se usa el atributo “**conversion-service**”.

En el código de ejemplo anterior hemos reemplazado el servicio de conversión por un bean de la clase **FormattingConversionServiceFactoryBean**... que es el servicio configurado por defecto en <mvc:annotation-driven/>. Pero hemos ampliado la lista de conversores por defecto con nuestra clase de ejemplo.

Se puede registrar un servicio de conversión personalizado en un controlador usando la anotación **@InitBinder**:

```
@InitBinder
public void prepararControlador(WebDataBinder bind) {
    bind.setConversionService(otro_servicio_de_conversión);
}
```

6.2 Usar el conversor

A diferencia del resto de utilidades que hemos visto en este capítulo, la forma más habitual de usar los servicios de conversión es programáticamente: Las entradas de datos del usuario son siempre textos, y hay herramientas más directas para convertir de String a Objeto que un conversor.

El uso ideal de conversores es cuando en nuestro programa necesitamos traducir los datos entre dos beans con múltiples propiedades, como se ha mostrado en los ejemplos de código vistos anteriormente. Siguiendo el ejemplo, supongamos que hemos registrado el conversor de manera global. Para usarlo en cualquiera de nuestras clases, sólo tenemos que usar inyección de dependencia:

```
@Autowired
private ConversionService conversor;
...
public Trabajador métodoEjemplo(DatosEntrada entrada) {
    return conversor.convert(entrada, Trabajador.class);
}
...
```

6.3 Otras interfaces

Existe una versión de Converter llamada **ConverterFactory**, diseñada para convertir un “árbol” de objetos a cierto tipo (por ejemplo, convertir de cualquier tipo de enumeración a cierta clase). Ver el manual de referencia para más información.

Tanto Converter como ConverterFactory están fuertemente tipadas, lo cual puede ser un inconveniente en ciertos casos. Si se necesita una forma de conversión más flexible es recomendable usar la interfaz **GenrericConverter**:

```
package org.springframework.core.convert.converter;

public interface GenericConverter {
    public Set<ConvertiblePair> getConvertibleTypes();
    Object convert(Object origen, TypeDescriptor tipoOrigen,
                                  TypeDescriptor tipoDestino);
}
```

El método **getConvertibleTypes()** informa sobre qué conversiones es capaz de realizar. Hay que devolver un conjunto de objetos de la clase **ConvertiblePair**; Es una clase muy sencilla que pide de qué clase a qué clase se puede convertir. Su constructor:

```
Public ConvertiblePair(Class<?> sourceType, Class<?> targetType)
```

El método **convert()** es el que realiza la conversión en sí. Devuelve el objeto destino. Sus parámetros son el objeto origen y los tipos a convertir. **TypeDescriptor** es una clase usada para representar tipos de datos, una especie de Class pero con unos cuantos métodos para hacer comprobaciones y conversiones.

La interfaz **ConditionalGenericConverter** extiende a la anterior. Permite cancelar la conversión en función del valor del método **matches()**:

```
public interface ConditionalGenericConverter extends GenericConverter {
     boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType);
}
```

7 Formateo de campos en Spring

Generalmente no sólo necesitamos convertir y validar los textos de entrada del usuario, sino que querremos modificar la salida de los mismos para que su apariencia se adecue a lo que necesitamos. Los PropertyEditor tradicionales de Spring sólo convierten de String (entrada del usuario) a Objeto (nuestras propiedades). Formatter realiza también el camino inverso: De String a Objeto y de Objeto a String.

Una vez configurado el formateador, para que se aplique la conversión de los textos enviados por el usuario al objeto que nos interese sólo tenemos que usar comandos en los métodos anotados del controlador. Pero para que Spring convierta nuestras propiedades a textos cuando generamos el código de HTML tenemos que usar las etiquetas JSTL de Spring: las típicas etiquetas de formularios o **<spring:eval/>** si sólo estamos generando textos:

```
<s:eval expression="e.NIF"/>
```

7.1 Interfaz Formatter

Para crear nuestro propio formateador tenemos que implementar la interfaz **Formatter<T>**. Supongamos el siguiente JavaBean:

```
public class DNI implements Serializable{
    private String numero;
    private String letra;
    ...
    ...
}
```

