



 Escuela Técnica Superior de
Ingeniería Informática



Documentación Oficial

Manual de usuario

Realizado por:

Javier Bonilla (javierbonillad@gmail.com)

Daniel Moreno (morenomdani@gmail.com)

Álvaro Valencia (alvarovalenciavp@gmail.com)

Dirigido por:

Rafael Corchuelo

RESUMEN

En este documento se presenta el manual de usuario de Sprouts Framework. Es un framework realizado sobre la base de Spring Framework 3. Esta herramienta tiene como finalidad mejorar un servicio ya existente para que sea más sencillo para los usuarios. De esta forma se pretenden ahorrar costes y mejorar la eficiencia en el trabajo al permitir realizar las mismas tareas en un menor tiempo y al ser menos propenso a errores al simplificar las partes más conflictivas.

El objetivo de este documento es presentar un manual para los usuarios que quieran hacer uso de este framework. En el mismo quedarán plasmados todos los detalles necesarios, para poder comprender el funcionamiento de la plantilla proporcionada como base para comenzar a trabajar. A lo largo de este manual podrá consultar cualquier aspecto técnico acerca del proyecto para aclarar para qué sirve este y cómo funciona el código implementado. Antes de consultar el manual de usuario, se recomienda visitar el documento correspondiente al Getting-Started.

El contenido sobre el que versará este manual es, principalmente, dar una explicación al código proporcionado. Además, se presentan unas pautas a seguir antes de comenzar a operar con la plantilla proporcionada. Se comenta el modelo de dominio, donde se tratan diversos aspectos para poder reflejar de forma fiel el dominio del problema a resolver con esta herramienta. También los repositorios, imprescindibles como punto de comunicación entre la aplicación y el sistema de gestión de base de datos. Los servicios, uno de los puntos más destacados del framework, ya que es donde recae el peso de las distintas operaciones que se realizan sobre las entidades. Los controladores, que también gozan de una gran relevancia y por ello están explicados con gran nivel de detalle, dado que son los que deben orquestar sobre quien recae el peso de las distintas peticiones que realizan los clientes del sistema de información web a desarrollar. Las vistas, donde se define cómo se va a mostrar la información a los clientes, dentro de esta sección, hay una parte dedicada especialmente al componente Datatable ya que tiene una cierta complejidad que hace que requiera una explicación más detallada. Además, se comenta cómo se lleva a cabo la integración con otros componentes para así poder hacer uso de dichos componentes. De igual forma se trata sobre qué elementos extra respecto al framework base proporciona nuestra herramienta, como es el aspecto de la seguridad frente al hacking.

CONTENIDO

Introducción

1. Preparando el entorno de desarrollo

1.1 Introducción

1.2 Importando y configurando la plantilla

1.3 Estructura de paquetes

1.3.1 Estructura de paquetes por defecto

1.3.2 Modificar la estructura de paquetes

1.4 Conclusión

2. Modelo de dominio

2.1 Introducción

2.2. Estructura de clases

2.3 Entidades de dominio

2.4. Datatypes

Alternativas en la implementación de Datatypes

Alternativa 1: implementación como @Embeddable

Alternativa 2: implementación como Serializable

Alternativa 3: implementación como @ElementCollection con solo un elemento

2.5. Forms

2.6. Conclusión

3. Repositorios

3.1 Introducción

3.2 Definición de repositorios

3.3 Paginación de repositorios

3.4 Conclusión

4. Servicios

- 4.1 Introducción
 - 4.2 Estructura de clases
 - 4.3 Servicios para entidades: AbstractService
 - 4.3.1 CreateService
 - 4.3.2 UpdateService
 - 4.3.3 DeleteService
 - 4.3.4 ListService
 - 4.3.5 ShowService
 - 4.3.6 CrudService
 - 4.4 Servicios para objetos de tipo Form. AbstractFormService
 - 4.4.1 CreateFormService
 - 4.4.2 UpdateFormService
 - 4.5 Validadores y reglas de negocio
 - 4.5.1 Creación de reglas de negocio
 - 4.5.2 Creación de validadores
 - 4.6 Ejecución de procedimientos almacenados desde servicios
 - 4.7 Conclusión
- 5 Controladores
- 5.1 Introducción
 - 5.2 Controladores del núcleo
 - 5.2.1 Authorizable
 - 5.2.2 Postable
 - 5.2.3 BaseController
 - PathVariables y contexto
 - Métodos de construcción de vistas
 - 5.2.4 GetController
 - 5.2.5 PostController
 - 5.2.6 Controladores de listado

- 5.2.7 SecurityController
 - 5.3 Controladores para casos de uso
 - 5.3.1 AbstractCreateController
 - 5.3.2 AbstractUpdateController
 - 5.3.3 AbstractDeleteController
 - 5.3.4 AbstractListController
 - 5.3.5 AbstractShowController
 - 5.3.6 AbstractPostController para entidades
 - 5.3.7 AbstractPostController para datatypes
 - 5.3.8 AbstractGetController
 - 5.3.9 Añadir atributos adicionales al modelo: interfaz AddsToModel
 - 5.4. Formateadores y parseadores
 - 5.4.1 Implementación de formateadores y parseadores
 - 5.4.2 Formateador de fechas: CustomDateFormat
 - 5.4.3 Formateador de números decimales: CustomDecimalFormat
 - 5.4.4 Formateador de unidades monetarias: CustomCurrencyFormat
 - 5.5 Conclusión
- 6. Componente datatables
 - 6.1. Introducción
 - 6.2. Clases de utilidad asociadas a DataTables
 - 6.2.1. DatatableJson
 - 6.2.2. Table
 - 6.2.3. Column
 - 6.2.4 TableBuilder
 - 6.3. Controladores asociados a DataTable
 - 6.3.1. GetCollectionController
 - 6.3.2. AbstractGetListDataController
 - 6.4. Construcción de las tablas

6.5. Conclusión

7. Vistas

7.1 Introducción

7.2 Tiles

7.3 Ficheros JSP

7.4 Ficheros de internacionalización

7.5 Etiquetas

7.4.1 Action-button

7.4.2 Alert

7.4.3 Button

7.4.4 Cancel button

7.4.5 Checkbox

7.4.6 Data-column

7.4.7 Data-table

7.4.8 Display-column

7.4.9 Display-image-column

7.4.10 Form

7.4.11 Hidden-field

7.4.12 Password-input

7.4.13 Pictures-list

7.4.14 Protected

7.4.15 Select

7.4.16 Social-account-sign-in

7.4.17 Submit-button

7.4.18 Submit-or-cancel

7.4.19 Textarea-input

7.4.20 Textbox-input

7.5 Clases relacionadas con las vistas

7.6 Conclusión

8 Integración con otros componentes

8.1 Introducción

8.2 Integración con Spring Social

8.3 Creación de reglas de negocio con Drools

8.4 Búsquedas full-text

8.5 Conclusión

9 Seguridad y prevención de hacking

9.1 Introducción

9.2 Estructura de clases

9.2 Protección contra el acceso no autorizado

9.3 Protección contra el hacking post

9.4 Conclusión

Conclusión

Bibliografía

INTRODUCCIÓN

El presente documento forma parte de la documentación oficial de Sprouts Framework.

Sprouts Framework está formado esencialmente por una plantilla de workspace para Maven, un proyecto plantilla que contiene todo lo necesario para construir su sistema de información web usando este framework, y un proyecto de utilidades que provee herramientas útiles durante el desarrollo de su aplicación web.

Sprouts Framework está construido sobre Spring Framework 3^[1]. Sprouts hereda, como filosofía principal: lo provisto es inmutable, lo que implica que si el framework incluye algún archivo configurado por defecto, ese archivo no debería ser modificado. Gracias a esta filosofía, se reduce la posibilidad de cometer fallos durante el desarrollo del proyecto. Sprouts Framework, evita que el desarrollador tenga que realizar tareas repetitivas con alta probabilidad de errores, y además como valor añadido se ofrece versatilidad. Se han llevado a cabo cambios y adaptaciones sobre la base de Spring para ofrecerle al usuario una gran flexibilidad. Con Sprouts, cualquier caso de uso podrá ser implementado por el desarrollador. Además, el desarrollador no tendrá que repetir código costoso y con alta probabilidad de fallos.

Sprouts ofrece integración con Spring Social^[2], Hibernate Search^[3] y Drools^[4], tecnologías que están a la orden del día en cuanto al desarrollo de sistemas de información web.

Este documento es el manual de usuario de Sprouts Framework. En él podrá encontrar información técnica y detallada sobre los componentes de Sprouts. Podrá ver detalles de implementación, y de forma continua se mostrarán ejemplos para que pueda sacar el máximo rendimiento a los componentes que este framework le ofrece.

En primer lugar, le enseñaremos a preparar su entorno de desarrollo, y a poner en marcha la plantilla de Sprouts Framework. Una vez montada la plantilla, describiremos aspectos básicos para que pueda implementar su aplicación web. Hablaremos del modelo de dominio, repositorios, servicios, controladores y vistas, mostrándole la estructura de clases que provee Sprouts Framework para facilitar el desarrollo. Para cada uno de estos apartados, seguiremos un enfoque down-top: empezaremos describiendo la estructura de clases internas, hasta llegar al nivel de abstracción más alto, esto es, las clases que se deben utilizar para implementar nuestros casos de uso.

Posteriormente, veremos en profundidad el componente datatable, el cual nos dotará de tablas de datos, con una implementación altamente eficiente.

El penúltimo punto tratado en esta documentación trata sobre las tecnologías adicionales que integra Sprouts Framework: Spring Social, Drools y Hibernate Search. El primero permite al usuario acceder a la aplicación mediante sus redes sociales, el segundo permite al desarrollador crear reglas de negocio complejas en un lenguaje sencillo e intuitivo, y el último proporciona soporte para búsquedas full-text.

El último apartado está dedicado a la seguridad y prevención de hacking en Sprouts Framework. Describiremos los mecanismos que Sprouts ofrece para prevenir el acceso no autorizado a recursos y el hacking post.

Acompañaremos las explicaciones técnicas con diagramas de clases UML, referencias a la documentación oficial de la tecnología que estemos describiendo, y ejemplos prácticos, para que el programador entienda de forma rápida el uso de cada clase y componente de Sprouts Framework. Los ejemplos que se muestran en el documento son de los proyectos Acme-Six-Pack y Acme-Barter. Los puede encontrar junto a la plantilla.

1. PREPARANDO EL ENTORNO DE DESARROLLO

1.1 Introducción

En este apartado se explicará brevemente cómo dar los primeros pasos para preparar la plantilla y comenzar a usarla. También se explica la configuración de paquetes del proyecto y cómo modificarla si se desea.

1.2 Importando y configurando la plantilla

En primer lugar, es necesario hacer uso del workspace proporcionado junto con la plantilla del framework. En este workspace se debe descomprimir la plantilla del framework e importarla en Eclipse como proyecto Maven. Para poder trabajar satisfactoriamente, también será necesario realizar la misma operación con el proyecto de utilidades que también se proporciona.

Es importante, antes de empezar a trabajar, configurar ambos proyectos para personalizar el proyecto principal. Para ello, hay que modificar los datos identificativos del proyecto en el fichero *pom.xml* y *web.xml*. Una vez configurado este proyecto, es necesario enlazar el proyecto de utilidades con el proyecto principal para poder hacer uso de las utilidades que este contiene.

El siguiente paso de configuración necesario, es configurar las variables de entorno: *JPA_DB_NAME*, *JPA_DIALECT*, *JPA_DRIVER*, *JPA_MANAGER_PASSWORD*, *JPA_MANAGER_USERNAME*, *JPA_URL*, *JPA_USER_PASSWORD* y *JPA_USER_USERNAME*. Estas variables son necesarias para que en el archivo *datasource.xml* se realice la configuración correctamente.

Por último, es necesario crear la base de datos y dar los permisos necesarios.

i Toda esta información se explica detalladamente en el documento Getting-Started.

1.3 Estructura de paquetes

1.3.1 Estructura de paquetes por defecto

A la hora de realizar un proyecto, este se puede llevar a cabo con diferentes formas en cuanto a la organización de los paquetes.

En Sprouts-Project se ofrece una organización de los paquetes por defecto. Los paquetes que vienen configurado por defecto son:

controllers

Paquete donde se implementan los controladores de la aplicación. Estos se organizan de forma jerárquica: se crea un subpaquete por cada tipo de actor en el sistema, y un

subpaquete especial, llamado home, donde se sitúan los controladores que no dependan de un actor. A su vez, estos paquetes disponen de otros subpaquetes, uno por entidad sobre la que el controlador tiene efecto. En estos paquetes, se definen las clases que implementarán los controladores

La estructura por lo tanto quedaría así:

- Controllers
 - Actor1
 - Entity1
 - CreateController
 - UpdateController
 - DeleteController
 - ShowController
 - ListController
 - Entity2
 - Actor2
 - Entity1

datatype

En este paquete van los objetos del modelo de dominio que se modelen como un datatype.

domain

Paquete donde se habrán de situar todas aquellas entidades del modelo de dominio.

es.us.lsi.dp

Paquete base de Sprouts-Framework y fundamental para el funcionamiento del mismo. Este paquete es inmutable, y no debería modificarse ningún contenido que esté dentro del mismo.

formatters

Paquete en el que se definen los formadores que se usarán en el proyecto.

forms

Paquete donde se crean formularios auxiliares cuando sean necesarios.

repositories

Paquete donde se sitúan los repositorios. Para cada entidad, se dispondrá de un repositorio donde se añadirán todos los métodos de consulta que sean necesarios.

services

Paquete donde se sitúan los servicios. Habitualmente, existirá un servicio por entidad, pero no tiene porqué ser así. En casos en los que se necesite realizar una acción sobre una

entidad de forma diferente, es normal que exista un servicio donde se realice de forma normal, y otro servicio diferente, que realice una operación ligeramente distinta.

Por ejemplo, se puede actualizar una entidad mediante un formulario (forma típica), o actualizarla para cambiar solo un atributo de ella.

social

Paquete en el que se encuentra la clase *SocialConfig*, que se utiliza para la configuración de Spring Social.

utilities

Aunque se recomienda férreamente hacer uso del proyecto de utilidades para las utilidades, es posible situarlas en este paquete.

validation

Este paquete se estructura en dos subpaquetes. Estos subpaquetes son “rules” y “validation”. En el paquete “rules” se sitúan todas las reglas de negocio que se generen cuando sea necesario en los servicios mientras que en el paquete “validation” se sitúan los validadores.

1.3.2 Modificar la estructura de paquetes

Aunque no se recomienda cambiar la organización de los paquetes, esta se puede realizar modificando los archivos de configuración *packages.xml* y *data.xml*, situados ambos en *resources/spring/config*. En el fichero *packages.xml* se le indica a Spring los paquetes que debe escanear para que se encargue de configurar automáticamente las clases dentro de estos.

La configuración referida a los paquetes *es.us.lsi.dp*.** no debe tocarse para no perturbar el correcto funcionamiento del framework. Los demás paquetes pueden modificarse libremente para adaptarlo a la organización que se quiera, mediante la etiqueta *<context:component-scan base-package="new-package"/>*. Hay que tener en cuenta, que las clases que se sitúen en los nuevos paquetes, deben tener las anotaciones pertinentes para que sean leídas por Spring, por ejemplo, *@Controller*, *@Service*, *@Repository*, *@Component...*

En caso de que los repositorios hayan cambiado de lugar, es necesario indicarlo en el archivo *data.xml*, mediante la etiqueta *<jpa:repositories base-package="new-package"/>*

Otras estructuras de paquetes

El equipo de desarrollo de Sprouts-Framework estudió otras estructuras de paquetes que no prosperaron.

1.4 Conclusión

Tras este primer capítulo inicial, se obtienen los conocimientos básicos para preparar la plantilla y modificar su configuración, incluyendo la estructura de paquetes, así como la forma de modificar esta.

2. MODELO DE DOMINIO

2.1 Introducción

Para empezar a implementar un sistema de información web, el primer paso es crear el modelo de dominio. Para ello, hay que implementar las clases POJO de las entidades o datatypes que tenga nuestro proyecto. Para evitar tener que hacerlo totalmente desde cero, Sprouts-Framework proporciona ya algunas clases para simplificar este proceso y hacerlo menos repetitivo.

2.2 Estructura de clases

En esta sección nos centramos en comentar cual es la estructura de clases seguidas, para implementar el modelo de dominio, además de comentar algunas de las clases que se proporcionan también como utilidades. Estas clases están creadas todas dentro del paquete `es.us.lsi.dp` y son principalmente las siguientes: `Validable`, `DomainEntity`, `DomainForm`, `DomainObject`, `Datatype`.

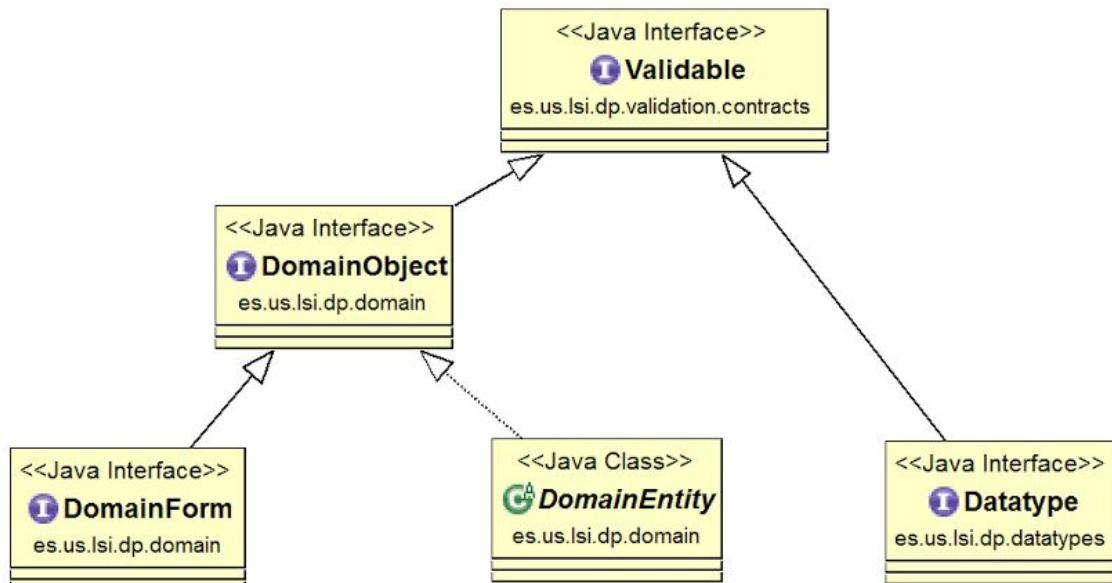


Diagrama de secuencia de la herencia de las clases implicadas.

