Validación e internacionalización con Spring MVC

Índice

1. Validación en Spring

Desde la versión 3, Spring ha simplificado y potenciado en gran medida la validación de datos, gracias a la adopción de la especificación JSR 303. Este API permite validar los datos de manera declarativa, con el uso de anotaciones. Esto nos facilita la validación de los datos introducidos en formularios HMTL antes de llegar al controlador. Nosotros veremos aquí la validación en el módulo MVC, que es lo que nos interesa, aunque ésta se puede aplicar a cualquier capa de nuestra aplicación Spring.

1.1. JSR 303 - Bean Validation

Como ya hemos comentado, la especificación JSR 303 permite especificar la validación de datos de manera declarativa, usando anotaciones. La implementación de referencia de este JSR es Hibernate Validator, de la que recomendamos consultar su documentación . No obstante, en nuestra discusión vamos a restringirnos al estándar, así que todo lo que vamos a ver es válido para cualquier otra implementación.

Veamos un ejemplo que nos mostrará lo sencillo y potente que es este API. Siguiendo con las ofertas de hoteles del tema anterior, la siguiente clase representaría una reserva de habitación hecha por un cliente, expresando las restricciones que debe cumplir cada campo con JSR 303:

```
package es.ua.jtech.spring.modelo;
import java.util.Date;
import javax.validation.Valid;
import javax.validation.constraints.Future;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;

public class Reserva {
    @Future
    private Date entrada;
    @Range(min=1,max=15)
    private int noches;
    @Min(10)
    private BigDecimal pagoAnticipado;
    @NotNull
    private TipoHabitacion tipohabitacion;
    @NotNull
    private Cliente cliente;
    //ahora vendrían getters y setters
    ...
```

Aquí hemos especificado que la fecha de entrada en el hotel debe ser posterior a la fecha actual (la del sistema), se puede reservar entre 1 y 15 noches de estancia, por anticipado se debe pagar un mínimo de 10 euros y que el cliente y el tipo de habitación no pueden ser nulos. Nótese que especificar una restricción no implica que el objeto la deba cumplir siempre (evidentemente en los históricos habrá reservas con fecha de entrada en el pasado). Simplemente, la especificación nos proporciona una manera sencilla de expresar

las restricciones y de comprobar si se cumplen en el momento que nos interese, llamando a un método del API. Normalmente dispararemos la validación al dar de alta o editar el objeto. Aunque disparar la validación es muy sencillo usando directamente el API del JSR (solamente hay que llamar a `Validator.validate()` sobre el objeto a validar) veremos que en Spring es todavía más simple, gracias a su integración con dicho API.

Aunque como se ha visto se pueden anotar las propiedades, también se pueden anotar los getters para obtener el mismo efecto (pero NO se deben anotar los setters).

La especificación ofrece un amplio conjunto de restricciones predefinidas. Hibernate Validator proporciona algunas adicionales, y además el usuario puede definirse las suyas propias. Aunque no es excesivamente complicado esto queda fuera del alcance de este tema. Se recomienda consultar la documentación de Hibernate Validator para ver la lista de las restricciones predefinidas y si lo deseas, sobre la definición de restricciones propias del usuario.

Por defecto, si un objeto referencia a otros, al validarlo no se comprobarán las restricciones de los referenciados. Aunque en algunos casos nos puede interesar lo contrario, por ejemplo validar al `Cliente` que ha hecho la reserva. Para esto se usa la anotación `@Valid`:

```
public class Reserva {
    ...
    @NotNull
    @Valid
    private Cliente cliente;
    //ahora vendrían getters y setters
    ...
```

Esta "validación recursiva" se puede aplicar también a colecciones de objetos, de modo que se validarán todos los objetos de la colección. Por ejemplo, si en una misma reserva se pudieran reservar varias habitaciones a la vez, podríamos tener algo como:

```
public class Reserva {
    ...
    @NotNull
    @NotEmpty
    @Valid
    private List<TipoHabitacion> habitaciones;
    ...
```

Donde la anotación `@NotEmpty` significa que la colección no puede estar vacía.

1.2. Validación en Spring MVC

La activación de la validación JSR 303 en Spring MVC es muy sencilla. Simplemente hay que asegurarse de que tenemos en el classpath una implementación del JSR. (la más típica es Hibernate Validator, como ya hemos comentado) y poner en el XML de definición de beans de la capa web la etiqueta `<mvc:annotation-driven/>`.