Una clase para aplicar formato a los campos de este tipo:

```
import java.text.ParseException;
import java.util.Locale;
import org.springframework.format.Formatter;

public class FormatearDNI implements Formatter<DNI>{

    @Override
    public String print(DNI dni, Locale locale) {
        return dni.getNumero() + "-" + dni.getLetra();
    }

    @Override
    public DNI parse(String texto, Locale locale) throws ParseException {
        if (texto==null) throw new ParseException("El DNI no puede ser
nulo", 0);
        if (!texto.matches("[0-9]{8}[-\\s]?[A-Z]"))
            throw new ParseException("Formato incorrecto", 0);
        DNI dni=new DNI();
        dni.setNumero(texto.substring(0, 8));
        dni.setLetra(texto.charAt(texto.length()-1) + "");
        return dni;
    }

}
```

Obliga a definir dos métodos, **print()**, que convertirá nuestro objeto a String y **parse()**, que convertira de String a un objeto de la clase que nos interese. En caso de que no se pueda convertir se lanzará la excepción **ParseException**. Por supuesto, tenemos acceso al Locale que se esté usando en ese momento.

7.2 Registrar el formateador

Una vez que lo tenemos todo definido, hay que indicarle a Spring cómo aplicarlo. Para eso necesitamos reeemplazar la típica clase postprocesadora de Spring. En este caso, **FormattingConversionServiceFactoryBean:**

```
import org.javi.es.vista.conanotaciones.DNIFactoriaFormateadorAnotado;
import org.springframework.format.FormatterRegistry;
import
org.springframework.format.support.FormattingConversionServiceFactoryBean;

public class RegistrarFormateadores extends
FormattingConversionServiceFactoryBean {

  @Override
  public void installFormatters(FormatterRegistry registro) {
    super.installFormatters(registro);
    registro.addFormatterForFieldType(DNI.class,new FormatearDNI());
    //Se pueden registrar tantos como se desee
  }
}
```

Esta clase permite registrar no sólo formateadores de este tipo; También se puede usar formateadores por anotaciones (ver más adelante) y conversores (de objeto a objeto). Por último, definimos el bean en los ficheros de configuración de Spring:

```
<mvc:annotation-driven conversion-service="servicioConversor"/>
<bean id="servicioConversor"
class="org.javi.es.vista.RegistrarFormateadores"/>
```

Cuando usemos “comandos” con propiedades de clase DNI, Spring **automáticamente** usará este formateador para convertir la petición del cliente a DNI y general el “value” correspondiente del formulario de HTML. Si se producen errores de validación, usará las claves del fichero de recursos tal como se describe en la validación mediante anotaciones; “typeMismatch” o bien “typeMismatch.nombreJavaBean.nombrePropiedad”.

7.3 Formatear mediante anotaciones

Es muy similar a lo explicado anteriormente, pero tiene una ventaja. Sólo formatearemos los campos marcados con la anotación. Esto nos permite por ejemplo definir un Formatter<String>, sin necesidad de definir una clase DNI para nuestros campos: Hará mucho más sencilla la persistencia de nuestras clases...

En primer lugar definimos la anotación que usaremos:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ElementType.METHOD, ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface DNIAnotado {

}
```

La anotación no contiene nada, ni está marcada de ninguna manera especial. Es sólo una bandera que asociaremos posteriormente con el formateador:

```
import java.text.ParseException;
import java.util.Locale;
import org.springframework.format.Formatter;

public class FormatearDNIAnotado implements Formatter<String>{

    @Override
```

```
    public String print(String dni, Locale locale) {
        return dni;
    }

    @Override
    public String parse(String texto, Locale locale) throws ParseException
{
        if (texto==null) throw new ParseException("El DNI no puede ser
nulo", 0);
        if (!texto.matches("[0-9]{8}[-\\s]?[A-Z]"))
                throw new ParseException("Formato incorrecto", 0);
        return  texto.substring(0, 8)+ "-" + texto.charAt(texto.length()-
1);
    }
```

Es similar al formateador que definimos anteriormente, aunque éste sólo usa objetos de clase String. ¿Cómo sabe Spring cuándo aplicar esta conversión “de String a String”? Tenemos que enganchar de algún modo la anotación con el formateador. Para eso usaremos la interfaz **AnnotationFormatterFactory<A extends Annotation>**:

```
import java.util.HashSet;
import java.util.Set;
import org.springframework.format.AnnotationFormatterFactory;
import org.springframework.format.Parser;
import org.springframework.format.Printer;

public class DNIFactoriaFormateadorAnotado implements
AnnotationFormatterFactory<DNIAnotado>{