Siguiendo el orden en el que aparecen en el árbol de herencia:

- **Validable:** esta interfaz es de la que heredan todas las interfaces que posteriormente serán implementadas y forman parte del modelo de dominio, ya sean entidades, formularios o datatypes. La funcionalidad de esta interfaz no es más que servir de

clasificador para la validación de los atributos de la clase en cuestión cuando se va a persistir en la base de datos. Ahora mismo no se necesita saber nada más allá de que es la raíz de este árbol de herencia.

En la sección correspondiente a la validación se comentará con detalle cómo se lleva a cabo esta y cómo entra en juego esta interfaz *Validable*.

- **DomainObject:** esta interfaz hereda de *Validable*, al igual que las demás. Sirve para que a la hora de programar los distintos métodos en los servicios y controladores abstractos, se pueda diferenciar entre las clases que representan a una entidad y aquellas que representan un Datatype.
- **DomainForm:** esta interfaz hereda de *DomainObject*. Esta interfaz, a su vez, debe ser implementada por todas aquellas clases que representen un objeto de tipo *Form*. En posteriores secciones de este apartado del modelo de domino se explicará detalladamente en qué consiste dicho objeto y cuando es útil su uso.

Al igual que su clase padre de la que hereda, sirve para poder diferenciar que clases representan un *Form* cuando se programan los métodos en servicios y controladores de forma genérica.

- **DomainEntity:** es una clase que representa una entidad del modelo de dominio, por ello es imprescindible que cualquier entidad propia de nuestro dominio extienda de esta. Implementa la interfaz *DomainObject* y *Serializable*, una clase propia de Java. La segunda de estas implementaciones sea realiza para que el objeto sea serializable y por tanto, se pueda persistir en una base de datos sin problema. Es un requisito que queda recogido en la documentación de Spring^[1], cualquier entidad que se vaya a persistir en una base de datos debería implementar esta interfaz.

Esta clase tiene la anotación `@Access(AccessType.PROPERTY)` propia de todas las entidades del modelo de dominio, para indicar que las anotaciones se añaden a los métodos getter en lugar de en los propios atributos.

Respecto a los atributos, se declara *id* y *version*, necesarios para todas las entidades a la hora de persistirlas en la base de datos para poder tener un control de las mismas. El identificador es un valor de tipo entero, que se genera de forma automática y que identifica de forma única a una instancia de una entidad en una tabla de la base de datos. La versión se utiliza principalmente para comprobar que un recurso no haya sido editado de forma concurrente. Además, contiene atributos referentes a la fecha de creación de la entidad, y la fecha en que ha sido actualizada por última vez, puesto que suele ser necesario tener constancia de las mismas y al tenerlos en esta clase genérica de la que heredan todas las entidades, es posible gestionarlas en un servicio genérico.

Se define cuál es el método de igualdad por defecto para una entidad: dos entidades son la misma si coincide su ID.

Por último, se proporcionan métodos que servirán de utilidad en otros lugares de la aplicación para hacer ciertas comprobaciones, como por ejemplo en los repositorios y servicios. Estos métodos son propiedades derivadas, que no se persisten en la base de datos de cada una de las entidades y se calculan en base a otros parámetros cuando sea necesario su uso.

A continuación se explica de forma breve cuál es la función de algunos de esos métodos:

- *retrieveCurrentVersion*: devuelve la entidad en su versión actual. En caso de no estar persistida aún en la base de datos se devuelve la propia instancia de la entidad con la que se llama al método. Si ya se encuentra almacenado en la base de datos, se recupera del repositorio correspondiente la versión que haya almacenada.
- *hasChanged*: devuelve true si la versión de la entidad con la que se llama al método no se corresponde con la versión más actual que se recupera con el método *retrieveCurrentVersion*. False en caso contrario, cuando sí coinciden las versiones. Este método se utiliza para comprobar si han tenido lugar operaciones concurrentes.
- *reconstruct*: este se encarga de devolver la entidad con los atributos, que el usuario ha introducido en un formulario de creación o edición de esta entidad, reconstruidos. Esto quiere decir que en la entidad que se devuelva, en esos atributos que ha introducido en el formulario, se tendrán los valores de la versión actual de dicha entidad. Este método se utiliza con el objetivo de evitar el hacking POST en los formularios. Para saber más acerca del sistema que evita el hacking POST, acuda a la sección 9, donde se trata este tema.

2.3 Entidades de dominio

Para crear entidades de dominio del problema, debemos crear una clase que extienda a *DomainEntity*. Todas las entidades del dominio irán en el paquete *domain*.

Para completar la entidad, hay que declarar los atributos y las relaciones con otras entidades, después generar los métodos de consulta y modificación de estos atributos. Seguidamente, habrá que añadir las etiquetas pertinentes de Javax^{[5][6][7]} e Hibernate^[8], para una correcta configuración, validación de campos y generación de la base de datos. Podemos consultar las anotaciones en su documentación.

La validación de colecciones no está implementada en ninguna de estas bibliotecas, por lo que para poder realizar estas validaciones individuales sobre los elementos de la colección, Sprouts-Framework ha añadido una librería desarrollada por el autor Jakub Jirutka, llamada *validator-collection-2.1.6*^[9]. El funcionamiento de esta librería es muy simple, ya que existe un mapeo entre las etiquetas más habituales de las librerías mencionadas anteriormente y las proporcionadas por esta librería. Es decir, esta librería provee la mayoría de las etiquetas que están en las anteriores librerías, ya que hace uso de ellas, pero aplicadas a colecciones.

La manera de usar estas etiquetas es introducir “Each” delante del nombre de la etiqueta correspondiente.

```

@ManyToMany
@NotNull
@EachNotNull
public Collection<Barter> getRequesteds() {
    return requesteds;
}

```

Para el ejemplo mostrado el funcionamiento de la etiqueta `@EachNotNull` es el siguiente: por cada uno de los elementos de la colección comprobará que no sea nulo.

- i La manera estándar de crear una entidad de dominio se puede encontrar explicado en el material de la asignatura Diseño y Pruebas del departamento de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla.

2.4. Datatypes

Un *datatype* es otro artefacto de modelado. Es un concepto que representa un objeto de valor o un objeto que se embebe. Estos objetos tienen una característica común: no tienen una identidad, ni la necesitan. Los datatypes suelen representar conceptos abstractos como enteros o números reales, cadenas de caracteres, fechas del calendario, cantidades de dinero.

Alternativas en la implementación de Datatypes

En general, hacer uso de un *Datatype* en una entidad, siendo este el tipo de uno de sus atributos, suele generar ciertos problemas a la hora de trabajar con los mismos.

A continuación se exponen las posibles alternativas a la hora de implementar un *Datatype*. El caso que se está contemplando en todos estos ejemplos consiste en una entidad que tendría uno de sus atributos como un *Datatype*. En caso de tener una colección, no debería haber ninguna problemática.

Alternativa 1: implementación como `@Embeddable`

Esta alternativa consiste en que en la clase que modela el *Datatype* en cuestión se tenga la anotación `@Embeddable` además de implementar la interfaz *Datatype*.

```

@Embeddable
@Access(AccessType.PROPERTY)
public class DatatypeClass implements Datatype{

```

Con esta anotación lo que se indica es que al persistir una entidad en la base de datos, en la que uno de los atributos que tenga sea de este tipo, se van a injectar los campos de la entidad con la anotación `@Embeddable` en la tabla de la otra entidad.

Pros:

- Se puede acceder a cualquiera de los campos del objeto de tipo Datatype a través de una consulta, puesto que los atributos forman parte de la tabla de la entidad donde se inyecta.
- Implementación sencilla.

Contras:

- Si este atributo que tiene la entidad en cuestión es *Nullable*, es decir, puede ser nulo, no se podría implementar de esta forma. Esto se debe a que cada campo del Datatype a su vez tiene sus validadores, y si se dejan como nulas estas columnas en la base de datos, saltarán los validadores correspondientes a cada uno de los atributos.
- Si el Datatype tiene muchos atributos, la tabla de la entidad que tiene un atributo del tipo de este Datatype aumentará en gran cantidad su número de columnas, haciendo que traer una entidad desde la base de datos suponga una carga de trabajo mayor.

Alternativa 2: implementación como Serializable

La segunda alternativa sería consistente en que el *Datatype* implemente la interfaz *Serializable*. En este caso se implementa la interfaz *Datatype*, que es una interfaz que a su vez extiende de *Serializable* y *Validable*.

```
@Access(AccessType.PROPERTY)
public class DatatypeClass implements Datatype{
```

De esta forma la entidad de tipo *Datatype* se almacena en la base de datos en la tabla de la otra entidad como un BLOB (*Binary Large Object*). La entidad que tiene un atributo del tipo del *Datatype* tan solo tendría una columna con el BLOB, que no es más que una codificación en binario del objeto en cuestión. A continuación un ejemplo de cómo aparece en la tabla de la base de datos, en la que un *Customer* tiene una *CreditCard* y se hace uso de esta implementación:

	id	createdAt	updatedAt	version	name	surname	userAccount_id	contactPhone	creditCard	customerType	socialIdentity_id
	3	NULL	NULL	0	Carlos	Fernández Pérez	3	666666666666	BLOB	NULL	NULL
	4	NULL	NULL	0	Beatriz	Núñez Jiménez	4	987997948998	BLOB	NULL	1
	5	NULL	NULL	0	Adrián	Fernández	5	9879489489489	BLOB	NULL	2
▶	6	NULL	NULL	0	Antonio	Navarro	6	9845684856878	NULL	NULL	NULL
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Pros:

- El atributo que sea del tipo del *Datatype* puede ser *Nullable*. No habría problemas como en el caso de la primera alternativa.
- Se añade a la tabla de la base de datos tan solo una columna, en la que se almacena la representación binaria del objeto, en lugar de una para cada atributo del *Datatype*.

Contras:

- No se puede acceder a ninguno de los atributos del objeto que sea de tipo *Datatype* a través de una consulta a la base de datos, puesto que la información que se almacena es tan solo una codificación binaria del mismo.

Alternativa 3: implementación como @ElementCollection con solo un elemento

La última de las posibles alternativas a utilizar al persistir un *Datatype* en la base de datos consiste en que al atributo de la entidad a modelar que sea de tipo *Datatype* se le añada la etiqueta `@ElementCollection`. Además, también sería necesario especificar mediante la anotación `@Size(max=1)` que el tamaño máximo de esta “colección” es de 1, en caso de que interesase tener tan solo un *Datatype* asociado a la entidad, como es en el caso de los ejemplos que se están exponiendo.

Mediante este workaround, lo que se realiza es aprovechar la forma en que se persiste en la base de datos, mediante JPA, las colecciones. Se realiza mediante una nueva tabla que contiene cada uno de los elementos de la colección junto con un ID que hace referencia a la entidad a la que pertenece dicho elemento de la colección.

Pros:

- Se puede tener un atributo de un tipo que sea *Datatype* como *Nullable*.
- Los *Datatype* están almacenados en una tabla de la base de datos y por tanto se puede consultar el valor de sus atributos.

Contras:

- Es un workaround, realmente no es una solución limpia tener que persistir un *Datatype* en la base de datos como si fuese una colección de un único elemento.
- Se crea una nueva tabla para cada uno de los *Datatypes* en la base de datos.

2.5. Forms

Un *form* es un objeto que provee exactamente los atributos que se necesitan en un formulario HTML. La idea es que los objetos del modelo del dominio como las entidades o *datatypes* deberían ser fáciles de reconstruir a partir del objeto de tipo *form*.

Situaciones en las que hacer uso de un *form* podría ser útil:

- Para intentar prevenir ciertos ataques de POST hacking. En aquellos formularios en los que se vaya a actualizar o crear una entidad y no sea necesario que el usuario introduzca valores para todos los atributos, se puede crear un objeto de tipo *form* teniendo solamente aquellos que sean imprescindibles. De esta forma se evita que el usuario pueda ver y modificar otros atributos que no debería.
- Cuando se necesiten formularios HTML que no se correspondan directamente con ninguna entidad. Un ejemplo claro es el formulario de registro. En este, se necesita un campo para repetir la contraseña y un checkbox para que el usuario acepte las condiciones. Estos atributos no forman parte de la entidad que representa al usuario

que se ha registrado, pero sin embargo son necesarios mostrarlos. Ahí es donde entra en acción el uso de un *form*.

- Cuando se necesite crear más de una entidad a partir de los datos introducidos en el formulario. Solamente será necesario cuando la entidad que interesa crear no tenga navegabilidad directa hasta la/s otra/s entidad/es a crear. De lo contrario, no sería necesario este *form*.
- Para tratar con los elementos de las listas o colecciones de forma individual. Si se crea un objeto de tipo *form*, es posible realizar los métodos CRUD sobre los elementos de forma individual, sin tener que editar toda la lista o colección al completo.

La implementación de un objeto que represente un *form* es sencilla, prácticamente idéntica a como se realiza para implementar una entidad del dominio. Cualquier clase que vaya a representar a un *form* debe implementar la interfaz *DomainForm*. No es necesario añadir ninguna anotación delante de la declaración de la clase a diferencia de las entidades. En la clase se declaran los atributos correspondientes a los campos que sean necesarios y sus correspondientes métodos getter y setter, operando de igual forma que se hace con las entidades.

2.6. Conclusión

Se conoce ya como construir cualquier tipo de elemento del modelo de dominio, de forma que se pueda reflejar de forma fiel cuál es el dominio del problema. De esta manera, queda lista la base sobre la que se van a cimentar cada uno de los distintos elementos que faltan por construir para completar un sistema de información web.

3. REPOSITORIOS

3.1 Introducción

Para realizar consultas contra la base de datos es fundamental disponer de repositorios que almacenen las consultas que sean necesarias. Por lo tanto vamos a ver cómo funcionan los repositorios.

3.2 Definición de repositorios

En Sprouts-Framework los repositorios son interfaces que pueden extender a dos tipos de interfaces: *PagingAndSortingRepository<Entity, Id>*, aplicable cuando se quiera tener consultas paginadas o *JpaRepository<Entity, Id>* si no se quiere disponer de consultas paginadas, donde *Entity* será la entidad para la que se está creando el repositorio, e *Id* será el tipo de objeto que representa al identificador de la entidad, generalmente un Integer. Estas dos interfaces proveen los métodos CRUD y algunas extensiones. Todos los repositorios deben estar situados en el paquete *repositories*.

Las interfaces contienen signaturas de métodos, que reciben los parámetros que sean necesarios para realizar la consulta. Estas tienen la anotación *@Query*, dentro de esta se define la consulta a realizar.

```
@Repository("BarterRepository")
public interface BarterRepository extends
PagingAndSortingRepository<Barter, Integer>{
    @Query("select (1.0*(select count(b) from Barter b))/(1.0*count(u))
from User u")
    Double averageOfBartersPerUser();
}
```

3.3 Paginación de repositorios

Cuando los resultados que una consulta vaya a devolver son muchos y nos interesa tenerlos paginados, se hace uso de la interfaz *PaginatingAndSorting<Entity,Id>*. En este tipo de repositorios se pueden incluir tanto consultas paginadas, como normales.

Para crear una consulta paginada, la signatura del método debe recibir por parámetro un objeto *Pageable* para que sea posible realizar la consulta de manera paginada. El método debe devolver un objeto del tipo *Page<Entity>*, donde *Entity* es la entidad sobre la que está realizando la consulta paginada. Una consulta paginada sería de esta forma:

```
@Query("select s from SocialIdentity s where s.user.id = ?1")
Page<SocialIdentity> findSocialIdentitiesByUserId(int userId, Pageable page);
```

3.4 Conclusión

Después de este capítulo, usted es capaz de distinguir entre los dos tipos de repositorios que se usan en este framework, y cual es el uso destinado para cada uno, así como utilizarlos.

4. SERVICIOS

4.1 Introducción

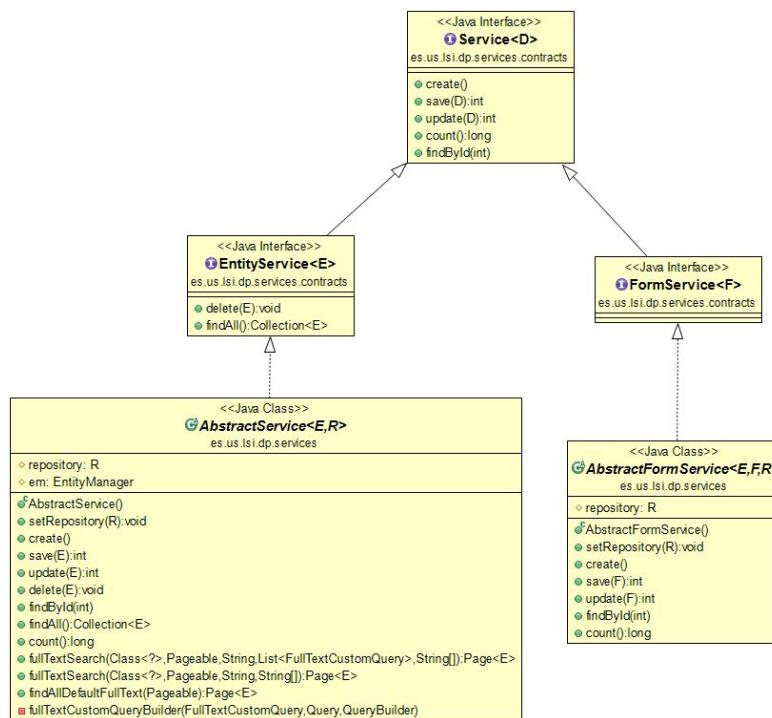
Sprouts Framework da soporte para implementar casos de uso básicos en pocas líneas de código. Además, ofrece la flexibilidad necesaria para que se puedan implementar casos de uso complejos de forma ordenada, evitando que el desarrollador repita código de forma innecesaria, eliminando así gran parte del riesgo de cometer errores.

En el presente apartado se explicará de forma técnica y detallada cómo funcionan los servicios en Sprouts Framework, y qué debe hacer el desarrollador para hacer uso de ellos.

