ANEXO: PROBLEMAS ENCONTRADOS

Investigación e implementación de formateadores y parseadores

Introducción

Cuando comenzamos a investigar sobre los formateadores y parseadores en Spring, teníamos la siguiente idea: que el desarrollador pudiese definir en una etiqueta el formato con el que quería que se mostrara un campo determinado, o cómo debía ser el formato de parseo, en el caso de que se tratara de un input. El formato se definiría en todo momento por medio de un recurso de internacionalización.

Alternativas

Comenzamos investigando los formatters de Spring. En Spring, un formatter se compone de un parser (que interpreta una cadena recibida por el usuario con un formato concreto) y un *printter* (formatea los datos que recibe de la aplicación para mostrárselo al usuario de forma adecuada). Los formatters se crean implementando las interfaces *Parser<>* y *Print<>*. Para que el framework hiciera uso de ellos, había que declararlos en un fichero de configuración. Nosotros lo hicimos dentro de *converters.xml*. Por ejemplo, para añadir un formateador por defecto de Spring para fechas (las formateaba según el *Locale*, es decir, según el idioma y la región del usuario):

```
<bean class="org.springframework.format.datetime.DateFormatter">

      <property name="stylePattern" value="SS"/><!-- Indica que la fecha
      se mostrará con un formato corto -->

</bean>
```

También se podía formatear campos concretos creando una anotación que se le añadiese al modelo de dominio, en el método *get* del campo al que quisiésemos aplicarle el *formatter*. El formato se podía especificar en la etiqueta o en un recurso de internacionalización. Sin embargo, descartamos esta alternativa ya que estábamos metiendo algo en el dominio que debería ir especificado en la vista, no en la definición de una entidad.

Otra alternativa era utilizar las etiquetas *fmt* en los ficheros JSP. El problema era que si bien se podía formatear los datos que venían del servidor, no era posible que dada una cadena con un formato personalizado escrito por el usuario, se enviara parseada al controlador. Por este motivo descartamos esta alternativa, a pesar de ser la más se ajustaba a nuestra primera idea en un principio.

La alternativa que escogimos fue la siguiente: realizar el formateo o el parseo en el controlador. Mediante un método que lleve la etiqueta *@InitBinder*, se pueden aplicar editores a determinados atributos de una entidad. Esta etiqueta hace que se llame al método al que se le aplique cada vez que se llame al controlador.

El método en el que se registra los editores se llama *initBinder*, está en *GetController* y recibe un objeto de tipo *WebDataBinder*. Para registrar un editor, debemos pasárselo

como parámetro al método *registerCustomEditor* del objeto *WebDataBinder*. Los detalles de implementación están especificados en los apartados 5.2.4 y 5.4 de la documentación oficial de Sprouts Framework.

Posteriormente, se decidió utilizar la clase *Formatter* de Java para darle formato a las cadenas, permitiendo la definición de formatos mediante una única cadena por cada tipo de dato. Se encontraron diversos problemas:

- Hemos probado el método *format* de String. Hemos conseguido formatear una fecha. La cadena que hay que especificar es poco legible: *%1$td/%1$tm/%1$tY %1$tH:%1$TM.* Para formatear números decimales, utilizando una coma en vezde un punto como separador decimal, usaríamos la siguiente cadena: *%(,.2f)*.
  - Posibles formatos para cadenas: https://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html
  - El siguiente problema es el uso de un editor, que es con lo que trabaja el método *initBinder* para formatear campos. No existen editores para formatear / parsear cadenas:
  - http://docs.spring.io/spring/docs/current/spring-framework-reference/html/validation.html
  - Es posible crearlo extendiendo la clase *PropertyEditorSupport* y redefiniendo los métodos getAsText (convierte el objeto a cadena) y *setAsText* (devolver una instancia de una clase a partir de su representación en cadena).
  - Mediante *String.format()* podríamos formatear un objeto. Pero para parsearlo tendríamos que recurrir a los formatters de cada tipo en particular.
  - Por otra parte, hemos intentado especificar el formato a de un número decimal empleando únicamente una cadena “##,###.##”. Con esto no podemos especificar que queremos que el separador decimal sea una coma en vez de un punto. Lanza una excepción al pasarle “##.###,##”.
  - http://tutorials.jenkov.com/java-internationalization/decimalformat.html Especifica claramente que el carácter “,” sirve para indicar separación de centenas, y el carácter “.” para separar decimales. La única forma de cambiar los separadores decimales o de centenas es pasándoselo explícitamente mediante un objeto de tipo *DecimalFormatSymbols*.