En la capa MVC el caso de uso más típico es validar el objeto cuando el usuario haya rellenado sus propiedades a través de un formulario. Normalmente en Spring MVC ese objeto nos llegará como parámetro del método del controller que procese los datos del formulario. Para validar el objeto antes de pasárselo al método lo único que hay que hacer es anotar el parámetro con `@Valid`. Por ejemplo:

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

import es.ua.jtech.spring.modelo.Reserva;
import javax.validation.Valid;
import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/reserva")
public class ReservaController {

    @RequestMapping(method=RequestMethod.POST)
    public String efectuarReserva(@Valid Reserva reserva,
                                  BindingResult result) {
        if (result.hasErrors())
           return "rellenarReserva";
    }
}
```

Como vemos, la validación declarativa se integra con el API de Spring de manejo de errores. Los posibles errores de validación se almacenarán en el parámetro de tipo `BindingResult`, dándonos la posibilidad de examinarlos con el API de Spring, o saltar de nuevo al formulario y mostrarlos con la etiqueta `<form:error>`, como ya vimos en el tema anterior.

Los mensajes de error se buscarán en un fichero .properties, al igual que en el tema anterior. La clave bajo la que se buscará el mensaje es generalmente el nombre de la restricción. El primer parámetro que se suele pasar es este nombre, y a partir de aquí los parámetros de la anotación. Por ejemplo, en el caso de la restricción `@Min(1)` para las noches de estancia podríamos hacer en el .properties

```
Min.noches = hay un mínimo de {1} noche de estancia
```

Cuidado, el `{1}` no significa literalmente un 1, sino que hay que sustituir por el segundo argumento (recordemos que empiezan en 0). En este caso la sustitución sería por el valor 1, pero así podemos cambiar la restricción para forzar más de una noche de estancia sin cambiar el código del mensaje de error.

Aunque por defecto la clave bajo la que se busca el mensaje en el properties es el nombre de la restricción, podemos cambiarla en la propia anotación, por ejemplo:

```
public class Reserva {
   ...
   @Min(value=1,message="minimoNoches")
```

```
   private int noches;
   ...
}
```

Podemos efectuar la validación llamando directamente al API JSR303. Esto puede ser útil en los casos en que no podamos usar la anotación @Valid en nuestro código. Por ejemplo, @Valid no funciona en la versión 3.0 (aunque sí a partir de la 3.1 inclusive) cuando los datos de entrada vienen el cuerpo de la petición como JSON o XML en lugar de ser parámetros HTTP (típico de REST). Aun en ese caso no hay mucho problema, ya que el API JSR303 es bastante sencillo:

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();
Set<ConstraintViolation<Oferta>> errores = validator.validate(oferta);
for (ConstraintViolation<Oferta> cv : errores) {
      System.out.println(cv.getMessage());
}
```

Para aplicar la validación solo hay que construir un `Validator` a través de una `ValidatorFactory` y llamar al método `validate` sobre el objeto a validar. La aplicación de la validación produce por cada error detectado un `ConstraintViolation`, parametrizado al tipo que se está validando. Este objeto contiene información sobre el error que se ha producido. Para más información se recomienda consultar la documentación del API.

2. Internacionalización y formateo de datos

Veremos en esta sección cómo preparar nuestra aplicación para que esté adaptada al idioma del usuario. A este proceso se le denomina **internacionalización** o *i18n* para abreviar (ya que en la palabra hay 18 letras entre la 'i' inicial y la 'n' final).

2.1. Internacionalización de los textos

La parte más tediosa de la i18n de una aplicación suele ser la traducción y adaptación de los mensajes del interfaz de usuario. En aplicaciones Java internacionalizadas casi siempre los mensajes se suelen almacenar en ficheros *properties* para no tener que modificar el código Java si hay que cambiar/añadir alguna traducción. La convención habitual es que los mensajes para cada idioma se almacenan en un fichero .properties separado, pero de modo que todos los ficheros comienzan por el mismo nombre aunque como sufijo del mismo se usa el *locale* del idioma en cuestión.