4.2 Estructura de clases

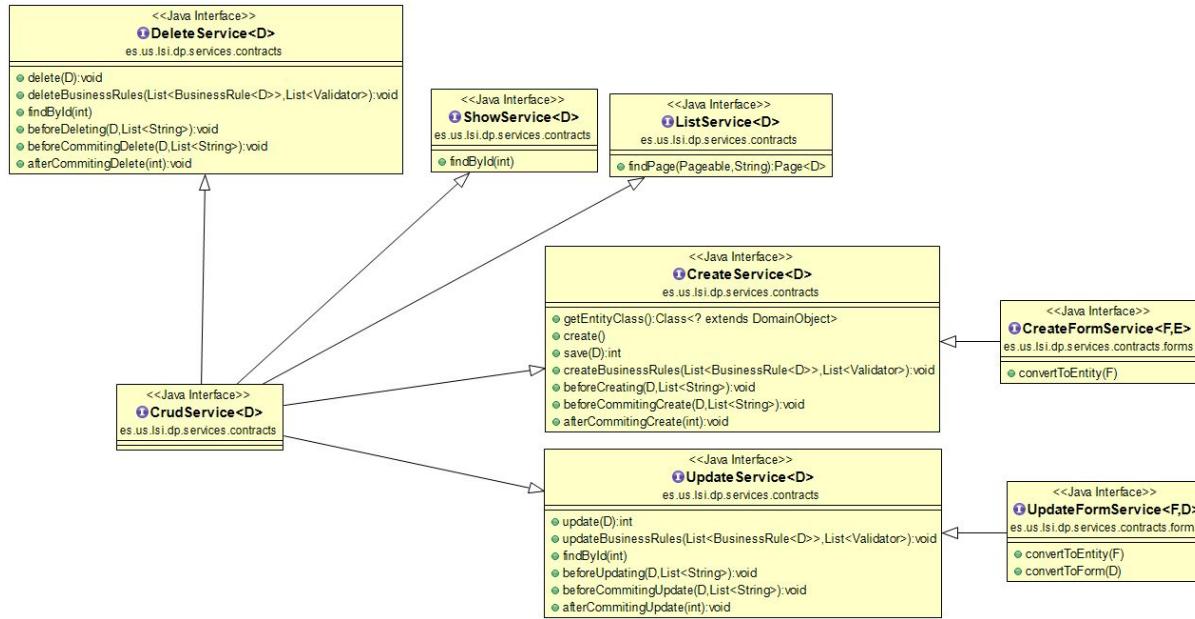
Sprouts Framework permite implementar dos tipos de servicios: uno para trabajar con entidades normales, y otro para trabajar con objetos formulario. Según el servicio que queramos crear, la clase heredará de *AbstractService* o *AbstractFormService*, y según el tipo de operación que este vaya a realizar, podrá implementar de una serie de interfaces u otras.

A continuación exponemos un diagrama de clases UML representando las relaciones entre las clases de servicios.



Service es una clase que contiene todos los métodos comunes a los dos tipos de servicios: el de entidades y el de objetos *form*. De esta interfaz heredan las interfaces *EntityService* y *FormService*. La primera contiene dos métodos: *delete* y *findAll*. La segunda no contiene ningún método, y su única finalidad es clasificar el servicio *AbstractFormService* como servicio de objetos *form*. *EntityService* es implementada por *AbstractService*.

Por otra parte, el desarrollador puede hacer uso de las interfaces para crear servicios que soporten distintos tipos de casos de uso:



Las interfaces *DeleteService*, *ShowService*, *ListService*, *CreateService*, *UpdateService* y *CrudService* pueden ser implementadas por servicios de tipo *AbstractService*, mientras que *CreateFormService* y *UpdateFormService* deben ser implementadas por servicios de tipo *AbstractFormService*.

En los siguientes apartados se va a explicar detalladamente cada clase que debe heredar o implementar nuestros servicios.

4.3 Servicios para entidades: *AbstractService*

Todo servicio referido a una entidad debe heredar de la clase *AbstractService*, especificando como parámetros genéricos la entidad con la que va a trabajar el servicio y el repositorio de dicha entidad. Por ejemplo, la cabecera de la clase *GymService* es la siguiente:

```
public class GymService extends AbstractService<Gym, GymRepository>
```

Podemos acceder al repositorio correspondiente, en todo momento, haciendo uso del atributo `repository` de `AbstractService`. Además, según las operaciones que queramos hacer con el servicio, debemos implementar las interfaces pertinentes.

`AbstractService` provee una serie de métodos destinados a realizar el primer o último paso de cada operación CRUD. Dependiendo del tipo de servicio, el controlador llamará un método u otro, siguiente un flujo distinto según el tipo de caso de uso. Se describen los métodos que se encargan de ejecutar las operaciones CRUD:

- `public E create():` crea una instancia de la entidad con la que el servicio está trabajando.
- `public int save(final E domainObject):` guarda el objeto que recibe como parámetro en el repositorio con el que trabaja el servicio.
- `public int update(final E domainObject):` actualiza el objeto que recibe como parámetro en el repositorio con el que trabaja el servicio.
- `public void delete(final E domainObject):` elimina el objeto que recibe como parámetro usando el repositorio con el que trabaja el servicio.
- `public E findById(final int id):` encuentra una entidad dado el id de esta en el repositorio con el que trabaja el servicio.
- `public Collection<E> findAll():` encuentra todas las entidades persistidas del tipo E, usando el repositorio con el que trabaja el servicio.



`AbstractService` provee también de una serie de métodos para trabajar con búsquedas Full-Text. Estos métodos se explicarán en el apartado Búsquedas Full-Text.

No aconsejamos que el desarrollador redefina estos métodos en la clase que extienda `AbstractService`. Sprouts Framework proporciona métodos que evitan que sea necesario redefinirlos. Sin embargo, si el caso de uso es demasiado complejo, es posible hacerlo. Tenga cuidado de no romper el flujo natural del servicio.

En los siguientes apartados se explicará cómo implementar cada tipo de servicio y cómo trabajan.

4.3.1 CreateService

Este tipo de servicios ofrecen la arquitectura necesaria para crear instancias de una entidad, comprobar reglas de negocio, y persistirlas en la base de datos. Deben ser implementadas por el servicio en el que estemos haciendo uso de la entidad, y reciben como parámetro genérico la entidad con la que vamos a trabajar. Como ejemplo exponemos la cabecera de la clase `GymService` haciendo uso de este servicio:

```
public class GymService extends AbstractService<Gym,  
GymRepository> implements CreateService<Gym>
```

A continuación se describen los métodos de esta interfaz. En primer lugar, presentamos el siguiente par de métodos, que vienen definidos en *AbstractService* y que, por tanto, no es necesaria su redefinición en el servicio que estemos creando.

- *public D create():* crea una instancia de la entidad con la que trabaje el servicio, haciendo uso del método *getEntityClass()* que veremos a continuación.
- *public int save(final D domainObject):* persiste el objeto que recibe como parámetro y devuelve un entero que representa el identificador con el que ha sido persistido. En la definición del método en *AbstractService*, se establece el atributo *createdAt* de la entidad al momento actual.

Los siguientes cinco métodos deben ser implementados y redefinidos en el servicio que hemos creado:

- *public Class<? extends DomainObject> getEntityClass():* es necesario implementar este método. Debe devolver la clase de la entidad con la que trabaja el servicio.

```
@Override  
public Class<? extends DomainEntity> getEntityClass() {  
    return Folder.class;  
}
```

- *public void createBusinessRules(final List<BusinessRule<D>> rules, final List<Validator> validators):* permite añadir reglas de negocio o validadores que se aplicarán a la entidad creada a partir de los datos que el usuario introduce en el formulario. Las reglas de negocio se declaran en el servicio usando la etiqueta *@Autowired*. En el apartado Validadores y reglas de negocio se explica este aspecto con detalle.

```
@Autowired  
private IsAuthorisedToManageFolder authorisedToManageFolder;  
  
@Override  
public void createBusinessRules(List<BusinessRule<Folder>> rules, List<Validator> validators) {  
    rules.add(authorisedToManageFolder);  
}
```

- *public void beforeCreating(final D validable, List<String> context):* recibe la entidad con la que estemos trabajando y un contexto. Explicaremos esto último más adelante. Este método será llamado dos veces: antes de mostrarle el formulario al usuario y cuando este envíe el formulario. Gracias a este método podremos mostrar en el formulario campos con valores por defecto (por ejemplo, una fecha de creación), y asegurarnos de que el usuario no varía su valor.

```

@Override
public void beforeCreating(Match validable, List<String> context) {
    Barter requested;
    int barterId;

    if (!context.isEmpty()) {
        barterId = new Integer(context.get(0));
        requested = barterService.findById(barterId);
        validable.setRequested(requested);
    }

    validable.setMoment(Moment.now());
    validable.setCancelled(false);
}

```

En este ejemplo se establecen los atributos *requested*, *moment* y *cancelled* del objeto *Match validable*. El atributo *requested*, se establece si el contexto que espera recibir el identificador de un objeto de tipo *Barter*, no está vacío. Al atributo *moment* se le aplica la fecha actual, y a *cancelled* el valor *false*. Recordamos que el método es llamado antes de enviar el formulario al usuario y cuando este hace submit, por lo que si el usuario trata de modificar alguno de estos atributos, esta quedará sin efecto, además de la propia protección que se proporciona para evitar el hacking.

- *public void beforeCommitingCreate(final D validable, List<String> context)*: es parecido al método *beforeCreating*, pero únicamente es llamado cuando el usuario envía el formulario, y antes de comprobar las reglas de negocio y validadores. Un ejemplo de uso de este método puede ser calcular algún atributo de la entidad a partir de algún otro atributo que el usuario haya puesto en el formulario.

```

@Override
public void beforeCommitingCreate(FeePayment validable, List<String> context) {
    Assert.notNull(validable);

    validable.setPaymentMoment(Moment.now());

    Date inactivationDay;
    inactivationDay = new Date(validable.getActivationDay().getTime() + 2592000000L);

    validable.setInactivationDay(inactivationDay);
}

```

En este ejemplo, se usa el método *beforeCommitingCreate* para establecer el atributo *paymentMoment* al momento actual, y para calcular el atributo *inactivationDay* a partir de la fecha de activación que el usuario ha escrito en el formulario.

- *public void afterCommitingCreate(final int id)*: es llamado cuando se persiste con éxito la entidad en la base de datos. El parámetro que recibe es el identificador de la entidad persistida. Es útil para modificar otras entidades una vez que el objeto ha sido persistido.

```

@Override
public void afterCommittingCreate(int id) {

    Gym aux;
    aux = repository.findOne(id);

    ServiceEntity serviceEntity;
    serviceEntity = serviceEntityService.findFitnessServiceEntity();

    ServiceOfGym serviceOfGym;
    serviceOfGym = serviceOfGymService.create();
    serviceOfGym.setCustomersTotalNumber(0);
    serviceOfGym
        .setDescription("Actividad física de movimientos repetidos que se planifica " +
                      "y se sigue regularmente con el propósito de mejorar o mantener el " +
                      "cuerpo en buenas condiciones.");
    serviceOfGym.setPictures(new ArrayList<String>());
    serviceOfGym.setGym(aux);
    serviceOfGym.setServiceEntity(serviceEntity);

    serviceOfGymService.save(serviceOfGym);

}

```

En el contexto del ejemplo que acabamos de presentar, todo objeto *Gym* debe estar relacionado con una entidad *ServiceOfGym*, que debe tener unos atributos por defecto, entre ellos un nombre y una descripción. El problema es que la navegabilidad es de *ServiceOfGym* a *Gym*, por lo que para relacionarlos, hay que establecer el atributo *gym* de *ServiceOfGym*. En este método se puede realizar esta operación, creando un nuevo *ServiceOfGym* y estableciendo como atributo *gym* el id del gimnasio recién creado.

4.3.2 UpdateService

Debemos implementar la interfaz *UpdateService* cuando queramos que el servicio sea capaz de llevar a cabo operaciones de actualización de entidades. Esta interfaz recibe como parámetro genérico la entidad con la que trabaja el servicio:

```

public class GymService extends AbstractService<Gym,
GymRepository> implements UpdateService<Gym>

```

A continuación se describen los métodos de esta interfaz. Los siguientes dos métodos vienen definidos por defecto en *AbstractService*, y no es necesaria su redefinición en el servicio que estemos creando:

- *public int update(D domainObject)*: llama al método *save* del repositorio. Sirve para actualizar una entidad, estableciendo el atributo *updatedAt* de la entidad a la fecha actual. Devuelve el identificador del objeto persistido.
- *public D findById(int id)*: encuentra una entidad dado el identificador de ésta.

Los siguientes cuatro métodos deben ser implementados y redefinidos en el servicio que hemos creado:

- `public void updateBusinessRules(List<BusinessRule<D>> rules, List<Validator> validators)`: permite añadir reglas de negocio o validadores que se aplicarán a la entidad actualizada con los datos del usuario. El uso del método es idéntico al del método `createBusinessRules` explicado en el apartado anterior.
- `public void beforeUpdating(final D validable, List<String> context)`: recibe el objeto con el que estamos trabajando y el contexto actual. El método es llamado antes de enviar el formulario al usuario y cuando éste lo envía. Con este método podremos mostrarle al usuario valores por defecto en el formulario, sin que el usuario pueda modificarlos.
- `public void beforeCommittingUpdate(final D validable, List<String> context)`: recibe el objeto reconstruido a partir de los datos que el usuario ha introducido en el formulario y el contexto. Es llamado cuando el usuario envía el formulario y antes de comprobar las reglas de negocio y validadores.

```
@Override
public void beforeCommittingUpdate(Barter validable, List<String> context) {
    validable.setCancelled(true);
}
```

El contexto de este método es un caso de uso en el que el usuario debe poder cancelar un *Barter*. Para ello, se usa un *UpdateService*. Cuando el usuario envía el formulario, este método cancela el *Barter*.

- `public void afterCommittingUpdate(final int id)`: este método se llama una vez que la actualización se ha llevado a cabo con éxito. Recibe el identificador de la entidad persistida.

```
@Override
public void afterCommittingUpdate(int id) {
    Match match;
    match = matchService.findMatchByBarterId(id);
    if (match != null) {
        match.setCancelled(true);
        matchService.update(match);
    }
}
```

Este método pertenece al caso de uso que se ha expuesto como ejemplo del método `beforeCommittingUpdate`. Cuando se cancela un *Barter*, también debe ser cancelado el *Match* en el que esté involucrado, si es que estaba involucrado en alguno. Este método encuentra el *Match* en el que está involucrado este *Barter* y lo cancela.

4.3.3 DeleteService

En el caso de que nuestro servicio tenga que eliminar algún objeto persistido, implementaremos la interfaz *DeleteService*, indicando la entidad con la que estemos trabajando de la siguiente manera:

```
public class GymService extends AbstractService<Gym,  
GymRepository> implements DeleteService<Gym>
```

A continuación se describen los métodos de esta interfaz. Los siguientes dos métodos vienen definidos por defecto en *AbstractService*, y no es necesaria su redefinición en el servicio que estemos creando:

- *public int delete(D domainObject)*: llama al método *delete* del repositorio. Sirve para eliminar la entidad que recibe como parámetro.
- *public D findById(int id)*: encuentra una entidad dado el identificador de esta.

Los siguientes cuatro métodos deben ser implementados y redefinidos en el servicio que hemos creado:

- *public void deleteBusinessRules(List<BusinessRule<D>> rules, List<Validator> validators)*: permite añadir reglas de negocio que se aplicarán antes de ejecutar la eliminación del objeto con el que estemos trabajando.
- *public void beforeDeleting(final D validable, List<String> context)*: recibe la entidad que va a ser eliminada.
- *public void beforeCommittingDelete(final D validable, List<String> context)*: recibe la entidad que va a ser eliminada y el contexto actual. Se le llama cuando el usuario envía el formulario confirmando la eliminación y antes de comprobar las reglas de negocio.

```
@Override  
public void beforeCommittingDelete(ServiceOfGym validable, List<String> context) {  
    Collection<Booking> bookings;  
  
    // Se eliminan las reservas que estuvieran asociadas al servicio que se  
    // va a borrar.  
    // También se eliminan los comentarios asociados al servicio a borrar.  
  
    bookings = bookingService.findBookingsByServiceOfGymId(validable.getId());  
    Assert.notNull(bookings);  
  
    bookingService.deleteAllReferredToServiceOfGym(bookings);  
    commentService.deleteAllReferredToServiceOfGym(validable);  
}
```

En este ejemplo, se elimina todo lo relacionado con un *ServiceOfGym* antes de proceder a su eliminación.

```
public class GymService extends AbstractService<Gym,  
GymRepository> implements CreateService<Gym>
```

- i** Los métodos *beforeCommitting* y *afterCommitting* de todas los tipos de servicios se ejecutan en una misma transacción.

- *public void afterCommittingDelete(final int id)*: es llamado una vez que el objeto ha sido eliminado con éxito. Recibe el identificador del objeto eliminado. Con este método podremos hacer operaciones una vez que el objeto haya sido eliminado.

4.3.4 ListService

Cuando necesitemos un listado de entidades, debemos usar un servicio que implemente la interfaz *ListService*, pasándole como parámetro genérico el tipo de entidad con la que estemos trabajando:

```
public class GymService extends AbstractService<Gym,  
GymRepository> implements ListService<Gym>
```

Al implementar esta interfaz, tendremos que implementar el siguiente método:

- *public Page<D> findPage(Pageable page, String searchCriteria)*: recibe un objeto de tipo *Pageable* con información sobre la paginación, y una cadena que representa un criterio de búsqueda. Devuelve un objeto de tipo *Page<D>*, siendo *D* la entidad que vamos a listar. Un ejemplo de uso es el siguiente:

```
@Override  
public Page<ServiceOfGym> findPage(Pageable page, String searchCriteria) {  
    Page<ServiceOfGym> result;  
  
    result = repository.findAll(page);  
    Assert.notNull(result);  
  
    return result;  
}
```

Estamos listando todos los objetos de tipo *ServiceOfGym* que se encuentran en la base de datos.

i Los servicios de Sprouts Framework también soportan búsquedas full-text. Para saber cómo se implementan, consulte el apartado Búsquedas Full-Text.

4.3.5 ShowService

La interfaz *ShowService* deberá ser implementada por el servicio en el que tengamos un caso de uso de mostrar información de una entidad.

```
public class GymService extends AbstractService<Gym,  
GymRepository> implements ShowService<Gym>
```

La interfaz *ShowService* tiene declarado el siguiente método:

- *public D findById(int id)*: encuentra un objeto del tipo con el que trabaje el servicio dado un id. Este método viene definido por defecto en *AbstractService*, llamando al método

findOne del repositorio con el que trabaje el servicio. Es posible y aconsejable redefinir este método en el servicio que estemos creando.

4.3.6 CrudService

Si queremos que nuestro servicio implemente las operaciones de creación, actualización, borrado, listado y mostrado de entidades, debemos implementar esta interfaz, que extiende a todos los tipos de servicios que se ha visto anteriormente.

```
public class GymService extends AbstractService<Gym,  
GymRepository> implements CrudService<Gym>
```

4.4 Servicios para objetos de tipo Form. AbstractFormService

Sprouts Framework permite trabajar de forma cómoda con objetos de tipo *Form*. De este tipo de objetos se habló en el apartado 2.5. Mediante los *forms* podemos realizar dos operaciones: crear o actualizar entidades. Para ello, en primer lugar necesitamos que nuestro servicio extienda la clase *AbstractFormService*, la cual, al implementar la interfaz *FormService*, implementa los métodos necesarios para crear y persistir una entidad a partir de un *form*:

- *public F create()*: crea una instancia del objeto *form* sobre el que estemos trabajando.
- *public int save(final F form)*: guarda el objeto *form* que recibe como parámetro, convirtiéndolo previamente a entidad.
- *public int update(final F form)*: actualiza el objeto *form* que recibe como parámetro, convirtiéndolo previamente a entidad.
- *public F findById(final int id)*: devuelve el objeto *form* dado el identificador de una entidad. Para ello, convierte la entidad cuyo identificador coincide con el recibido como parámetro a un objeto *form*.