   @Override
   public Set<Class<?>> getFieldTypes() {
       Set<Class<?>> set = new HashSet<Class<?>>();
       set.add(String.class);
       return set;
   }

   @Override
   public Printer<String> getPrinter(DNIAnotado annotation, Class<?>
fieldType) {
       return new FormatearDNIAnotado();
   }

   @Override
   public Parser<String> getParser(DNIAnotado annotation, Class<?>
fieldType) {
       return new FormatearDNIAnotado();
   }
}
```

Nos pide la Anotación que vamos a usar. En sus métodos **getPrinter()** y **getParser()** es donde indicamos qué clase (qué objeto) se va a encargar de realizar la conversión. Por supuesto, tiene que se una clase que implemente Formatter. **GetFieldTypes()** indica qué clases es capaz de convertir. En este caso, String.

Para usar la nueva anotación:

```
public class Persona {
    private String nombre;
    ...
    @DNIAnotado
    private String Dni;
    ...
}
```

Si persona se usa como un comando en Spring, el “Dni” se formateará automáticamente.

7.4 Registrar el formateador

Como en el caso anterior, tenemos que indicar a Spring que tenemos un nuevo formateador. Se realiza con la misma clase del ejemplo inicial, pero internamente usaremos otro método:

```
import org.javi.es.vista.conanotaciones.DNIFactoriaFormateadorAnotado;
import org.springframework.format.FormatterRegistry;
import
org.springframework.format.support.FormattingConversionServiceFactoryBean;

public class RegistrarFormateadores extends
FormattingConversionServiceFactoryBean {

  @Override
  public void installFormatters(FormatterRegistry registro) {
    super.installFormatters(registro);
    registro.addFormatterForFieldAnnotation(new
DNIFactoriaFormateadorAnotado());
  }
}
```

Sólo queda incluirlo en los ficheros de XML de Spring:

```
<mvc:annotation-driven conversion-service="servicioConversor"/>
<bean id="servicioConversor"
class="org.javi.es.vista.RegistrarFormateadores"/>
```

7.5 Anotaciones predefinidas

Hay dos anotaciones ya predefinidas para dar formato a fechas y a números.

- **@DateTimeFormat**. Formateo de fechas. Admite atributos como “iso” (DATE, DATETIME, TIME) o “pattern”, para que la fecha aparezca exactamente como se desea. Es necesario incorporar la biblioteca JODA (Java date and time API).
- **@NumberFormat**. Formateo de números. Admite “pattern” o “style” (CURRENCY, NUMBER, PERCENT).

7.6 Más ejemplos

Un caso muy típico es la conversión de fechas. Aunque @DateTimeFormat es cómoda, no permite mucha flexibilidad: Escribir espacios antes o después de la fecha, admitir a la vez diferentes caracteres separadores, etc. Una posible solución (sólo es un ejemplo, se puede mejorar mucho):

```
public class FechaFormatter implements Formatter<Date>{
private String año;

    public FechaFormatter(String formato) {
        if (formato.equals("LARGO")) año="yyyy";
        else if (formato.equals("CORTO")) año="yy";
        else throw new
IllegalArgumentException(this.getClass().getCanonicalName()
                                                + ": Formato no
soportado.");
    }
    private SimpleDateFormat getSDF(String idioma) {
        SimpleDateFormat sdf;
        if (idioma.equals("en")) sdf=new SimpleDateFormat("MM/dd/" +
año);
        else if (idioma.equals("eu")) sdf=new SimpleDateFormat(año +
"/MM/dd");
        else sdf=new SimpleDateFormat("dd/MM/" + año);
        sdf.setLenient(false);
        return sdf;
```

```
        }
        @Override
        public String print(Date object, Locale locale) {
            SimpleDateFormat sdf=this.getSDF(locale.getLanguage());
            return sdf.format(object.getTime());
        }
        @Override
        public Date parse(String texto, Locale locale) throws ParseException
    {
            SimpleDateFormat sdf=this.getSDF(locale.getLanguage());
            texto=texto.trim().replaceAll("[\\s\\./-]+", "/");
            return sdf.parse(texto);
        }
    }
```

La anotación que lanzará este formateador:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.PARAMETER})
public @interface FormatoFecha {
    public String value() default "LARGO";
}
```