Estructura de paquetes

En este apartado se documenta una posible alternativa para estructurar los paquetes del proyecto, que fue estudiada durante el desarrollo de Sprouts Framework. Esta es similar a la que actualmente provee por defecto, pero la organización de los repositorios, servicios y controladores es diferente.

La reorganización que se intentó realizar se hizo con la intención de mejorar el trabajo en grupo, y separar los repositorios, servicios y controladores por casos de uso, de tal forma que un programador con un caso de uso asignado, no tuviese que modificar el código elaborado por otro programador.

Para ello se planteó la siguiente estructura:

- useCases
  - useCase1
    - repositories
    - servicies
    - controllers
  - useCase2
    - repositories
    - servicies
    - controllers

De esta forma, habría tantos repositorios, servicios y controladores por entidad como se necesitasen. Estos tendrían el mismo nombre y esto suponía un problema. Aunque se distinguían por los nombres asignados dentro de las clases mediante anotaciones (@Service("useCase1"), esto evitaba problemas de configuración, pero no era práctico. Los motivos por los que se decidió descartar esta idea, fueron dos:

- El primero es que cuando se hacía uso de un repositorio o servicio, a la hora de importarlo surgían problemas al haber varios con el mismo nombre, y había que indicarle de manera manual el paquete donde se encontraba. Esto dificulta la programación y podía dar lugar a errores, algo contra lo que Sprouts-Framework lucha.
- El segundo motivo, es que aún separando los casos de uso de esta manera, había ocasiones en los que un programador tenía que modificar el código de otro para poder llevar a cabo su caso de uso, por lo que no se conseguía el efecto esperado.

Implementación de Datatables

Era necesario poder añadir imágenes, URLs, personalizar las columnas y muchas más posibilidades a las columnas del componente Datatables. En un primer intento se trató de hacer de forma que esto se realizara desde el fichero .jsp de la etiqueta *data-table*. Pero se descubrió que no era posible, dado que el cuerpo de las tablas es generado de forma automática por el componente, a través de sus funciones con JavaScript. Se probaron posibles alternativas que proporcionaba el componente, para poder editar el contenido de una celda, pero no tuvieron éxito.

Se optó por modificar el método *getRows* de la clase *DefaultTilesViewTableBuilder*. De esta forma se modifica el JSON que se envía al lado del cliente. No es una solución muy limpia, puesto que los ficheros JSON se plantean para tener una estructura clara y sencilla, y actualmente se le están insertando etiquetas HTML y más cosas que hacen que no sea JSON puro.

Eliminación del archivo persistence.xml

No se ha podido dar una solución completa a este tema, pero se ha minimizado el impacto de este fichero.

Dado que Spring e Hibernate buscan el fichero *persistence.xml* para realizarle un parseo con el fin de encontrar la configuración contenida en él, no se ha podido eliminar. Sin embargo, el fichero tan solo tiene que estar presente en la ruta que Hibernate y Spring lo buscarán. El contenido del fichero es mínimo, ya que tan solo contiene la etiqueta <*persistence*> con algunas indicaciones para indicar que se trata de un archivo de persistencia.

Las propiedades que antes debían estar en el fichero *persistence.xml* no son necesarias, ya que ahora se configuran en el fichero *datasource.xml*.

De esta manera, cuando se cree un *EntityManagerFactory*, se creará a partir de la información del *datasource.xml* y no como se realizaba antes, que era con la información de la unidad de persistencia configurada en el *persistence.xml*.

Respecto a las utilidades *QueryDatabase* y *PopualteDatabase*, la utilidad *PopulateDatabase* estaba adaptada para que el *EntityManagerFactory* se creará a partir de la información contenida en el fichero *datasource.xml*. Sin embargo, la utilidad *QueryDatabase* no estaba adaptada, y seguía haciendo uso de la unidad de persistencia configurada en el *persistence.xml* para crear el *EntityManagerFactory*, así que se ha adaptado para que se realice de igual manera que en el *PopulateDatabase* y se pueda eliminar toda la configuración redundante del fichero.xml.

Redirección al loguear usuarios

Cuando un usuario se registraba, y queríamos redirigir a otra vista donde necesitaba estar logueado, al loguearse, salía un error 403.