La principal peculiaridad de los servicios que operan sobre objetos *form* es la necesidad de convertir una entidad a *form* o viceversa, según estemos trabajando con un *CreateFormService* o un *UpdateFormService*. En los próximos apartados explicaremos ambos tipos de servicios.

Al extender la clase *AbstractFormService*, se deben proporcionar los siguientes parámetros genéricos: la entidad, el *form* y el repositorio que opere con la entidad. El siguiente ejemplo ilustra la cabecera de una clase que representa el servicio que trabaja con objetos *form* para *Booking*:

```
public class BookingFormService extends  
AbstractFormService<Booking, BookingForm, BookingRepository>
```

4.4.1 CreateFormService

CreateFormService es una interfaz que debe ser implementada por aquellos servicios en los que necesitemos crear una entidad a partir de un objeto *form*. Recibe como parámetros genéricos el objeto *form* y la entidad con los que trabaja el servicio.

```
public class BookingFormService extends  
AbstractFormService<Booking, BookingForm, BookingRepository>  
implements CreateFormService<BookingForm, Booking>
```

Este servicio crea instancias de una entidad a partir de su representación en un objeto tipo *form*. Se presentará al usuario un formulario en el que debe completar los atributos del objeto *form*, y cuando este lo envíe, el servicio lo convertirá a entidad y lo persistirá.

La interfaz *CreateFormService* extiende a *CreateService*, por tanto, será necesario redefinir los métodos que se explicaron en el apartado 4.3.1 *CreateService*. Las principales diferencias son que el método *getEntityClass* debe devolver la clase del objeto *form*, y los métodos *beforeCreating* y *beforeCommittingCreate* reciben el objeto *form*.

Además de estos métodos, debemos implementar el método *convertToEntity*, que recibe un objeto *form* y devuelve una entidad. Aquí debemos especificar cómo convertir el formulario a la entidad que representa. Por ejemplo, el siguiente método convierte un formulario de *Booking* en una entidad de dicho tipo. Para ello, establece cada uno de los atributos de *Booking* según lo que el usuario escribió en el formulario, y que está recogido por el objeto *form*:

```

@Override
public Booking convertToEntity(BookingForm bookingForm) {
    Assert.notNull(bookingForm);

    Booking result;
    ServiceOfGym serviceOfGym;
    Date endMoment;
    Date date;
    Customer customer;

    date = Moment.now();
    Assert.notNull(date);

    customer = customerService.findByPrincipal();
    Assert.notNull(customer);

    // Se recupera el servicio del gimnasio que se ha reservado a partir de su id.
    serviceOfGym = serviceOfGymService.findById(bookingForm.getServiceOfGymId());
    Assert.notNull(serviceOfGym);

    // Se calcula el atributo derivado fecha de finalización sumando a
    // la fecha solicitada la duración de la reserva.
    double durationDouble = bookingForm.getDuration() * 3600000.0;
    endMoment = new Date((bookingForm.getRequestedMoment().getTime() + (long) durationDouble));
    // Se crea un Booking y se le asignan los valores introducidos por
    // el usuario en el formulario y los que se acaban de calcular.
    result = new Booking();
    result.setCreationMoment(date);
    result.setCustomer(customer);
    result.setRequestedMoment(bookingForm.getRequestedMoment());
    result.setDuration(bookingForm.getDuration());
    result.setServiceOfGym(serviceOfGym);
    result.setEndMoment(endMoment);

    return result;
}

```

4.4.2 UpdateFormService

Para aquellos servicios en los que necesitemos usar un objeto *form* para actualizar una entidad, implementaremos la interfaz *UpdateFormService*. Recibe como parámetros genéricos el objeto *form* y la entidad con los que trabaja el servicio.

```

public class UserAccountFormService extends
AbstractFormService<UserAccount,
UserAccountForm,UserAccountRepository> implements
UpdateFormService<UserAccountForm, UserAccount>

```

Para llevar a cabo esta operación con un objeto *form*, será necesario convertir la entidad a actualizar en *form*, y cuando el usuario envíe el formulario, convertiremos el *form* en entidad y la persistiremos.

La interfaz *UpdateFormService* extiende a *UpdateService*, por tanto, será necesario redefinir los métodos que se explicaron en el apartado 4.3.2 *UpdateService*. Las principales diferencias son que los métodos *beforeUpdating* y *beforeCommittingUpdate* reciben el objeto *form*.

Además de estos métodos, debemos implementar `convertToForm`, que recibe una entidad y devuelve un objeto de tipo `form`, y `convertToEntity`, que recibe un objeto `form` y devuelve una entidad. En el primero, convertiremos una entidad en un objeto `form`, y en el segundo, convertiremos el objeto `form` (con los datos introducidos por el usuario) en entidad.

A modo de ilustración mostramos los métodos `convertToForm` y `convertToEntity` del servicio `UserAccountChangePasswordService`, el cual permite la modificación de la contraseña de la cuenta de usuario.

```
@Override  
public UserAccountForm convertToForm(UserAccount entity) {  
    UserAccountForm result;  
    result = new UserAccountForm();  
    result.setUsername(entity.getUsername());  
    return result;  
}
```

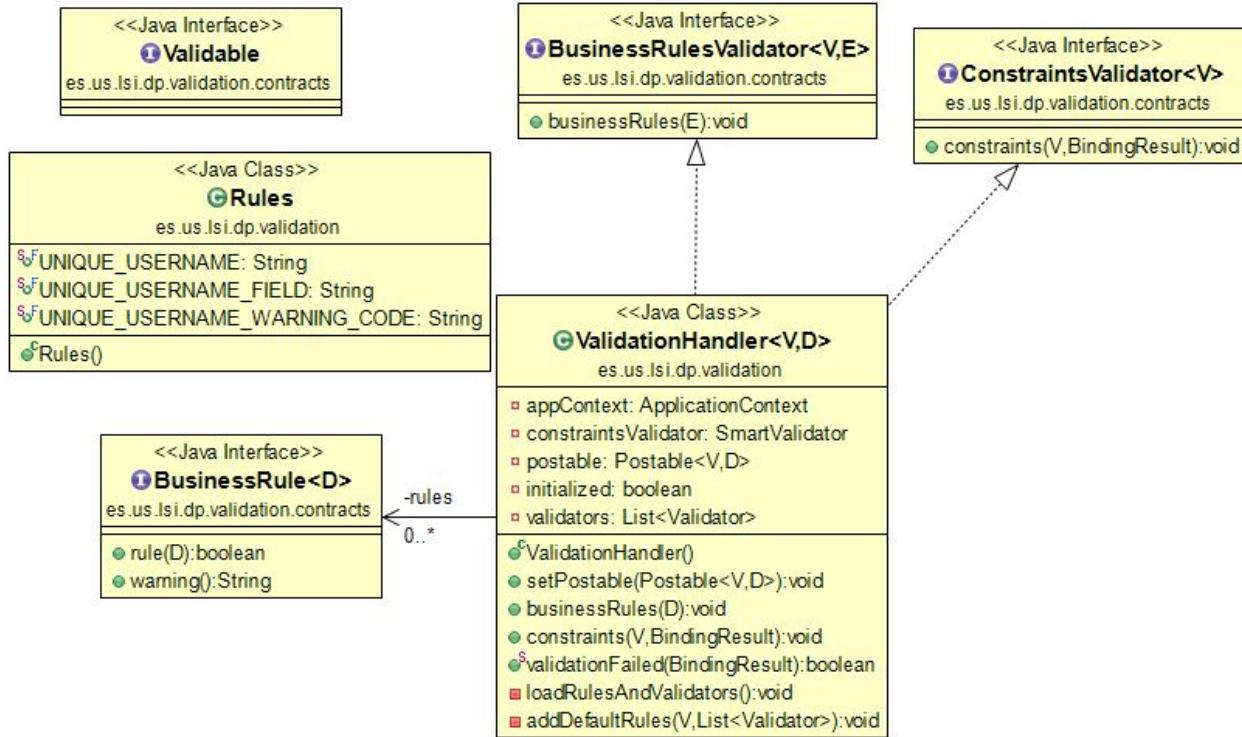
Como podemos ver, en el método `convertToForm` creamos el objeto `form` y establecemos los atributos que se mostrarán al usuario por defecto en el formulario, obteniéndolos a partir de la entidad con la que vayamos a tratar.

```
@Override  
public UserAccount convertToEntity(UserAccountForm form) {  
    Assert.notNull(form);  
  
    UserAccount userAccount;  
    String hashedPassword;  
  
    userAccount = SignInService.getPrincipal();  
    Assert.notNull(userAccount);  
  
    // Hasdeo de la contraseña  
    hashedPassword = PasswordEncoder.encode(form.getPassword());  
    Assert.notNull(hashedPassword);  
  
    userAccount.setUsername(form.getUsername());  
    userAccount.setPassword(hashedPassword);  
  
    return userAccount;  
}
```

Aquí se actualizan los atributos aportados por el usuario, en este caso la contraseña y el nombre de usuario.

- i Para hacer uso del método `findById` definido por defecto en el `AbstractFormService`, es necesario que el servicio implemente la interfaz `UpdateFormService`. El método, a partir de un identificador, crea el objeto form llamando al método `convertToForm`. Si en el controlador queremos usar otro método para obtener la entidad a actualizar, debemos llamar a `convertToForm` para convertir la entidad a objeto form.

4.5 Validadores y reglas de negocio



En este framework la validación se divide en dos partes: validación de formularios y comprobación de reglas de negocio.

Para implementar estas dos validaciones, se dispone de la clase `ValidationHandler`, en el paquete `es.us.lsi.dp.validation`. Esta clase hace uso de varias interfaces que vamos a explicar:

- *BusinessRule<D extends Valifiable>*: esta interfaz especifica los métodos que se tienen que implementar cuando se cree una regla de negocio. Dispone de dos cabeceras de métodos.
 - o *boolean rule(D domainObject)*: recibe como parámetro un objeto que debe ser del tipo del que se compone `BusinessRule`. Con el objeto recibido por parámetro, que es el objeto sobre el que se va a comprobar la regla de negocio, se debe proporcionar una expresión booleana que certifique que se cumple la regla.
 - o *String warning()*: es un método que debe devolver el código del mensaje de error que se haya especificado en los `messages.properties`.
- *BusinessRulesValidator<V extends Valifiable, E extends Valifiable>*: es la interfaz que debe implementar la clase `ValidationHandler` para indicarle que va a realizar labores de comprobación de reglas de negocio.

- o *businessRules(E domainEntity)*: recibe una entidad de dominio por parámetro sobre la que se quiere realizar la comprobación de las reglas de negocio. Este método se debe encargar de comprobar todas las reglas asociadas a ese objeto.
- *ConstraintsValidator<V extends Validable>*: es la interfaz que debe implementar la clase *ValidationHandler* para indicarle que va a realizar labores de validación.
 - o *void constraints(V validable, BindingResult bindingResult)*: recibe el objeto a validar, ya sea un formulario o una entidad de dominio y un objeto *BindingResult* para depositar los errores en caso de que falle la validación. Este método debe encargarse de validar el objeto.
- *Validable*: interfaz de la que deben extender todos los objetos que se vayan a usar en la clase *ValidationHandler*.

Una vez vistas las interfaces de las que hace uso *ValidationHandler*, vamos a pasar a explicar cómo realiza la validación y la comprobación de las reglas de negocio.

Esta clase recibe como parámetros, dos objetos que extienden de *Validable* de los que se hará uso dentro de ella. Además, debe implementar las dos interfaces mencionadas anteriormente para realizar validaciones y comprobación de reglas de negocio: *ConstraintsValidator<V>* y *BusinessRulesValidator<V, D>*.

Vamos a explicar todos los métodos que contiene esta clase:

- *setPostable(final Postable<V, D> postable)*: este método se encarga de inicializar el atributo *postable*, que será el controlador desde el que se llama a esta clase.
- *constraints(final V domainObject, finalBindingResult)*: Este método se llamará desde el *postController* para comprobar las validaciones.

Se encargará, sino están ya inicializadas las listas de reglas de negocio y validadores, de llamar al método *loadRulesAndValidators* para inicializarlas.

Seguidamente, llama al método *addDefaultRules* para añadir los validadores por defecto que se incorporan en este paquete.

Después se hace la validación propiamente dicha, haciendo uso de una librería de Spring.

- *validationFailed(final BindingResult bindingResult)*: Es un método para poder saber en el *postController* si ha habido errores y procesarlos en ese controlador.
- *loadRulesAndValidators()*: Se llama desde el método *constraints* para inicializar las reglas de negocio y los validadores con listas vacías. Seguidamente, se llama al método *businessRules* de *PostController*, haciendo uso del atributo *postable*. De esta manera, este controlador ya tendría las reglas de negocio y validadores inicializados.

Pero esta es una primera inicialización, cuando se vayan a comprobar las reglas de negocio o validación desde el controlador que vaya a realizar esto, por ejemplo *AbstractCreateController*, se modifican las reglas de negocio y validadores, haciendo

uso de los métodos adecuados en los servicios, ya que se sobrescribe el método `businessRules(final List<BusinessRule<D>> rules, final List<Validator> validators)`. De esta manera, los validadores y reglas de negocio, ya no estarían vacíos, sino que contendrían las reglas y validadores especificadas en los servicios correspondientes.

- `addDefaultRules(addDefaultRules(final V domainObject, final List<Validator> validators))`: Este método añade unos validadores por defecto, que se utilizarán en el registro de un usuario. Estos validadores están definidos en el paquete `es.us.lsi.dp.validation.validators`. Para añadirlos desde ese paquete, se hace uso del objeto `AppContext`. Con este objeto, se buscan los validadores anotados con los códigos que contiene la clase `Rules`.
- `businessRules(final D domainObject)`: Este método recibe un objeto sobre el que habrá que comprobar las reglas de negocio. Para ello, hay que comprobar que el objeto supere con éxito todas y cada una de las reglas de negocio almacenadas en la lista `rules`.

4.5.1 Creación de reglas de negocio

Para crear una regla de negocio, debemos crear una clase que implemente la interfaz `BusinessRule<D extends Validable>`, recibiendo como parámetro genérico el tipo del objeto sobre el que se comprobará la regla de negocio. Al hacer esto, el desarrollador tendrá que redefinir los métodos `rule` y `warning`. El primero recibe el objeto a validar. Devolverá `true` si se cumple la regla de negocio, y `false` si no se cumple. El segundo método no recibe nada, y debe devolver una cadena que representa el código de internacionalización declarado en un fichero `message.properties`, conteniendo el mensaje a mostrar en caso de que no se cumpla la regla de negocio.

i La clase que implementa la comprobación de una regla de negocio debe llevar la etiqueta `@Component`. De esta manera, podremos acceder a la regla de negocio desde, por ejemplo, un servicio, por medio de la etiqueta `@Autowired`.

Sirva como ejemplo la clase `IsAuthorisedToManageFolder`, del proyecto Acme-Barter, en el cual se comprueba si un actor tiene permiso para acceder a una determinada carpeta.

```

@Component
public class IsAuthorisedToManageFolder implements BusinessRule<Folder> {
    @Autowired
    private ActorService actorService;

    @Override
    public boolean rule(final Folder folder) {

        Actor actor;
        UserAccount userAccount;
        userAccount = SignInService.getPrincipal();
        actor = actorService.findActorByUserAccount(userAccount.getUsername());
        return actor.getId() == folder.getActor().getId();
    }

    @Override
    public String warning() {
        return "folder.isAuthorised.error";
    }
}

```

4.5.2 Creación de validadores

Los validadores fuerzan a que se cumpla que un campo de un formulario respete unas determinadas restricciones en cuanto a su valor, o que tenga un valor correcto respecto a otro campo. Un ejemplo de validador sería la clase encargada de comprobar que, al registrar un usuario, los dos campos “nueva contraseña” coinciden.

En Sprouts Framework podemos crear validadores implementando la interfaz *Validation* de Spring. Al hacer esto, tendremos que redefinir dos métodos: *supports*, que recibe una clase y devuelve *true* si esa clase es instancia de una clase en concreto, y *validate*, que recibe un objeto y un objeto de tipo *Errors*. Este método debe, en primer lugar, hacer un cast al tipo de objeto que vayamos a validar, y luego comprobar que se cumple la condición de validación. Si no se cumple, usaremos el método *rejectValue* del objeto de tipo *Errors* recibido como parámetro, pasando el nombre del campo que produjo el error, y un código de internacionalización, conteniendo el mensaje a mostrar en caso de que no se cumpla la validación.

- i La clase que implementa la comprobación de una regla de negocio debe llevar la etiqueta *@Component*. De esta manera, podremos acceder a la regla de negocio desde, por ejemplo, un servicio, por medio de la etiqueta *@Autowired*.

Sirva como ejemplo la clase *OldPasswordValidator*, del proyecto Acme-Barter:

```

@Component
public class OldPasswordValidator implements Validator {

    @Autowired
    private UserService userService;

    @Override
    public boolean supports(Class<?> clazz) {
        return UserAccountForm.class.isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {
        UserAccountForm uaf = (UserAccountForm) target;

        String oldPasswordHashed = PasswordEncoder.encode(uaf.getOldPassword());

        if (!oldPasswordHashed.equals(userService.findByPrincipal().getUserAccount().getPassword()))
            errors.rejectValue("oldPassword", "acme.validators.old-password");
    }
}

```

4.6 Ejecución de procedimientos almacenados desde servicios

En este apartado, explicaremos cómo ejecutar desde un servicio de la aplicación un procedimiento almacenado en una base de datos MySQL. Necesitaremos crear el procedimiento, dar permisos al usuario de la base de datos que usa la aplicación web para acceder a este, y finalmente crear un método en el servicio que llame a dicho procedimiento.