Un *locale* es una combinación de idioma y país (y opcionalmente, aunque la mayoría de veces no se usa, una variante o dialecto). Tanto el país como el idioma se especifican con códigos ISO (estándares ISO-3166 e ISO-639). Aunque el locale en Java exige especificar tanto idioma como país, en Spring y en casi todos los frameworks podemos usar solamente el código de idioma si no deseamos especificar más. De este modo, los

mensajes internacionalizados se podrían guardar en archivos como:

```
mensajes_es_ES.properties   //Español de España
mensajes_es_AR.properties   //Español de Argentina
mensajes_es.properties      //Español genérico a usar en otro caso
mensajes_en.properties      //Inglés
mensajes.properties         //fallback, o fichero a usar si no hay otro
apropiado
```

Así, el fichero mensajes_es_ES.properties, con los mensajes en idioma español (es) para España (ES), podría contener algo como lo siguiente:

```
saludo = Hola, bienvenido a la aplicación
error= lo sentimos, se ha producido un error
```

El *framework* usará automáticamente el fichero apropiado al locale actual. En caso de poder usar varios se elegirá el que mejor encaje. Así, si en el ejemplo anterior el locale actual fuera idioma español y país España se usaría el primer archivo en lugar del español genérico (que se seleccionaría por ejemplo si el locale actual fuera idioma español y país México). Si no hay ninguno que encaje se usará el que no tiene sufijo de locale (en nuestro ejemplo, el último de los archivos).

Como hemos visto en la sesión anterior, los mensajes de error en Spring se almacenan también en ficheros .properties y su localización se define habitualmente con un bean de tipo `ResourceBundleMessageSource`. Para los textos de la interfaz web se usa el mismo mecanismo. En realidad los mensajes de error también se pueden internacionalizar automáticamente de la misma forma que los demás mensajes, simplemente generando los .properties adecuados.

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

Solo nos falta colocar los mensajes internacionalizados en la interfaz web. Suponiendo, como hemos hecho hasta ahora, que usamos JSP para la interfaz, podemos emplear la etiqueta `<spring:message/>`. Esta etiqueta tiene como atributo principal `code`, que representa la clave del properties bajo la que está almacenado el mensaje. Por ejemplo:

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
...
<spring:message code="saludo"/>
```

de modo que en ese punto del JSP aparecería el mensaje que en el .properties del locale actual estuviera almacenado bajo la clave "saludo".

2.2. Cambio del locale

Para que una aplicación pueda estar internacionalizada se tiene que guardar de alguna manera cuál es el locale con el que desea verla el usuario. El sitio más habitual para guardar el locale es la sesión o una cookie en el navegador, consiguiendo así que cada usuario pueda mantener el suyo propio. Además tiene que haber algún mecanismo para consultar cuál es el locale actual y cambiarlo. Spring da soporte a todo esto: ofrece interceptores que nos permitirán cambiar el locale sin más que llamar a determinada URL y tiene clases propias que guardan el locale del usuario actual. Veamos cómo funciona todo esto.

Por defecto Spring toma el locale de las cabeceras HTTP que envía el navegador con cada petición. Por tanto, si el usuario tiene configurado el navegador en el idioma deseado, la aplicación aparecerá automáticamente en el mismo idioma. No obstante, es posible que el usuario tenga instalado el navegador en un idioma pero prefiera ver las páginas en uno distinto. En ese caso tenemos que cambiar el lugar de donde toma Spring el locale, pasando a guardarlo en una cookie o en la sesión, según la implementación elegida de la clase que guarda el locale. Además, podemos configurar un interceptor que permitirá cambiar el locale de manera sencilla. Esto lo configuraremos en el XML de la capa web (típicamente `dispatcher-servlet.xml`)

```
...
<!-- el id debe ser "localeResolver -->
<bean id="localeResolver"
class="org.springframework.web.servlet.i18n.CookieLocaleResolver"/>