La factoría que proporciona el servicio y asocia la anotación con la clase de Java:

```
public class FactoriaFormatoFecha
            implements AnnotationFormatterFactory<FormatoFecha> {
    @Override
    public Set<Class<?>> getFieldTypes() {
        Set conjunto=new HashSet<Class>();
        conjunto.add(Date.class);
        return conjunto;
    }
    @Override
    public Printer<Date> getPrinter(FormatoFecha anotación, Class<?> tipo)
{
        return new FechaFormatter(anotación.value());
    }
    @Override
    public Parser<Date> getParser(FormatoFecha anotación, Class<?> tipo) {
        return new FechaFormatter(anotación.value());
    }
}
```

Y por último, la clase necesaria para registrar la anotación (y no perder las que ya vienen definidas):

```
public class RegistrarFormateadores
            extends FormattingConversionServiceFactoryBean {
    @Override
    public void installFormatters(FormatterRegistry registro) {
        super.installFormatters(registro);
        registro.addFormatterForFieldAnnotation(new
FactoriaFormatoFecha());
    }
}
```

Sólo queda usar esta clase en el fichero de configuración de Spring:

```
<mvc:annotation-driven conversion-service="servicioConversor"/>
<bean id="servicioConversor"
      class="org.javi.ejemplo.vista.formatos.RegistrarFormateadores"/>
```

8 Validación estándar de Java: JSR-303

La especificación JSR-303 (sólo para Java 6) explica cómo se pueden definir validaciones usando anotaciones, así como una serie de anotaciones ya creadas para las validaciones más habituales.

Para que las validaciones se apliquen es necesario implementar la interfaz javax.validation.Validator, importar clases que lo hagan o usar un marco de trabajo como Spring que incorpore dichas clases.

8.1 Definir restricciones propias

Hay que definir una anotación y una clase que implemente la validación que necesitamos. Un ejemplo de anotación:

```
package ejemplo;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import javax.validation.Constraint;

@Constraint(validatedBy=ValidarMayusculas.class)
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Mayusculas {
    String message() default "mensaje por defecto";
    Class<? extends Payload>[] payload() default {};
    Class<?>[] groups() default {};
    //atributos propios: minimo, maximo, etc.
}
```

8.1.1 message

La norma recomienda usar “resource bundle keys” (claves del fichero de recursos) para traducir los mensajes de error. Indica que “message” debería ser parte de clave.

Spring, por defecto resuelve el mensaje de error usando estas claves (en este orden. Supongo que se ha usado la restricción “Size”):

1. El nombre sin cualificar de la anotación: “Size”

2. El nombre de la anotación seguido del bean en el que se ha usado y la propiedad marcada: “Size.empelado.apellidoUno”.

3. El valor de message. Spring usa este valor como el mensaje que aparecerá si no se indica nada en el fichero de recursos (nada impide interpretarlo como una clave con la etiqueta adecuada). También lo usa como un parámetro de las claves que se hayan definido.

4. “typeMismatch” es el “nombre de anotación” usado cuando no se puede convertir el tipo de datos.

Si se quiere personalizar la resolución de claves, como siempre no hay más que definir un bean que extienda a la clase usada por defecto: org.Springframework.context.support.DefaultMessageSourceResolvable.

En los parámetros del texto asociado a la clave aparecerán todos los valores usados en la anotación: Nombre del campo, del bean, de la clase, mínimos y máximos, message, etc. El orden varía de una anotación a otra.

8.1.2 payload

Información adicional para validar el campo, por ejemplo, “quiero una validación más estricta”, o “no distinguir entre mayúsculas y minúsculas”. Para hacerlo más genérico no se pide un array de String, sino clases que implementen la interfaz Payload (es un truco muy típico, una especie de enumeración). Un ejemplo de uso:

```
package ejemplo;

public class Prohibir {
    public static class Nada implements Payload {};
    public static class Acentos implements Payload {};
    public static class CaracteresEspeciales implements Payload {};
}

public class Trabajador {
    @Mayusculas (payload= Prohibir. Acentos.class)
    private String nombre;
    ...
    @Mayusculas (payload= Prohibir. CaracteresEspeciales.class)
    private String NSS
}
...
```

8.1.3 groups

Dentro de un bean se marcarán para validar muchas de sus propiedades propiedades. A veces interesa validar sólo alguna de ellas. Por defecto, todas las marcas de validación pertenecen al grupo “Default”. Éste es el grupo que se valida si no se indica nada; Pero se puede cambiar con el atributo “groups”:

```
    public interface Obligatorio {}

    public class Empleado {

          @Size(min=3,max=50,groups={Default.class,Obligatorio.class})
          private String nombre;

          @Size(min=3,max=50, groups={Default.class,Obligatorio.class})
          private String apellidoUno;

          @Size(min=3,max=50)
          private String apellidoDos;
          ...
    }
```