Supongamos que tenemos el siguiente procedimiento MySQL definido:

```

DROP PROCEDURE IF EXISTS `acme-barter`.cancelBarters;
DELIMITER //
CREATE PROCEDURE `acme-barter`.cancelBarters()

BEGIN
update `acme-barter`.barter b2 set b2.cancelled = true where b2.id in(
select id from (
    SELECT b.*, m.offerSignedDate, m.requestSignedDate
    FROM `acme-barter`.barter b
    left JOIN `acme-barter`._match m on (m.offered_id = b.id or
m.requested_id = b.id)
    WHERE b.moment <= DATE_ADD(NOW(), INTERVAL -1 MONTH)
) j where j.offerSignedDate is null or j.requestSignedDate is null);
END //
DELIMITER ;

```

Este procedimiento marca como cancelados aquellos Barters que no han sido firmados un mes después de su creación. Lo primero que debemos hacer es dar permiso al usuario acme-user de la base de datos para que pueda ejecutar el procedimiento. Para ello en la Workbench de MySQL ejecutamos lo siguiente:

```
GRANT EXECUTE ON PROCEDURE `acme-barter`.cancelBarters TO 'acme-user'@'%';
```

A continuación, ejecutamos el script anterior, que otorgará los permisos necesarios al usuario correspondiente para ejecutar el procedimiento en la base de datos Acme-Barter.

- i** Cada vez que poblemos la base de datos, será necesario crear de nuevo el procedimiento. (Posible trabajo futuro: que la utilidad PopulateDatabase.java introduzca también procedimientos).

Para llamar al procedimiento desde la aplicación, escribimos el siguiente método en el servicio correspondiente (en nuestro caso, *BarterService*):

```
public void cancelBarters() {  
    EntityManager entityManager =  
        entityManagerFactory.createEntityManager();  
    Query storedProcedure = entityManager.createNativeQuery("CALL  
    `acme-barter`.cancelBarters()");  
    entityManager.joinTransaction();  
    storedProcedure.executeUpdate();  
}
```

Previamente, hemos declarado objeto *EntityManagerFactory* como *@Autowired*:

```
@Autowired  
private EntityManagerFactory entityManagerFactory;
```

A partir del *EntityManagerFactory*, creamos el *EntityManager*, a partir del cual podremos llamar al método *createNativeQuery*, que recibe como parámetro la sentencia SQL que llama al procedimiento que hemos almacenado en la base de datos. Por último, llamamos a *joinTransaction()* para que el procedimiento se ejecute como una transacción, y al método *executeUpdate()*, para que finalmente ejecute el procedimiento.

4.7 Conclusión

Gracias a las clases que provee Sprouts Framework, crear servicios es una tarea muy sencilla si nuestros casos de uso son simples, y si son complejos, le quita la responsabilidad al desarrollador de tener que preocuparse por una compleja estructura de métodos, ya que Sprouts dota de varios métodos que ayudan a organizar y evitar la repetición de código.

Tanto para trabajar con entidades como con objetos de tipo *form*, el programador ya no se debe preocupar de las tareas más elementales y repetitivas del proceso de creación de

servicios. Sprouts intenta cubrir la mayor parte de casos de uso básicos, ofreciendo al programador una gran flexibilidad.

5 CONTROLADORES

5.1 Introducción

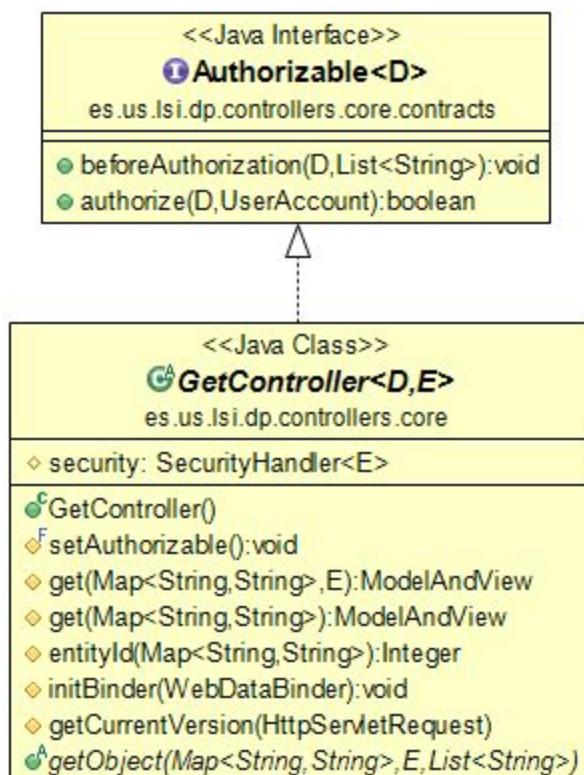
Los controladores son los encargados de coordinar todas las peticiones del usuario. Como con los servicios, Sprouts Framework intenta cubrir todos los posibles casos de uso básicos, ofreciéndole al programador una gran flexibilidad.

En este capítulo se documentan todos los tipos de controladores que ofrece Sprouts Framework. Vamos a explicar, en primer lugar, los controladores del núcleo del framework. Son aquellos que hacen funcionar a los controladores de cada caso de uso específico, que veremos en el apartado 5.3.

5.2 Controladores del núcleo

5.2.1 Authorizable

La interfaz *Authorizable* permite realizar tareas de protección contra el post-hacking y control de permisos. Es implementada por *GetController*, lo que permite que este último sea tratado como un objeto *Authorizable*.



Como podemos ver, declara dos métodos: *beforeAuthorization* y *authorize*. El método *beforeAuthorization* es implementado por las clases *AbstractCreateController*, *AbstractUpdateController*, *AbstractDeleteController* y *AbstractPostController*. En la implementación de los tres primeros, llama a los métodos *beforeCreating*, *beforeUpdating* o *beforeDeleting*, respectivamente. La implementación de *beforeAuthorization* en *AbstractPostController* es un método vacío.

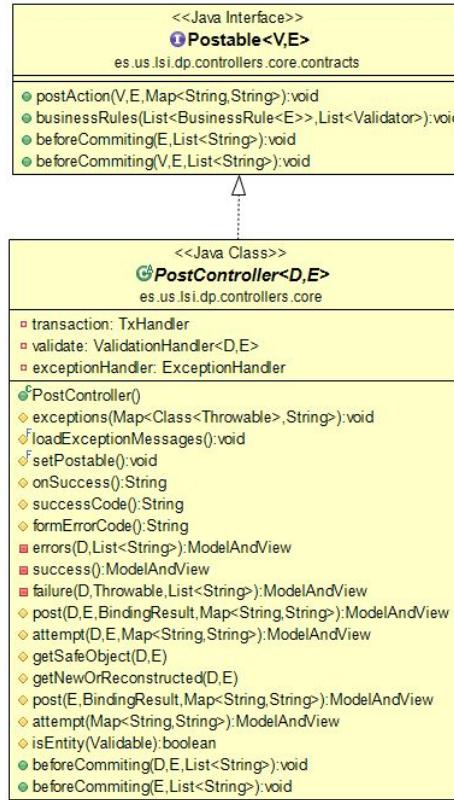
Por otra parte, el método *authorize* se debe implementar en los controladores que el desarrollador crea y hereda de alguna de estas clases: *AbstractCreateController*, *AbstractUpdateController*, *AbstractDeleteController* y *AbstractShowController*. En *AbstractPostController* está definido por defecto, y devuelve siempre *true*.

```
public void beforeAuthorization(final D object, List<String> context);
public boolean authorize(final D domainObject, final UserAccount principal);
```

El método *beforeAuthorization* es llamado siempre al obtener la vista de un formulario de edición, y al enviar dicho formulario. Por otra parte, el método *authorize* se usa dentro del *SecurityHandler*, y será explicado en el capítulo Seguridad y prevención de hacking.

5.2.2 Postable

La interfaz *Postable* provee métodos que deben ser usados por todos aquellos controladores que requieran una acción HTTP POST. Es implementada por la clase *PostController*.



A continuación describimos los métodos de esta interfaz:

postAction(V object, E entity, Map<String, String> pathVariables): el primer argumento que recibe es un objeto tipo *Validable*, por lo que puede tratarse de una entidad o de un *datatype*. El segundo, es una entidad, y el último son las *pathVariables*. Hablaremos de ellas más adelante. Según el tipo de controlador, este método será tratado de una forma u otra:

En los `AbstractCreateController`, `AbstractUpdateController` y `AbstractDeleteController`, el primer argumento representa el objeto seguro (*safeObject*), es decir, el objeto sobre el que se han hecho las operaciones *beforeAuthorization* y *beforeCommitting*, y al que se le ha pasado el filtro anti post-hacking. Sobre ese objeto se han comprobado las reglas de negocio. El segundo argumento, *entity*, representa el objeto reconstruido a partir del formulario. Es un objeto, por tanto, no seguro. Debido a ello, cuando estas clases redefinen el método *postAction*, lo hacen llamando al método *save*, *update* o *delete* del servicio correspondiente, pasándoles como argumento el objeto seguro (primer argumento).

En cambio, en la clase `AbstractPostController` de *datatypes*, el primer argumento representa el *datatype*, y el segundo la entidad persistida en la base de datos.

public void businessRules(List<BusinessRule<E>> rules, List<Validator> validators): recibe una lista de reglas de negocio y validadores. El funcionamiento de estos se explicó en los servicios. En las clases `AbstractCreateController`, `AbstractUpdateController` y `AbstractDeleteController`, son redefinidos de forma que llaman al método *createBusinessRules*, *updateBusinessRules* o *deleteBusinessRules* del servicio

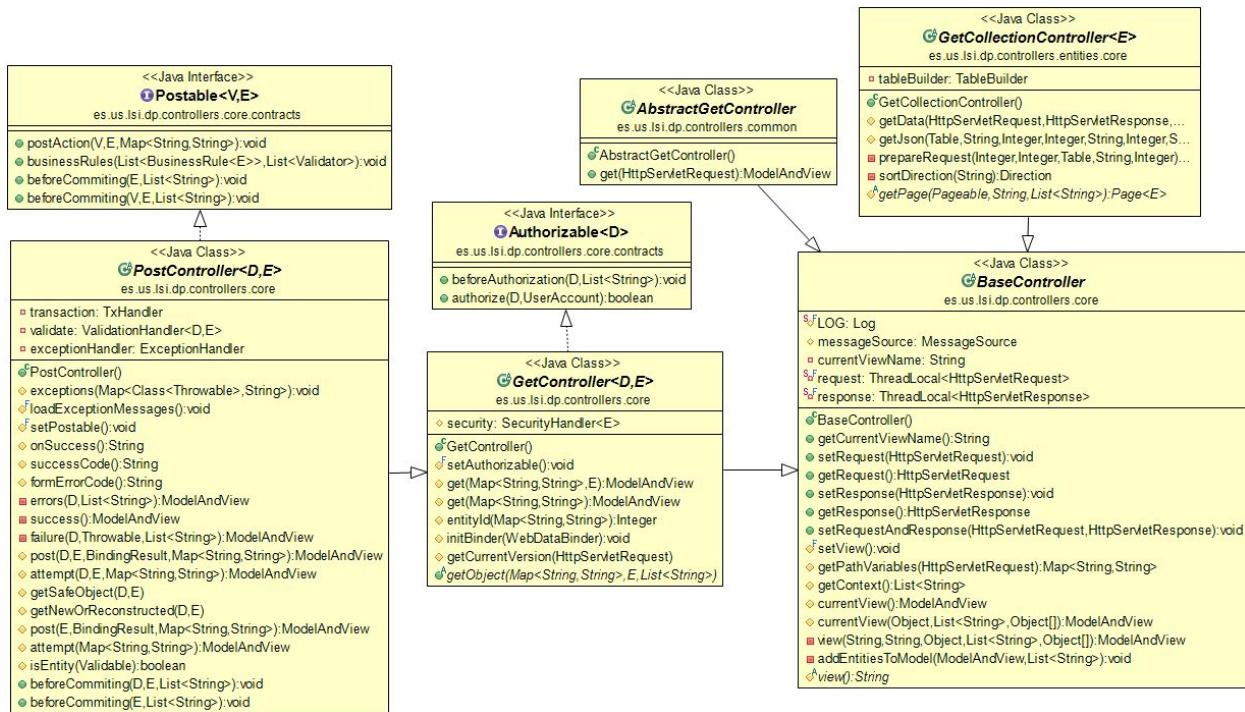
correspondiente. En *AbstractPostController* de *datatypes*, el método debe ser implementado en el controlador que cree el usuario. *businessRules* es llamado en el método *post* de *PostController*, antes de persistir la entidad en la base de datos.

public void beforeCommitting(E object, List<String> context): recibe el objeto reconstruido a partir del formulario enviado, que luego será convertido en *safeObject*, y el contexto. En las clases *AbstractCreateController*, *AbstractUpdateController* y *AbstractDeleteController*, son redefinidos de forma que llaman al método *beforeCommitingCreate*, *beforeCommitingUpdate* o *beforeCommitingDelete* del servicio correspondiente. En *AbstractPostController* de *datatypes* y de entidades, el método está definido de forma vacía por defecto, y no debe ser redefinido en ningún momento.

public void beforeCommiting(V datatype, E object, List<String> context): la idea es parecida a la del método anterior, pero pensado para *datatypes*. Es llamado a la vez que el anterior, pero es necesario que sea redefinido en el controlador que extienda a *AbstractPostController* de *datatypes*. Permite realizar operaciones sobre el *datatype* y la entidad antes de que la entidad *object* sea persistida en la base de datos.

5.2.3 BaseController

BaseController es la clase de la que todo controlador hereda. Provee métodos y atributos que ayudan a gestionar cualquier caso de uso.



En primer lugar, tenemos el atributo *messageSource*, de tipo *MessageSource*, declarado como *protected*. Al ser un componente de Spring, es posible inyectarle el objeto mediante la etiqueta *@Autowired*. Este atributo nos permitirá acceder, desde cualquier controlador, a los mensajes de internacionalización. Para usarlo, debemos pasarle una cadena que represente el código del mensaje, un array de objetos a inyectar en el mensaje (si no tenemos que

inyectar nada debe ser nulo), y un objeto de tipo *Locale*, indicando el idioma en el que el usuario tiene la aplicación.

```
@Autowired  
protected MessageSource messageSource;
```

El siguiente atributo almacena el nombre de la vista actual. Se establece mediante el método *setView*. Este método obtiene el nombre de la vista a partir del método *view*, declarado como abstracto en *BaseController*, y que debe ser redefinido en el controlador que implementemos. Como podemos ver, tiene la etiqueta *@PostConstruct* en su declaración. Esto hará que se ejecute después de injectar la instancia del controlador, es decir, cada vez que accedamos a una URI atendida por un controlador, el método *setView* será llamado^[10] [11].

También existe un método público *getCurrentViewName*, que devuelve el valor del atributo *currentViewName*. Es usado por el método *getSafeObject* de *PostController* para evitar el post-hacking, de manera que, como veremos en el capítulo de seguridad y protección contra el post-hacking, podrá obtener los campos marcados como protegidos en el fichero JSP de la vista.

```
private String currentViewName;  
  
@PostConstruct  
protected final void setView() {  
    currentViewName = view();  
}  
  
public String getCurrentViewName() {  
    return currentViewName;  
}  
  
protected abstract String view();
```

Los siguientes dos métodos son de tipo *ThreadLocal<HttpServletRequest>* y *ThreadLocal<HttpServletResponse>*, llamados *request* y *response*, respectivamente. Los objetos de tipo *ThreadLocal* representan un hilo de ejecución, y permite asociarle un objeto determinado, permitiendo su establecimiento y consulta dentro del mismo hilo. Lo que representan ambos atributos es, por tanto, la petición a la que está atendiendo el servlet y la respuesta que va a lanzar, respectivamente, y únicamente quedan accesibles dentro del mismo hilo de ejecución^[12].

```
private static final ThreadLocal<HttpServletRequest> request;  
private static final ThreadLocal<HttpServletResponse> response;  
  
public void setRequest(final HttpServletRequest request) {  
    BaseController.request.set(request);  
}
```

```

public HttpServletRequest getRequest() {
    return request.get();
}

public void setResponse(final HttpServletResponse response) {
    BaseController.response.set(response);
}

public HttpServletResponse getResponse() {
    return response.get();
}

public void setRequestAndResponse(final HttpServletRequest request, final
HttpServletResponse response) {
    setRequest(request);
    setResponse(response);
}

```

A continuación describiremos los métodos destinados a la generación de vistas y obtención de parámetros del contexto.

PathVariables y contexto

En primer lugar tenemos el método *getPathVariables*, que recibe la petición emitida al servlet. A partir de dicha petición, se obtiene un objeto de tipo *Map<String, String>*, al que llamaremos *pathVariables*. Representa los parámetros declarados en las URIs como “/{context}”. En la versión actual de Sprouts, sólo trabajaremos con el contexto, al que se puede acceder obteniendo el valor del mapa cuya clave es “context”. Se puede especificar otros parámetros en las URIs, declarando el nombre de la siguiente forma: “/{mi_parametro}”, y accediendo al valor cuya clave es “mi_parametro”. A continuación mostramos un ejemplo:

Si queremos pasar por medio de la URI un identificador y una cadena representando una fecha, declararemos la URI de esta forma:

“miEntidad/rol/{context}/{fecha}/hazAlgo.do”

Un ejemplo de URI con el patrón anterior puede ser el siguiente:

“miEntidad/rol/5/27-03-1995/hazAlgo.do”

Y sería posible acceder a esas path variables de esta forma:

pathVariables.get(“context”); → devuelve la cadena “5”.

pathVariables.get(“fecha”); → devuelve la cadena 27-03-1995.

Sprouts hace uso del contexto para permitir una mayor flexibilidad de cara al desarrollador, sin tener que tocar demasiado las URIs. Así pues, queda justificada la existencia del contexto, que se obtiene en el método definido a continuación: *getContext*. Devuelve una

lista de cadenas, y se obtiene a partir de la clave “context” del mapa *pathVariables*. El contexto puede tener tantos parámetros como se desee, y deben estar separados por caracteres coma (“,”). A continuación mostramos el mismo ejemplo que hemos usado para explicar las path variables, pero ahora lo haremos usando el contexto:

“miEntidad/rol/{context}/hazAlgo.do”

Ejemplos de URI con el patrón anterior:

“miEntidad/rol/5,27-03-1995/hazAlgo.do”

“miEntidad/rol/5,27-03-1995,cadena_adicional/hazAlgo.do”.

Para obtener la cadena “5”: *context.get(0)*;

Para obtener la cadena “27-03-1995”: *context.get(1)*

Para obtener la cadena “cadena_adicional”: *context.get(2)*.

Si accedemos a una posición en la que no existe ningún parámetro, obtendremos una excepción de tipo *IndexOutOfBoundsException*.

Aconsejamos que los parámetros se pasan a la URI usando el contexto, ya que permite más flexibilidad al programador, y evitará tener que declarar nuevas path variables. Viene declarado por defecto en todas las URIs que atienden a los casos de uso de creación, actualización, listado, muestra de información, y eliminación de entidades. Cuando se expliquen estos controladores, quedará más claro cuando presentemos el formato que tienen por defecto las URIs. Asimismo, en los casos de uso más complejos, especificaremos cuándo debemos declarar explícitamente el contexto o las path variables.