<mvc:interceptors>
   <bean
class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"/>
</mvc:interceptors>
...
```

En el fragmento de XML anterior, el `localeResolver` es el bean encargado de guardar el locale del usuario. La implementación elegida en este caso lo guarda en una cookie, como puede deducirse del nombre de la clase. Para guardarlo en la sesión usaríamos en su lugar la clase `SessionLocaleResolver`. Como decíamos antes, por defecto en Spring el `localeResolver` usa la cabecera `Accept-language` de las peticiones del navegador, pero este tiene el problema de que no podemos cambiar el locale (si lo intentamos se generará un error).

Por otro lado, la clase `LocaleChangeInterceptor` es un interceptor. Es decir, es una clase que intercepta las peticiones HTTP antes de que lleguen a los controllers. Esta clase cambia el locale si la petición HTTP actual tiene un parámetro `locale` igual al locale deseado. Por ejemplo, la petición `index.do?locale=en` cambiaría el locale actual a inglés, además de hacer lo que haga normalmente el controller asociado a `index.do`. Nótese que esto lo podríamos hacer "a mano", obteniendo una referencia al `localeResolver` actual y llamando a un método en él para cambiar el locale, pero es mucho más cómodo hacerlo con el interceptor.

2.3. Formateo de datos

Según el locale, existen convenciones distintas en el formato para mostrar fechas y números. Sin Spring esto lo haríamos con etiquetas JSTL (por ejemplo . Sin embargo, para introducir datos en formularios hay que hacer la conversión inversa: de String a Date o a número, cosa que no podemos solucionar con JSTL. En Spring 3 se introducen varias anotaciones para expresar el formato de los campos, que internamente usan clases ("formatters") capaces de convertir a y desde String a otros tipos. Ya hay anotaciones definidas para formatear fechas y números, aunque podríamos hacer nuestros propios "formateadores".

Veamos cómo podríamos formatear los campos de la clase `Reserva`:

```
public class Reserva {
    @Future
    @DateTimeFormat(style="S-")
    private Date entrada;
    @Range(min=1,max=15)
    private int noches;
    @Min(10)
    @NumberFormat(style=NumberFormat.Style.CURRENCY)
    private BigDecimal pagoAnticipado;
    @NotNull
    private TipoHabitacion tipohabitacion;
    @NotNull
    private Cliente cliente;
    //ahora vendrían getters y setters
    ...
```

La anotación `@DateTimeFormat` formatea fechas. El atributo `style` indica el estilo de formateo con dos caracteres. El primer carácter es para el estilo de la fecha y el segundo para el de la hora. Se admiten los caracteres "S" (short), "M", (medium), "L" (long), "F" (full) y el guión que indica que no nos interesa mostrar esa parte. Así, en el ejemplo, `@DateTimeFormat(style="S-")` indica que queremos ver la fecha en formato corto, sin hora. En el locale español, por ejemplo el 28 de diciembre de 2009 a las 12:00 se mostraría como 28/12/09 (recordemos que no nos interesaba la hora), mientras que en el locale inglés sería 12/28/09. Se recomienda consultar el javadoc del API de Spring para más información sobre otros modos de formatear fechas.

La librería JodaTime

Hasta la versión 3.1 de Spring incluida, la anotación `@DateTimeFormat` requería que la librería JodaTime estuviera presente en el classpath. Esta librería *open source* no es parte de Spring, pero se puede obtener, junto con las demás dependencias de Spring, de la web de SpringSource, o bien de su propia web. JodaTime es una implementación alternativa al `Date` y `Time` del API de Java. Si has usado alguna vez fechas en Java habrás visto que su API es potente y flexible, pero también...digamos...algo retorcido. JodaTime simplifica considerablemente el manejo de fechas, manteniendo la flexibilidad si es necesaria. A partir de la versión 3.2, ya no es obligatorio el uso de JodaTime para esta anotación, pero si está disponible se usará en lugar de la librería estándar.

Como habrás "adivinado", la anotación `@NumberFormat` formatea números. El parámetro

style puede tener valores distintos indicando si queremos mostrar un número (con coma decimal en el locale español, o punto decimal en el ingles, por ejemplo), una moneda (aparecerá el símbolo de la moneda del locale actual) o un porcentaje.

Con esto podemos leer y convertir datos de entrada en formularios. O sea, de String al tipo deseado. Para el caso contrario, en el que queremos mostrar un dato y que aparezca formateado (de objeto a String), podemos usar la etiqueta `<spring-eval/>`

```
Fecha de entrada: <spring:eval expression="reserva.entrada" />
```

Igual que se usan para convertir datos introducidos en formularios, estas anotaciones también se pueden usar para convertir parámetros HTTP al tipo deseado, por ejemplo:

```
public class TareasController {
  @RequestMapping("tareas/crear")
  public int nuevaTarea(@RequestParam
                        @DateTimeFormat(style="S-") Date fecha, ...) {
    ...
  }
}
```