Cuando se lanza el validador sobre una clase de java, la instrucción es algo así como:

```
Validador.validate(empleado);
```

Eso ejecutará la validación sobre los campos marcados con anotaciones “Default” (los tres en este caso). Pero también se puede hacer de otro modo:

```
Validador.validate(empleado,Obligatorio.class);
```

Sólo se usarán las marcas Obligatorias.

8.1.4 Constraint

Aparte de marcar la anotación como una futura validación, indica qué clase va a efectuar el trabajo:

```
@Constraint(validatedBy=ValidarMayusculas.class)
```

8.2 Definir la clase que validará

Como es habitual, basta con definir una clase que implemente la interfaz ConstraintValidator:

```
public interface ConstraintValidator<A extends Annotation, T> {
     void initialize(A constraintAnnotation);
     boolean isValid(T value, ConstraintValidatorContext context);
}
```

El método *initialize*() sirve para iniciar la validación. Se le pasa por parámetro la anotación (con sus valores) que ha disparado el proceso.

El método *isValid*() es quien realiza la validación en sí. Recibe dos argumentos; "value" es el valor a comprobar y "context" el contexto de validación. Es un objeto que implementa *ConstraintValidatorContext,* que a su vez proporciona métodos para generar mensajes de error adicionales. Un ejemplo sencillo:

```
package ejemplo;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class ValidarLetrasMayusculas implements
                                ConstraintValidator<LetrasMayusculas,
String> {

    @Override
    public void initialize(LetrasMayusculas anotación) {
        //El ejemplo es sencillo. No hay nada que iniciar
    }

    @Override
    public boolean isValid(String valor, ConstraintValidatorContext
context) {
        //Ya hay validadores para comprobar si es null...
        if (valor==null || valor.equals("")) return true;
        return valor.matches("^[A-ZÑÁÉÍÓÚÜ]+$");
    }
}
```

8.3 Usar la anotación

Validar mediante anotaciones es muy simple: basta con marcar las propiedades que nos interesen de cualquier JavaBean y asegurarnos que en el fichero de recursos están las claves correspondientes:

```
public class Persona {
    @LetrasMayusculas
    private String nombre;
    @LetrasMayusculas
    private String apellidoUno;
    ...
}
```

En el fichero de recursos de la aplicación (depende del marco de trabajo cómo definirlo):

```
#Esta clave se aplica a todos...
LetrasMayusculas=El texto de {0} debe estar en mayúsculas.
#... a no ser que se defina una más concreta:
LetrasMayusculas.persona.apellidoUno=Los apellidos deben estar en
mayúsculas.
```

Las anotaciones se aplicarán de forma automática si usamos JPA al persistir o modificar cualquier entidad, aunque por supuesto se puede configurar el ORM para desactivar esta característica. Por ejemplo en Hibernate:

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
```

```
        <property name="persistenceXmlLocation"
                  value="classpath:/com/javi/fws/modelo/persistence.xml"/>
        <property name="persistenceUnitName" value="FabricaWebSpringPU"/>
        <property name="validationMode" value="NONE"/>
    </bean>
```

Si están decorando un comando, es necesario usar las anotaciones @Valid o @Validate para que se activen (están explicadas en los apartados siguientes):