Ambos métodos tienen el modificador de acceso *protected*, por lo que es posible acceder a las path variables y al contexto desde cualquier controlador, aunque la mayoría de métodos recibe el contexto como parámetros para acceder directamente a la lista de cadenas.

```
protected Map<String, String> getPathVariables(final HttpServletRequest request) {
    Map<String, String> result;

    result = (Map<String, String>) request.getAttribute(HandlerMapping.URI_TEMPLATE_VARIABLES_ATTRIBUTE);

    return result;
}

protected List<String> getContext() {
    return ContextParser.parse(getPathVariables(getRequest()));
}
```

Métodos de construcción de vistas

Los siguientes tres métodos permiten generar vistas: los dos primeros son llamados desde los controladores más específicos, y el último es un método auxiliar, es llamado desde el segundo *currentView*, y se encarga de generar la vista a partir de ciertos parámetros.

El primer método *currentView* no recibe ningún parámetro. Es llamado desde *AbstractGetController* y *AbstractListController*. Devuelve una vista dado un nombre (definido por el usuario), y añade al modelo y atributo “_viewName”, cuyo valor es el nombre de la vista, y es necesario para mostrar tablas con el componente Datatable. Para crear la vista se usa el método *Response.create()*, que explicaremos en el método *view*. Si nuestro controlador implementa la interfaz *AddsToModel*, añadirá los atributos que el desarrollador haya especificado al modelo.

El segundo método *currentView*, recibe un objeto, que será aquel con el que se trabaje en la vista, el contexto, y una serie de parámetros de tipo *Object*. El tamaño de ese array debe ser múltiplo de dos, y cada dos celdas se interpretarán como parejas clave - valor. Serán añadidas al modelo. De ahora en adelante, nos referiremos a estos parámetros como *arguments*.

Para devolver la vista, se hace uso del método *view*, que recibe el nombre de la vista, el nombre que tendrá la clave en el modelo del objeto principal con el que trabajará la vista, el objeto, el contexto y el parámetro *arguments*. El método *view*, crea la vista llamando al método estático *create* de la clase *Response*, localizada en el paquete *es.us.lsi.dp.utilities*. Recibe el nombre de la vista y el parámetro *arguments*. Este método crea la vista a partir del nombre proporcionado, y añade al modelo los objetos especificados en el parámetro *arguments*. Luego, añade al modelo el objeto que se mostrará en la vista, dándole como nombre *modelObject*. Esta cadena viene representada por el atributo estático *MODEL_OBJECT_NAME* de la clase *Codes*. Asimismo, añade al modelo una cadena que representa el nombre de la vista actual, la cual es obtenida del atributo *currentViewName* de *BaseController*. El nombre del atributo del modelo será *_viewName*, definido por el atributo estático *VIEW_NAME* de la clase *Codes*.

Por último, el método comprueba si el controlador creado implementa la interfaz *AddsToModel*. Si es así, llamará al método *addEntitiesToModel* de *BaseController*, que explicaremos a continuación.

```
protected ModelAndView currentView() {
    ModelAndView result;
    result = Response.create(currentViewName, Codes.VIEW_NAME,
    currentViewName);
    if (this instanceof AddsToModel)
        addEntitiesToModel(result, getContext());
    return result;
}

protected ModelAndView currentView(final Object domainObject, final
List<String> context, final Object... arguments) {
    ModelAndView result;
```

```

        result = view(currentViewName, Codes.MODEL_OBJECT_NAME, domainObject,
context);

    return result;
}

private ModelAndView view(final String viewName, final String domainObjectName,
final Object modelObject, final List<String> context,
final Object... arguments) {
    ModelAndView result;
    boolean usesEntities;
    result = Response.create(viewName, arguments);

    result.addObject(Codes.MODEL_OBJECT_NAME, modelObject);
    result.addObject(Codes.VIEW_NAME, currentViewName);

    usesEntities = this instanceof AddsToModel;

    if (usesEntities)
        addEntitiesToModel(result, context);

    return result;
}

```

La misión del método `addEntitiesToModel` es añadir los atributos al modelo que el usuario especificó en el método `addToMany`, implementando en el controlador la interfaz `AddsToModel`. En primer lugar, se hace una conversión forzosa a un objeto de tipo `AddsToModel` de la instancia del controlador con la estamos trabajando. Luego, llama al método `addToMany` de la instancia, pasando como parámetro un `Map<String, Object>` vacío. De esta manera, se añadirán al mapa los objetos que se especificaron en nuestro controlador. En el apartado 5.3.9 encontrará más información acerca de esta interfaz.

```

private void addEntitiesToModel(final ModelAndView modelAndView, final
List<String> context) {
    AddsToModel usesEntities;
    Map<String, Object> objects;

    objects = new HashMap<>();
    usesEntities = (AddsToModel) this;

    usesEntities.addToMany(objects, context);

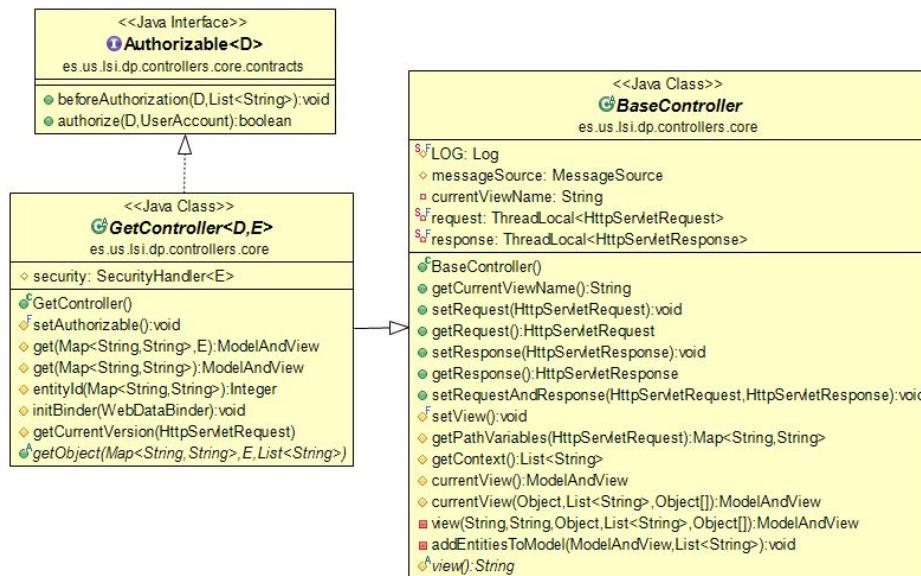
    modelAndView.addAllObjects(objects);
}

```

5.2.4 GetController

`GetController` nos dota de métodos para responder a peticiones de tipo HTTP GET. Puede ser llamado para obtener cualquier tipo de vista que implique trabajar con un objeto del

modelo de dominio. *GetController* es, además, usada por *PostController*. Por consiguiente, es llamada para obtener la entidad a usar en formularios, o para crear una vista en la que sea necesario mostrar información de una entidad.



Como podemos ver, hereda de *BaseController*, e implementa la interfaz *Authorizable*. La función de esta interfaz se explicó en el apartado 5.2.1.

GetController declara un atributo de tipo *SecurityHandler<E>*, donde *E* es el tipo de la entidad con la que trabajará el controlador. La función de este atributo dentro de este controlador, es verificar que el usuario que provoca la llamada al controlador esté realmente autorizado para hacerlo. La clase *SecurityHandler* se explica con detalle en el capítulo Seguridad y prevención de Hacking.

```
// Authorization
@Autowired
protected SecurityHandler<E> security;
```

El siguiente método, *setAuthorizable*, establece el objeto de tipo *Authorizable* con el que debe trabajar la instancia de la clase *SecurityHandler* que acabamos de describir. El objeto será la instancia del controlador. Este método tiene la etiqueta *@PostConstruct*^{[10][11]} cuya finalidad fue explicada en el apartado 5.2.3. En resumen, hace que el método sea llamado al crear una instancia de *GetController*.

```
// Initialisation
@PostConstruct
protected final void setAuthorizable() {
    security.setAuthorizable(this);
}
```

El método *get* se encarga de devolver la vista que responde a una petición HTTP GET. Existen dos posibles invocaciones: una en la que recibe la entidad a mostrar, y otra en la que no la recibe. Este último es llamado desde *AbstractCreateController*, *AbstractUpdateController*, *AbstractShowController* y *AbstractDeleteController*, entre otros, desde el método que captura la petición HTTP GET. Por otra parte, el método *get* que recibe una entidad es llamado desde *AbstractPostController* del paquete *controllers.datatypes*. A continuación describimos el comportamiento de ambos métodos.

En primer lugar, se obtiene el objeto que se enviará a la vista: *domainObject*. Para ello, llamamos al método *getObject*, declarado en *GetController* como abstracto, y que, pasando como parámetros las path variables, una entidad de dominio, y el contexto, permiten obtener el objeto a enviar a la vista. El objeto *entity* que se envía al método *getObject*, y que es recibido por *get*, se usa en el caso de trabajar con *datatypes*, de manera que facilita la obtención del datatype con el que trabajar a partir de la entidad en la que se hace uso de este.

Una vez que se obtiene el objeto que se le pasará a la vista (entidad de dominio o datatype), llamamos al método *beforeAuthorization*, definido por la interfaz *Authorizable*, y recibiendo como parámetros el objeto a devolver por la vista y el contexto. Este método fue explicado en el apartado 5.2.1. Cabe destacar que únicamente se llama a este método si el objeto *entity*, recibido por *get*, es nulo, es decir, si no estamos trabajando con *datatypes*, ya que, como hemos explicado, si no es nulo es porque se llama desde *AbstractPostController* del paquete *controllers.datatypes*.

Posteriormente, se llama al método *authorize* del objeto *security*, declarado en *GetController*, pasando como parámetro el objeto con el que trabajará la vista. Brevemente explicado, este método llama al método *authorize* declarado por la interfaz *Authorizable*, y que debe ser redefinido por cualquier controlador que extienda a *GetController*. La función de esta llamada es comprobar que el usuario principal esté autorizado para acceder al objeto que se mostrará en la vista.

Finalmente, para obtener la vista, se hace una llamada a *currentView*, método implementado en *BaseController*, recibiendo el objeto a mostrar en la vista y el contexto.

```
// Responses
protected ModelAndView get(final Map<String, String> pathVariables,
final E entity) {
    ModelAndView result;
    D domainObject;
    E entityDomainObject;

    domainObject = getObject(pathVariables, entity,
    ContextParser.parse(pathVariables));

    Assert.notNull(domainObject);

    entityDomainObject = entity;

    if (entity == null) {
```

```

        entityDomainObject = (E) domainObject;
        beforeAuthorization(entityDomainObject,
ContextParser.parse(pathVariables));
    }

    security.authorize(entityDomainObject);

    result = currentView(domainObject,
ContextParser.parse(pathVariables));

    return result;
}

// Alternative invocations
protected ModelAndView get(final Map<String, String> pathVariables) {
    return get(pathVariables, null);
}

public abstract D getObject(Map<String, String> pathVariables, E entity,
List<String> context);

```

El siguiente método, *entityId*, simplemente obtiene el identificador de una entidad a partir de las *pathVariables*. Para ello, se obtiene la cadena situada en la posición más avanzada de las *path variables*. En la práctica, la cadena que saca es el contexto. El funcionamiento de las *pathVariables* y el *contexto*, se explica en el apartado 5.2.3.

```

protected Integer entityId(final Map<String, String> pathVariables) {
    String pathVariable;
    Integer result;

    pathVariable = Collections.max(pathVariables.keySet());

    result = Integer.valueOf(pathVariables.get(pathVariable));

    return result;
}

```

El método *initBinder* tiene la finalidad de formatear los datos que se mostrarán en la vista y parsear los datos introducidos por el usuario en un formulario. La etiqueta *@InitBinder* produce que el método sea llamado antes de enviar la vista y cuando el usuario envíe un formulario. El objeto de tipo *WebDataBinder* que recibe como parámetro permite especificar los editores que se aplicará al objeto que se mostrará en la vista^[13]. Para obtener los editores, se hace uso de tres tipos de formateadores que fueron creados para trabajar con ellos: *CustomDateFormat*, *CustomCurrencyFormat* y *CustomDecimalFormat*.

En primer lugar, se obtienen los formateadores de dos formas: la primera es a partir del controlador que el usuario crea. Para ello, debe haber implementado la interfaz

AddCustomFormat. De esta manera, se obtienen los formateadores que el usuario definió en el método *addCustomFormats*. La segunda forma de obtener los formateadores es a partir de la clase *DefaultFormats*, donde se definen los formateadores por defecto dentro del método *getDefaultFormats*.

Luego, para cada formateador recogido, se llama al método *registerCustomEditor* del objeto *WebDataBinder*. A este método hay que pasarle como parámetro el tipo del atributo de la entidad a la que aplicarle el editor, el nombre del campo, y un editor, al que se le debe pasar el formato.

```
@InitBinder
protected void initBinder(WebDataBinder binder) {

    List<CustomFormat> formats = new ArrayList<>();

    if (this instanceof AddCustomFormat) {
        AddCustomFormat addCustomInternationalization =
(AddCustomFormat) this;
        addCustomInternationalization.addCustomFormats(formats);
    }

    formats.addAll(DefaultFormats.getDefaultFormats());

    for (CustomFormat format : formats) {
        format.setDataSource(messageSource);
        if (format instanceof CustomDateFormat) {
            CustomDateFormat dateFormat = (CustomDateFormat)
format;
            if (dateFormat.getField().equals(""))

binder.registerCustomEditor(dateFormat.getType(), new
CustomDateEditor(dateFormat.getFormat(), true));
            else

binder.registerCustomEditor(dateFormat.getType(), dateFormat.getField(),
new CustomDateEditor(dateFormat.getFormat(), true));
        }

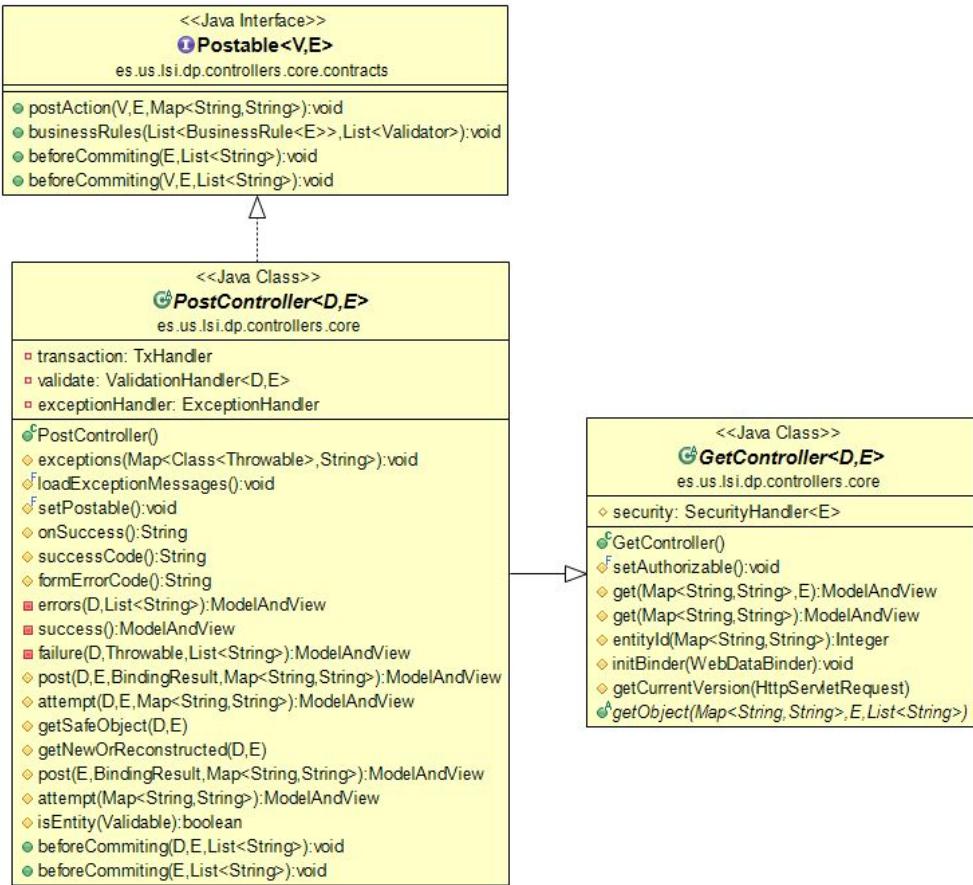
        if (format instanceof CustomCurrencyFormat) {
            CustomCurrencyFormat currencyFormat =
(CustomCurrencyFormat) format;
            if (currencyFormat.getField().equals(""))

binder.registerCustomEditor(currencyFormat.getType(), new
CustomNumberEditor(currencyFormat.getType(), currencyFormat.getFormat(),
true));
            else
                binder.registerCustomEditor(currencyFormat.getType(),
currencyFormat.getField(), new
CustomNumberEditor(currencyFormat.getType(),
currencyFormat.getFormat(), true));
        }
    }
}
```

```
        if (format instanceof CustomDecimalFormat) {
            CustomDecimalFormat decimalFormat =
(CustomDecimalFormat) format;
            if (decimalFormat.getField().equals(""))
                binder.registerCustomEditor(decimalFormat.getType(), new
CustomNumberEditor(decimalFormat.getType(), decimalFormat.getFormat(),
true));
        else
            binder.registerCustomEditor(decimalFormat.getType(),
decimalFormat.getField(), new
CustomNumberEditor(decimalFormat.getType(),
decimalFormat.getFormat(), true));
    }
}
```

5.2.5 PostController

La clase *PostController* está destinada a realizar todas las operaciones que impliquen una respuesta a una petición HTTP POST.



PostController hereda de *GetController*, lo cual permite usar sus métodos para crear las vistas que supongan una respuesta a una petición HTTP POST. También implementa la interfaz *Postable*, que fue descrita con detalle en el apartado 5.2.2.

PostController tiene tres atributos privados. El primer de ellos es *transaction*, de tipo *TxHandler*. Esta clase provee métodos para definir transacciones. Nos interesan los siguientes métodos, que serán usados en esta clase:

- *begin()*: indica el comienzo de una transacción.
- *commit()*: finaliza la transacción con éxito.
- *rollback()*: cancela la transacción.

El siguiente atributo es *validate*, de tipo *ValidationHandler*. Esta clase se describe en el capítulo 2.6. El último atributo es *exceptionHandler*, de tipo *ExceptionHandler*. Permite gestionar las excepciones.

```

// Transactions
@Autowired
private TxHandler transaction;

// Validation
@Autowired
private ValidationHandler<D, E> validate;

```

```
// Exceptions
@Autowired
private ExceptionHandler exceptionHandler;
```

El siguiente método, se ejecuta al crear una instancia de *PostController*, y sirve para establecer el atributo *validate*, pasando como parámetro la instancia actual de *PostController*. Tiene la etiqueta *@PostConstruct*^{[10][11]}, lo que produce que se invoque este método al crear una instancia de *PostController*.

```
@PostConstruct
protected final void setPostable() {
    validate.setPostable(this);
}
```

Los siguientes tres métodos llaman al método *currentView*, y son llamados en respuesta a una acción POST. El primer método es *errors*, y es llamado en caso de que existan errores en el formulario enviado por el usuario. El método *currentView* recibe como parámetro el objeto de dominio con el que se estaba trabajando en el formulario, el contexto, el código *failureCode* (atributo público *FAILURE_CODE* de la clase *Codes*), y el nombre del mensaje de internacionalización que refleja el mensaje de error. Este código viene dado por el método *formErrorCode*, que simplemente devuelve la cadena “*form-error.message*”.

El siguiente método, *success*, se encarga de redirigir a una URI en caso de que la acción POST se lleve a cabo con éxito. La URI a donde debe redirigir viene dada por el método *onSuccess*. Por defecto, va a redirigir la URI de listado de entidades con la que trabaje el controlador. Esta URI viene definida por el atributo estático *DEFAULT_SUCCESS_CODE* de la clase *Codes*.

Por último, el método *failure* es llamado en caso de que se produzca un error durante la transacción. En este caso, se devolverá una excepción, obtenida a través del atributo *exceptionHandler*.

```
// Responses
private ModelAndView errors(final D domainObject, final List<String>
context) {
    return currentView(domainObject, context, Codes.FAILURE_CODE,
formErrorCode());
}

protected String formErrorCode() {
    return Codes.DEFAULT_FORM_ERROR_CODE;
}

private ModelAndView success() {
```

```

    return Response.redirect(onSuccess(), Codes.SUCCESS_CODE,
successCode());
}

protected String successCode() {
    return Codes.DEFAULT_SUCCESS_CODE;
}

private ModelAndView failure(final D domainObject, final Throwable oops,
final List<String> context) {

    ModelAndView result;

    LOG.error(oops, oops);

    result = currentView(domainObject, context, Codes.FAILURE_CODE,
exceptionHandler.message(oops), Codes.THROWABLE, oops);

    return result;
}

```

A continuación, explicamos el comportamiento del método post. Tiene dos posibles invocaciones. La primera de ellas, recibe un objeto de tipo *DomainObject*, que puede ser una entidad o un *datatype*, un objeto de tipo *DomainEntity*; un objeto de tipo *BindingResult*, donde se establecen los resultados de las validaciones, y las path variables. Esta invocación es realizada desde el otro método post de *PostController*, y desde *AbstractPostController* del paquete *controllers.datatypes*. La otra invocación, recibe una *DomainEntity*, un objeto de tipo *BindingResult*, y las path variables. Este método se invoca únicamente desde *AbstractCreateController*, *AbstractUpdateController*, y *AbstractDeleteController*.

Para describir el método post, debemos tener en cuenta el rol de cada uno de los siguientes objetos:

- *entityOrDatatype*: objeto de tipo *DomainObject* que será un *datatype* si la invocación se hace desde *AbstractPostController* del paquete *controllers.datatype*. De lo contrario, será una *DomainEntity*.
- *entity*: objeto de tipo *DomainEntity*. Si la invocación se produce desde *AbstractPostController* del paquete *controllers.datatype*, esta entidad será la que hace uso del *datatype entityOrDatatype*. De lo contrario, será el mismo objeto que *entityOrDatatype*.
- *safeObject*: objeto de tipo *DomainObject*, que se obtiene a partir del método *getSafeObject*, que recibe como parámetros *entityOrDatatype* y *entity*. Este método comprueba si el objeto *entityOrDatatype* es una instancia de *DomainEntity*. De ser así, llama al método *getPostHackingSafeObject* del atributo *security* de la clase *GetController*, que es de tipo *SecurityHandler*. Lo que hace el método es, obtener el objeto que el usuario ha modificado, restableciendo los campos marcados como protegidos en la vista al valor que tenía antes de que el usuario los modificara. Si por

el contrario, *entityOrDatatype* no es una instancia de *DomainEntity*, devolverá el objeto *entityOrDatatype* que recibió como parámetro.

- *newOrReconstructed*: se obtiene mediante el método *getNewOrReconstructed*, que recibe como parámetros *safeObject* y *entity*. Simplemente, comprueba si *entity* es una instancia de *DomainEntity*. Si es así, devuelve *safeObject*. De lo contrario, devuelve *entity*. Las validaciones se aplicarán al objeto *newOrReconstructed*.
- *entityToAuthorize*: si *entityOrDatatype* es una instancia de *DomainEntity*, devuelve *safeObject*, de lo contrario, devuelve *entity*.

Una vez que sabemos esto, podemos describir el método *post*. El método entero se ejecuta como una única transacción, que empieza después de declarar los atributos, y termina o bien cuando se produce alguna excepción, o bien cuando el método *attemp* ejecuta con éxito el método *postAction*.

El método comienza llamando a *beforeAuthorization*. Una vez hecho esto, obtiene el *safeObject*.

El siguiente paso es obtener el objeto *entityToAuthorize*. Sobre este objeto, se aplicará el método *authorize* del atributo *security* de *GetController*. Este método comprueba si el usuario principal está autorizado para continuar la transacción. En el caso de estar trabajando con entidades, *entityToAuthorize* será *safeObject*. Si estamos trabajando con *datatypes*, será *entity*.

A continuación se llama al método *constraints* del atributo *validate*. Este método aplica los validadores de la entidad con la que estamos trabajando. Recibe como parámetro *safeObject* y *bindingResult*, objeto sobre el que se especificarán los resultados de la validación.

Una vez hecho esto, se comprueba si hubo errores de validación. Si es así, entonces la transacción se cancela, llamando al método *rollback* del atributo *transaction*, y devuelve una vista informando de los errores del formulario. Para ello, se hace uso del método *errors*.

Si las validaciones tuvieron éxito, se llama a un par de métodos *beforeCommitting*. El primero, es definido por la interfaz *Authorizable*, implementada por *GetController*, y cuyo comportamiento se define en los controladores que heredan de *PostController*. Luego, se llama a *beforeCommitting*, de la interfaz *Postable*. La finalidad de este método es que el desarrollador pueda realizar modificaciones en el objeto enviado por el usuario. Se llama únicamente cuando el usuario envía el objeto.

A continuación, se llama al método *beforeCommitting*, que es distinto al anterior, puesto que este está destinado a tratar con *datatypes*, de manera que se le pasa como parámetros los objetos *entityOrDatatype* y *entity*.

Se vuelve a obtener el *safeObject* a partir del objeto *entity* o *entityOrDatatype* al que se le aplica el *beforeCommitting*. Una vez hecho esto, comprobamos las reglas de negocio. Estas se comprueban dentro de un bloque *try/catch*. En primer lugar, se obtiene el objeto *newOrReconstructed*, sobre el que se aplicarán las reglas de negocio, llamando al método *businessRules* del atributo *validate*. En caso de que alguna regla falle, se capturará dentro

del catch, donde se cancelará la transacción, llamando al método *rollback* de *transaction*. Una vez hecho esto, devolverá la vista indicando el fallo, invocando al método *failure*, previamente descrito.

- i** Tenga en cuenta que, si en los métodos *beforeCreating*, *beforeUpdating*, *beforeCommittingCreate*, etc, modifica atributos que en el formulario estaban marcados como protegidos, los cambios aplicados sobre ellos quedarán sin efecto, puesto que se recuperará el valor que tenía ese atributo en la base de datos.

Finalmente, si hasta ahora no hubo ningún problema, obtenemos la vista llamando al método *attempt*. Este método llama a *postAction*, definido en la interfaz *Postable*, y que debe ser implementado por los controladores específicos de cada caso de uso. Si la operación que se realiza en el método *postAction* se ejecuta con éxito, se redirigirá a una vista indicando el éxito, llamando al método *success*. Tras esto, se llama al método *commit* de la transacción, para finalizarla. Si durante la ejecución de *postAction* hubo algún problema, se devolverá una vista indicando el fallo, llamando al método *failure*, y se cancela la transacción.

```
protected ModelAndView post(final D entityOrDatatype, final E entity,
final BindingResult bindingResult, final Map<String, String>
pathVariables) {
    D safeObject;
    E newOrReconstructed;
    E entityToAuthorize;
    ModelAndView result = null;
    boolean success = true;

    transaction.begin();
    beforeAuthorization(entity, ContextParser.parse(pathVariables));

    safeObject = getSafeObject(entityOrDatatype, entity);

    entityToAuthorize = isEntity(entityOrDatatype) ? (E) safeObject : entity;

    security.authorize(entityToAuthorize);

    validate.constraints(safeObject, bindingResult);

    // We can't continue because business rules may need access some
    // fields
    // the user didn't fill in.
    if (ValidationHandler.validationFailed(bindingResult)) {
        success = false;
        transaction.rollback();
        result = errors(entityOrDatatype,
ContextParser.parse(pathVariables));
    }
}
```

```

        if(success) {
            beforeCommitting(entity, ContextParser.parse(pathVariables));
            // This method if defined in this class and does nothing. It is
            // needed
            // to redefine if we are dealing with datatypes
            beforeCommitting(entityOrDatatype, entity,
ContextParser.parse(pathVariables));

            safeObject = getSafeObject(entityOrDatatype, entity);

            try {
                newOrReconstructed = getNewOrReconstructed(safeObject,
entity);
                validate.businessRules(newOrReconstructed);
            } catch (final Throwable oops) {
                transaction.rollback();
                return failure(safeObject, oops,
ContextParser.parse(pathVariables));
            }

            result = attempt(safeObject, entity, pathVariables);
        }
        return result;
    }

protected ModelAndView attempt(final D entityOrDatatype, final E entity,
final Map<String, String> pathVariables) {
     ModelAndView result;

    try {
        postAction(entityOrDatatype, entity, pathVariables);
        result = success();
        transaction.commit();
    } catch (final Throwable oops) {
        result = failure(entityOrDatatype, oops,
ContextParser.parse(pathVariables));
        transaction.rollback();
    }

    return result;
}

protected D getSafeObject(final D entityOrDatatype, final E entity) {
    D result;
    E safeObject;

    result = entityOrDatatype;

    if (isEntity(entityOrDatatype)) {
        safeObject = security.getPostHackingSafeObject(entity,
getCurrentViewName(), getRequest(), getResponse());
    }
}

```

```

        result = (D) safeObject;
    }
    return result;
}

protected E getNewOrReconstructed(final D safeObject, final E
newVersion) {
    E result;

    result = isEntity(safeObject) ? (E) safeObject : newVersion;

    return result;
}

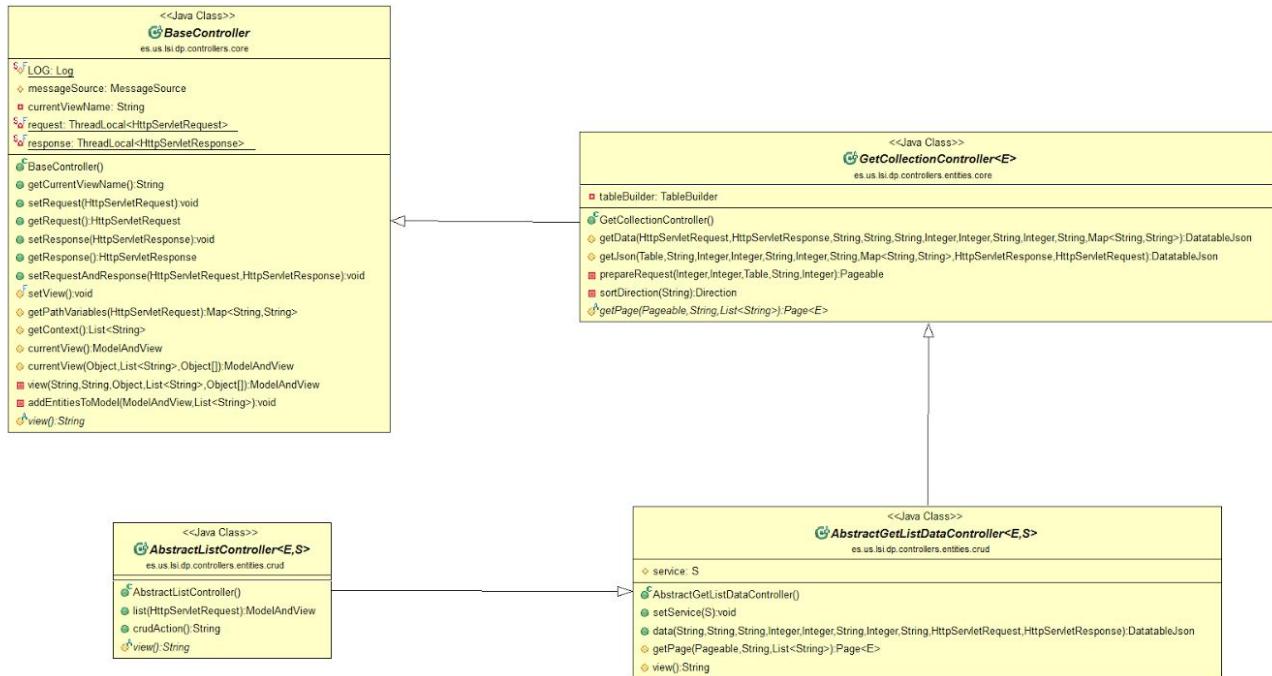
// Alternative invocations
protected ModelAndView post(final E newVersion, final BindingResult
bindingResult, final Map<String, String> pathVariables) {
    return post((D) newVersion, newVersion, bindingResult,
pathVariables);
}

protected ModelAndView attempt(final Map<String, String> pathVariables)
{
    return attempt(null, null, pathVariables);
}

```

5.2.6 Controladores de listado

En lo que respecta a los controladores de listado, la estructura de clases existente es la siguiente:



En primer lugar, se dispone de una clase *GetCollectionController*. Esta clase hereda del controlador base ya comentado anteriormente *BaseController*, del cual van a extender la mayoría de los controladores del núcleo a usar. Esta clase se encarga de devolver los datos en JSON necesarios para el componente DataTable.

A su vez, otra clase *AbstractGetListDataController* hereda de *GetCollectionController*. Esta última tiene como función recibir las peticiones GET del usuario cuando solicita los datos de una tabla de un listado y devolvérselos.

Finalmente, está la clase *AbstractListController*. Hereda de *AbstractGetListDataController* y de esta clase a su vez deben heredar todos los controladores de listado de los que se quieran hacer uso en el proyecto. El objetivo de este controlador es dirigir la petición del usuario para devolver una vista donde aparezca el listado correspondiente.

5.2.7 SecurityController

La clase *SecurityController* sirve para gestionar la entrada y salida del canal HTTPS. Concretamente, controla que al iniciar sesión, se redirija al usuario a una URI con canal HTTPS, y al cerrarla, a un canal HTTP.

La clase tiene dos métodos:

- *manageSecurity*: responde a las peticiones GET a la URI “/security/manage-security.do”. Devuelve la vista de bienvenida de la aplicación. Se redirigirá al usuario a esta URI al iniciar sesión.
- *manageSecurityLogout*: responde a las peticiones GET a la URI “/security/manage-security-logout.do”. Devuelve la vista de bienvenida de la aplicación. Se redirigirá al usuario a esta URI al cerrar sesión.

En el fichero *security.xml*, se especifica que al acceder a “/security/manage-security.do”, el usuario entrará en una conexión HTTPS, y al acceder a “/security/manage-security-logout.do”, la conexión será HTTP.

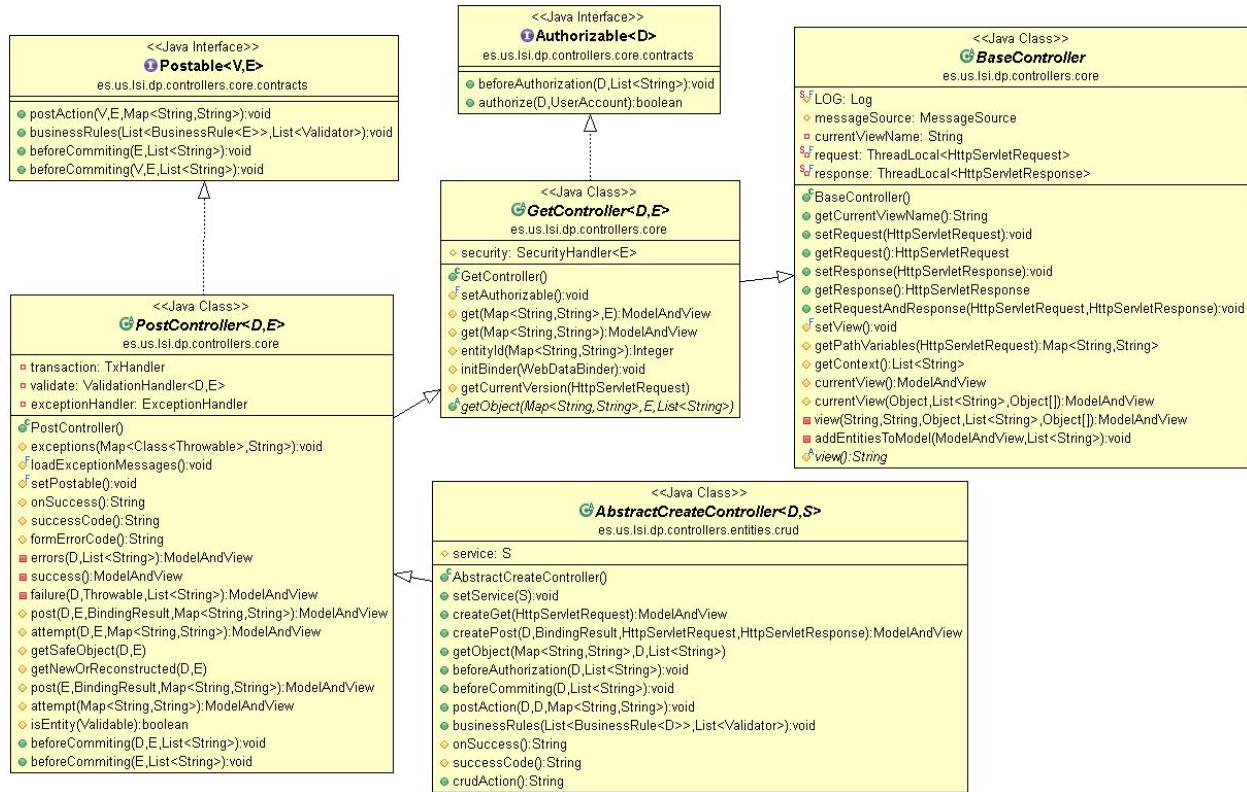
5.3 Controladores para casos de uso

5.3.1 AbstractCreateController

Los controladores que vayan destinados a la creación de entidades, deben extender a *AbstractCreateController*. Debe recibir como parámetros genéricos una clase que extienda a *DomainObject*, es decir, una entidad o un objeto *form*, y el servicio que trabaje con dicha clase. Además de extender a esta clase, deben tener las etiquetas *@Controller*, que recibe como argumento un identificador único del controlador, y *@RequestMapping*, recibiendo la primera parte de la URI a la que responderá este controlador. Por convenio, la primera parte estará compuesta por el nombre de la entidad con la que trabaja el controlador y el rol que puede acceder a la vista. Sirva como ejemplo la cabecera de la clase *CreateController* de *Folder* (proyecto Acme-Barter):

```
@Controller("folderCreateAdministrator")
@RequestMapping("folder/administrator")
public class CreateController extends AbstractCreateController<Folder,
    FolderService> {
```

Vamos a describir el funcionamiento de la clase *AbstractCreateController* para entender cómo se gestionan los casos de uso de creación de entidades.