```
    @ResponseBody
    @RequestMapping("/crear.html")
    public Respuesta crear(@Valid Cliente c, BindingResult errores) {
    ...
```

Las anotaciones pueden también se usadas mediante ficheros de configuración en XML. Ver el manual de referencia de la norma JSR-303 (143 páginas) para más información.

8.4 Validaciones predefinidas

Hay una serie de validaciones simples definidas por defecto en el paquete “javax.validation.constraints”.

- @**Null**. La propiedad validada debe ser null.
- @**NotNull**. La propiedad validado no debe ser null.
- @**AssertTrue**. La propiedad debe ser true.
- @**AssertFalse**. Lo contrario.
- @**Min(valor)**. Permite indicar un valor mínimo.
- @**DecimalMin(valor)**. Como el anterior, pero admite String.
- @**Max(valor)**. Valor máximo que puede alcanzar la propiedad.
- @**DecimalMax(valor)**. Como el anterior, pero admite String.
- @**Size**. Tamaño de un texto. Se puede indicar el tamaño **min** y **max** del mismo.
- @**Digits**. Dígitos enteros y decimales que admite un número. Se indica con las propiedades **fraction** e **integer**.
- @**Past**. Indica que una fecha debe ser anterior a la del sistema.
- @**Future**. Fecha posterior a la del sistema.
- @**Pattern**. El texto debe coincidir con una expresión regular. Se indica mediante el atributo **regexp**.
- @**Valid**. Se suele usar para anotar listas u objetos de Java con propiedades anotadas. Hace que las validaciones se ejecuten en cascada a través de los elementos de la colección o de las propiedades de la clase. Por supuesto, hace falta un entorno de ejecución que entienda dicha anotación.

También existe la posibilidad de anotar varias veces una propiedad con el mismo tipo de anotación. Para eso se han definido anotaciones que permiten arrays de las anotaciones anteriores: El nombre es el del tipo de anotación que pueden contener seguido de “.List”: **Min.List**, **Max.List**, **Size.List**, etc.

8.5 Validaciones predefinidas en Hibernate

Spring (y lógicamente Hibernate) poseen validaciones adicionales. Hay que tener en cuenta que esto no es java estándar, por lo que si se usan nuestro código dependerá lógicamente del paquete "org.hibernate.validator.constraints".

- **@CreditCardNumber**. Número de tarjeta de crédito
- **@Email**. Correo electrónico.
- **@Length**. El texto debe estar incluido entre el valor de los parámetros **min** y **max**.
- **@NotBlank**. El texto no puede ser una cadena vacía.
- **@NotEmpty**. El texto no puede ser nulo ni cadena vacía.
- **@Range**. Número incluido entre **min** y **max**.
- **@ScriptAssert**. Evalua una expresión de "script" sobre el elemento especificado. Hay que incorporar soporte para lenguajes script dentro de Java...
- **@URL**. Aspecto de URL

8.6 Otros recursos

El paquete **org.apache.commons.validator** de apache no tiene que ver con anotaciones, pero resulta muy fácil de encapsular en las mismas: Posee muchos métodos para validar de manera "tradicional" cosas como DNS, IP, ISBN, formatos numéricos, etc. Por supuesto, también se puede usar de manera tradicional en nuestro código de Java y mediante ficheros de XML...

8.7 Problema con Spring MVC 3.0.2 y @Valid

La forma habitual de lanzar la validación desde un controlador de Spring es anotar el comando con **@Valid**. Eso hará que se validen todos los campos anotados del javabean:

```
@Controller
@RequestMapping("/empleado/")
public class ControladorEmpleado {

...

RequestMapping(value="modificar.html")
public String procesarModificar(@Valid Empleado empleado,
                                                BindingResult resul) {
    if (resul.hasErrors())...
    ...
}

...

}
```

Por desgracia, la anotación no permite especificar sobre qué **grupos de anotaciones** queremos realizar la validación. Si no queremos validar ciertos campos anotados, no podremos evitarlo. La solución es sencilla. Simplemente, tenemos que lanzar la validación manualmente. A continuación se muestra una clase con un método estático que realiza dicha tarea:

```
package ejemplos;

import java.util.*;
import javax.validation.*;
```

```
import org.springframework.validation.Errors;

/** Idea copiada y adaptada de http://digitaljoel.nerd-
herders.com/2010/12/28/spring-mvc-and-jsr-303-validation-groups/ . Spring 3.0.2
no admite grupos de validación en la anotación @Valid. Hasta que no se modifique
(¿en 3.1?), si se desea validar por anotaciones y grupos, no queda más remedio
que lanzar la validación a mano. */

public class ValidacionManual {
    /** Método estático para validad por grupos.
     * @param result La lista que contiene los errores que se han encontrado.
     * @param objeto El objeto a validar. Se supone que usa anotaciones JSR-303.
     * @param classes Las interfaces que marcan los grupos de validación.
     * @return Devuelve true si no se han encontrado errores.
     */

    public static boolean isValid(Errors result, Object objeto, Class<?>...
classes ) {
        if ( classes == null || classes.length == 0 || classes[0] == null )
            classes = new Class<?>[] { Default.class };

        Validator validator=Validation.buildDefaultValidatorFactory()
                                                              .getValidator();
    Set<ConstraintViolation<Object>>violations=
                                          validator.validate(objeto,classes);
        for ( ConstraintViolation<Object> v : violations ) {
            Path path = v.getPropertyPath();
            String propertyName = "";
            if ( path != null ) {
                for ( Node n : path )
                    propertyName += n.getName() + ".";
                propertyName=propertyName.substring(0,propertyName.length()-1);
            }
            String constraintName=v.getConstraintDescriptor().
                              getAnnotation().annotationType().getSimpleName();
            Object lista[]=argumentosValidador(propertyName,
                                v.getConstraintDescriptor().getAttributes());
            if ( propertyName == null || "".equals(  propertyName  ))
                result.reject( constraintName, lista, v.getMessage());
            else result.rejectValue(propertyName,
                                          constraintName,lista,v.getMessage());
        }
        return violations.isEmpty();
    }

    private static Object[] argumentosValidador(String propiedad,
                                                 Map <String,Object> mapa) {
        List lista=new ArrayList();
        if (propiedad!=null && !propiedad.equals("")) lista.add(propiedad);
        for (String clave:mapa.keySet())
            if (!clave.equals("payloads") && !clave.equals("groups"))
                lista.add(mapa.get(clave));
        if (lista.isEmpty()) return null;
        else return lista.toArray();
    }
}
```

Un ejemplo de uso para el caso anterior:

```
  RequestMapping(value="modificar.html")
  public String procesarModificar(Empleado empleado, BindingResult resul) {
      if (!ValidacionManual.isValid(resul,empleado,ValidarSalario.class)...
      ...
  }
```

Hay que tener en cuenta que aunque la validación automática no se realice, Spring siempre unirá los datos de la petición del cliente con el comando, en este caso con “empleado”. Si por algún motivo no puede realizar dicha unión, por ejemplo porque el formato de fecha es incorrecto o porque se ha tratado de introducir un texto en un número, Spring generará un error automáticamente, independientemente de que nosotros validemos o no.

Con el código de ejemplo, ese hipotético error lo tendríamos disponible en “resul”, y se uniría a los que generaríamos con el método “isValid()”.

Tal como está, el código no comprueba que la lista de errores suministrada esté vacía; Es posible que se hayan producido errores previos de formateadores o de editores de propiedades. Si se quiere que compruebe también los errores de conversión previos hay que cambiar la última línea del método isValid(), aunque quedaría más elegante comprobarlo antes de llamar al método:

```
//return violations.isEmpty();
return !result.hasErrors();
```

8.8 Anotación @Validated

Aunque ya existía en la versión tres de Spring, sólo funciona correctamente a partir de la **versión cuatro**. Se usa como **reemplazo** de @Valid, precisamente para solventar los problemas comentados en el punto anterior; Esta anotación sí que nos permite especificar qué grupos de anotaciones queremos usar.

El ejemplo anterior quedaría:

```
RequestMapping(value="modificar.html")
public String procesarModificar(@Validated(ValidarSalario.class) Empleado empleado,
                                BindingResult resul) {
    if (resul.hasErrors())...
    ...
}
```

Esta anotación no es estándar, pertenece al marco de trabajo Spring.

8.9 Problema con Glassfish 3.1 y anotaciones múltiples

Al parecer hay un problema con el paquete “bean-validator.jar” suministrado con Glassfish, que provoca errores cuando se anota una propiedad con dos o más restricciones:

```
@Length(min=3,max=50)
@NotNull
private String nombre;
```

Esto provoca NullPointerException “en algún sitio” que a su vez provoca el error “javax.validation.ValidationException: Call to TraversableResolver.isReachable() threw an exception”. Seguramente se corregirá en futuras versiones (o bajando un paquete más actualizado y reemplazándolos módulos del servidor). Mientras tanto, si es necesario unir varias validaciones se puede emplear un truco:

```
@Size(min=3,max=50)
@NotNull
@Constraint(validatedBy={})
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.PARAMETER})
public @interface SinNulosYTam {
    public String message() default "la compuesta";
     Class<? extends Payload>[] payload() default {};
    Class<?>[] groups() default {};
}
```

Basta con escribir una anotación sin clase para resolver la validación pero que a su vez esté anotada por todas las restricciones que nos interesen.

Curiosamente sólo es necesario cuando se aplican a la vez. Si pertenecen a diferentes grupos de validación (y se aplican por separado) funcionarán correctamente.