En primer lugar, *AbstractCreateController* tiene un atributo llamado *service* con modificador de acceso *protected*, que representa el servicio con el que trabaja el controlador. Gracias a este atributo, en cualquier parte de nuestro controlador podremos acceder al servicio.

```
protected S service;
```

A continuación, tenemos los métodos *createGet* y *createPost*. Ambos tienen la etiqueta *@RequestMapping*, indicando que responderán a las peticiones HTTP GET y HTTP POST, respectivamente, y que responden a las URIs declaradas en la clase *Codes* como atributos estáticos.

Ambos métodos responderán a las URIs representadas por los códigos *CREATE_MAPPING_VALUE* y *CREATE_MAPPING_VALUE_PARAMS*. El primero representa todas las URIs que empiecen como se especifica en la etiqueta *@RequestMapping* de la cabecera de la clase del controlador con el que estemos trabajando, y terminen en `"/create.do"`, mientras que el segundo permite recibir parámetros, que llamaremos contexto o path variables, antes de `create.do`: `"/{context}/create.do"`.

El contexto es una cadena separada por comas (","), que sirve para especificar parámetros mediante una URI, por ejemplo, el identificador de una entidad. El contexto es recibido como parámetro por varios métodos de controladores y servicios como una lista de enteros, en la que cada posición representa una cadena.

El método *createGet* devuelve un objeto de tipo *ModelAndView*. Para ello, llama al método *get*, pasándole un mapa que se obtiene a partir de la petición HTTP recibida como parámetro. De aquí se obtendrá el contexto. La vista devuelta por este método representa un formulario de creación de entidades. El método *get* es heredado por *PostController*, que a su vez lo hereda de *GetController* y, como explicamos antes, crea una vista a partir del objeto devuelto por *getObject*.

```
@RequestMapping(value = {Codes.CREATE_MAPPING_VALUE,
    Codes.CREATE_MAPPING_VALUE_PARAMS}, method = RequestMethod.GET)
public ModelAndView createGet(@final HttpServletRequest request) {
    return get(getPathVariables(request));
}
```

El método *createPost* es llamado cuando se realiza una petición POST sobre las URLs descritas anteriormente, por ejemplo, al enviar un formulario. Recibe la entidad con la que estamos trabajando con los datos del formulario, un objeto *BindingResult* con los resultados de la validación, que se comprobará dentro del método *post* de *PostController*, y dos objetos que representan la petición y respuesta HTTP. En primer lugar, se llama a *setRequestAndResponse* de *BaseController*, pasándole como argumentos la petición y respuesta HTTP, para que se pueda acceder a ellos en cualquier momento durante la transacción.

Por último, llama al método *post* de *PostController*, pasándole como argumentos la entidad reconstruida a partir de los datos del formulario, el objeto *BindingResult*, y, al igual que en el método *createGet*, un mapa, a partir del cual se sacará el contexto.

El método *post*, como ya describimos anteriormente, se encarga de ejecutar la transacción de persistir la entidad contra la base de datos.

```
@RequestMapping(value = {Codes.CREATE_MAPPING_VALUE,
    Codes.CREATE_MAPPING_VALUE_PARAMS}, method = RequestMethod.POST)
public ModelAndView createPost(@ModelAttribute(Codes.MODEL_OBJECT_NAME) final D
    domainObject, final BindingResult bindingResult, HttpServletRequest request,
    final HttpServletResponse response) {
    setRequestAndResponse(request, response);
    return post(domainObject, bindingResult,
        getPathVariables(request));
}
```

AbstractCreateController implementa el método *getObject*. Devuelve una instancia de la entidad con la que trabaja el controlador, llamando al método *create* del servicio.

```
@Override
public D getObject(@final Map<String, String> pathVariables, @final D entity,
    List<String> context) {
```

```
        return service.create();
    }
```

Se definen tres métodos de la interfaz *Postable*: *beforeCommiting*, que llama al método *beforeCommitingCreate* del servicio; *postAction*, que se encarga de llamar a *save* del servicio y *afterCommitingCreate*; *businessRules*, que llama a *createBusinessRules* pasándole las listas con las reglas de negocio y validadores. Se define también el método *beforeAuthorization* de la interfaz *Authorizable*. Este método llama a *beforeCreating*.

Todos estos métodos son llamados desde el método *post* de *PostController*, excepto *postAction*, que es llamado desde el método *attempt*, también de *PostController*.

```
@Override
public void beforeAuthorization(final D domainObject, List<String> context) {
    service.beforeCreating(domainObject, context);
}

@Override
public void beforeCommiting(D entity, List<String> context) {
    service.beforeCommitingCreate(entity, context);
}

@Override
public void postAction(final D domainObject, final D duplicatedDomainObject,
final Map<String, String> pathVariables) {
    int id = service.save(domainObject);
    service.afterCommitingCreate(id);
}

@Override
public void businessRules(final List<BusinessRule<D>> rules, final
List<Validator> validators) {
    service.createBusinessRules(rules, validators);
}
```

Por último, se definen los métodos *onSuccess* y *successCode*, los heredados de *PostController*. El primero devuelve la URI a la que redirigirá en caso de éxito. Por defecto redirigirá a “*URI declarada en @RequestMapping en la cabecera del controlador*”/*list.do*, tal como viene dado en el código *CREATE_REDIRECT_VIEW_NAME* de la clase *Codes*. Es posible redefinir este método en nuestro controlador para redirigir a una URI distinta. El segundo método devuelve la cadena “*create.success.message*”. Es el código del mensaje de internacionalización que se mostrará cuando se cree con éxito una entidad. Está definido en los ficheros “*/resources/messages/messages.properties*” y “*/resources/messages/messages_es.properties*”.

Se define también el método *crudAction*, que devuelve la cadena “*creating*”. Esta cadena se añade al modelo con la clave “*crudAction*”. Gracias a esta cadena podremos comprobar en la vista qué tipo de operación estamos llevando a cabo.

```
@Override  
protected String onSuccess() {  
    return Codes.CREATE_REDIRECT_VIEW_NAME;  
}  
  
@Override  
protected String successCode() {  
    return Codes.CREATE_SUCCESS_CODE;  
}  
  
@ModelAttribute("crudAction")  
public String crudAction() {  
    return Codes.CRUD_ACTION_CREATING;  
}
```

Una vez descritos los métodos de *AbstractCreateController*, vamos a ver qué métodos debe definir el usuario en el controlador que cree, y qué métodos puede redefinir.

Al extender la clase *AbstractCreateController*, obligatoriamente el desarrollador deberá implementar los métodos *authorize*, por la interfaz *Authorizable*, y *view*, por *BaseController*. El primero es llamado por el método *authorize* de la clase *Authorization*, que a su vez es llamado por el método *authorize* de *SecurityHandler*, que a su vez es llamado en el método *get* de *GetController*, por lo que, en definitiva, se hace uso de él al obtener el formulario. Recibe el objeto a crear y la cuenta del usuario autenticado. Debe devolver *true* si el usuario está autorizado para acceder al formulario, y *false* si no lo está. En caso de que no lo esté, devolverá un error HTTP 403.

Por su parte, el método *view* devuelve el nombre de la vista declarada en un fichero *tiles.xml* que devolverá este controlador. Es llamado por *setView* de *BaseController*, que establece el valor del atributo *currentView*. De esta manera, el nombre de la vista quedará siempre accesible desde cualquier parte del controlador.

Otros métodos cuya implementación no es obligatoria, pero en algunos casos de uso necesaria, son *getObject* y *onSuccess*. El primero puede ser interesante redefinirlo si queremos obtener el objeto, con el que vamos a trabajar, a partir de un método distinto a *create* del servicio, y *onSuccess* para cambiar la URI por defecto a la que nos redirigirá en caso de éxito.



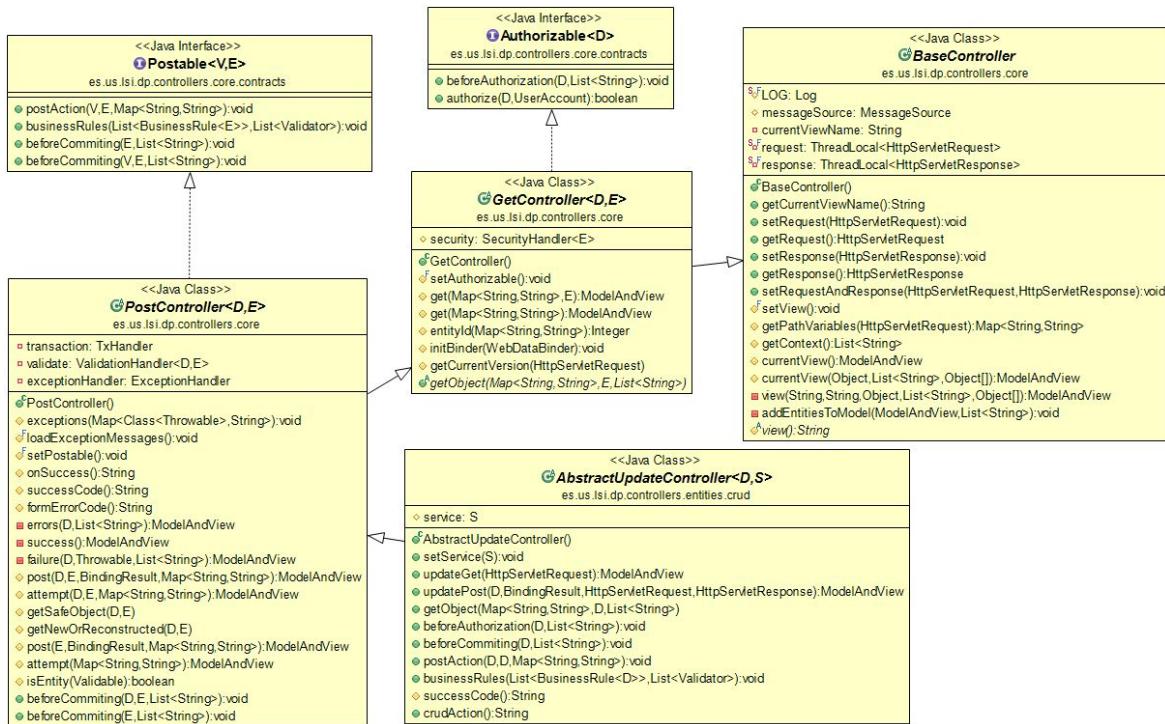
La entidad con la que trabajamos se añade al modelo con la clave “*modelObject*”. Si queremos añadir más objetos al modelo, nuestro controlador tendrá que implementar la interfaz *AddsToModel*.

5.3.2 AbstractUpdateController

Los controladores que vayan destinados a la actualización de entidades, deben extender a *AbstractUpdateController*. Debe recibir como parámetros genéricos una clase que extienda a *DomainObject*, es decir, una entidad o un objeto *form*, y el servicio que trabaja con dicha clase. Además de extender a esta clase, deben tener las etiquetas *@Controller*, que recibe como argumento un identificador único del controlador, y *@RequestMapping*, recibiendo la primera parte de la URI a la que responderá este controlador. Por convenio, la primera parte estará compuesta por el nombre de la entidad con la que trabaja el controlador y el rol que puede acceder a la vista. Sirva como ejemplo la cabecera de la clase *UpdateController* de *Folder*, del proyecto Acme-Barter:

```
@Controller("folderUpdateAdministrator")
@RequestMapping("folder/administrator")
public class UpdateController extends AbstractUpdateController<Folder,
    FolderService> {
```

Vamos a proceder a describir el funcionamiento de *AbstractUpdateController*, para entender cómo se gestionan los casos de uso de actualización de entidades. Esta clase es muy parecida a *AbstractCreateController*, por lo que encontrará varias referencias al apartado en el que se describe dicha clase.



En primer lugar, al igual que en *AbstractCreateController*, tenemos el atributo `protected service`, lo que nos va a permitir acceder desde nuestro controlador al servicio de nuestra entidad.

```
protected S service;
```

Seguidamente tenemos los métodos *updateGet* y *updatePost*. El propósito de ambos es idéntico a *createGet* y *createPost* de *AbstractCreateController*. La principal diferencia son las URLs a las que responderá este controlador. Como podemos ver en las etiquetas `@RequestMapping`, la parte final de la URI viene representada por los códigos *UPDATE_MAPPING_VALUE* y *UPDATE_MAPPING_VALUE_PARAMS*. Ambos responderán a las URLs que empiecen como se indica en la etiqueta `@RequestMapping` de la cabecera del controlador y que terminen en “*/update.do*” o “*/{context}/update.do*”.

```
@RequestMapping(value = {Codes.UPDATE_MAPPING_VALUE,
    Codes.UPDATE_MAPPING_VALUE_PARAMS }, method = RequestMethod.GET)
public ModelAndView updateGet(@final HttpServletRequest request) {
    return get(getPathVariables(request));
}

@RequestMapping(value = {Codes.UPDATE_MAPPING_VALUE,
    Codes.UPDATE_MAPPING_VALUE_PARAMS }, method = RequestMethod.POST)
public ModelAndView updatePost(@ModelAttribute(Codes.MODEL_OBJECT_NAME) @final D
    domainObject, @final BindingResult bindingResult, @final HttpServletRequest
    request, @final HttpServletResponse response) {
    setRequestAndResponse(request, response);
    return post(domainObject, bindingResult,
        getPathVariables(request));
}
```

El método *getObject* llama por defecto al método *findById* del servicio con el que trabaja el controlador. El identificador se obtiene a partir de la posición 0 del contexto. Es decir, que el contexto debe recibir siempre en primer lugar el identificador de la entidad que vamos a actualizar. Por ejemplo, dada la siguiente URI: “*folder/administrator/5,2,mas_parametros /update.do*”, se buscará la entidad cuyo *id* es 5 para actualizarla.

Podemos evitar este comportamiento redefiniendo el método *getObject* en nuestro controlador. Esto es útil, por ejemplo, si no queremos obtener la entidad que vamos a actualizar a partir de un identificador, o bien si queremos encontrar la entidad usando otro método de nuestro servicio.

```
@Override
public D getObject(@final Map<String, String> pathVariables, @final D entity,
    List<String> context) {
    return service.findById(entityId(pathVariables));
}
```

Los siguientes cuatro métodos se explicaron en el apartado *AbstractCreateController*. Las diferencias son que el método *beforeAuthorization* llama a *beforeUpdating*, *beforeCommiting* a *beforeCommitingUpdate*, *postAction* al método *update* y *afterCommitingUpdate*, y *businessRules* llama a *updateBusinessRules* del servicio.

Al igual que en *AbstractCreateService*, *beforeAuthorization* es heredado de la interfaz *Authorizable*, y los demás métodos son heredados de *Postable*. Todos son llamados en el método *post* de *PostController*, excepto *postAction*, que es llamado desde *attempt* de *PostController*.

```
@Override  
public void beforeAuthorization(final D domainObject, List<String> context) {  
    service.beforeUpdating(domainObject, context);  
}  
  
@Override  
public void beforeCommiting(D entity, List<String> context) {  
    service.beforeCommitingUpdate(entity, context);  
}  
  
@Override  
public void postAction(final D domainObject, final D duplicatedDomainObject,  
final Map<String, String> pathVariables) {  
    int id = service.update(domainObject);  
    service.afterCommitingUpdate(id);  
}  
  
@Override  
public void businessRules(final List<BusinessRule<D>> rules, final  
List<Validator> validators) {  
    service.updateBusinessRules(rules, validators);  
}
```

Los dos últimos métodos son *successCode* y *crudAction*. El funcionamiento y el propósito de ambos es idéntico al explicado en *AbstractCreateController*. El método *successCode* devuelve el código “*UPDATE_SUCCESS_CODE*”, que representa la cadena “*update.success.message*”. Es el código del mensaje de internacionalización que se mostrará cuando se cree con éxito una entidad. Está definido en los ficheros “/resources/messages/messages.properties” y “/resources/messages/messages_es.properties”.

```
@Override  
protected String successCode() {  
    return Codes.UPDATE_SUCCESS_CODE;  
}  
  
@ModelAttribute("crudAction")  
public String crudAction() {  
    return Codes.CRUD_ACTION_UPDATING;
```

}

Una vez descritos los métodos de *AbstractUpdateController*, vamos a ver qué métodos debe definir el usuario en el controlador que cree, y qué métodos puede redefinir.

Al extender la clase *AbstractUpdateController*, el desarrollador deberá definir obligatoriamente los métodos *authorize* y *view*. La utilidad y el funcionamiento de ambos es idéntico al descrito en *AbstractCreateController*.

Otros métodos, cuya implementación no es obligatoria, pero en algunos casos de uso es necesaria, son *onSuccess* y *getObject*. El primero (que esta vez no está implementado en *AbstractUpdateController*, si no en *PostController*), nos permitirá cambiar la URI por defecto a donde redirigirá la aplicación en caso de éxito al actualizar (por defecto hará que la aplicación redirija a “*URI declarada en @RequestMapping en la cabecera del controlador*”/list.do).

Como hemos explicado antes, el método *getObject* debe ser necesariamente redefinido si no necesitamos un identificador para encontrar la entidad a actualizar o si vamos a usar otro método para encontrarla. A continuación mostramos un ejemplo de redefinición del método *getObject*, que devuelve un objeto tipo *form* para actualizar la cuenta de usuario a partir del principal:

```
@Override  
public UserAccountForm getObject(Map<String, String> pathVariables,  
UserAccountForm entity, List<String> context) {  
    return service.convertToForm(SignInService.getPrincipal());  
}
```



La entidad con la que trabajamos se añade al modelo con la clave “*modelObject*”. Si queremos añadir más objetos al modelo, nuestro controlador tendrá que implementar la interfaz *AddsToModel*.

5.3.3 AbstractDeleteController

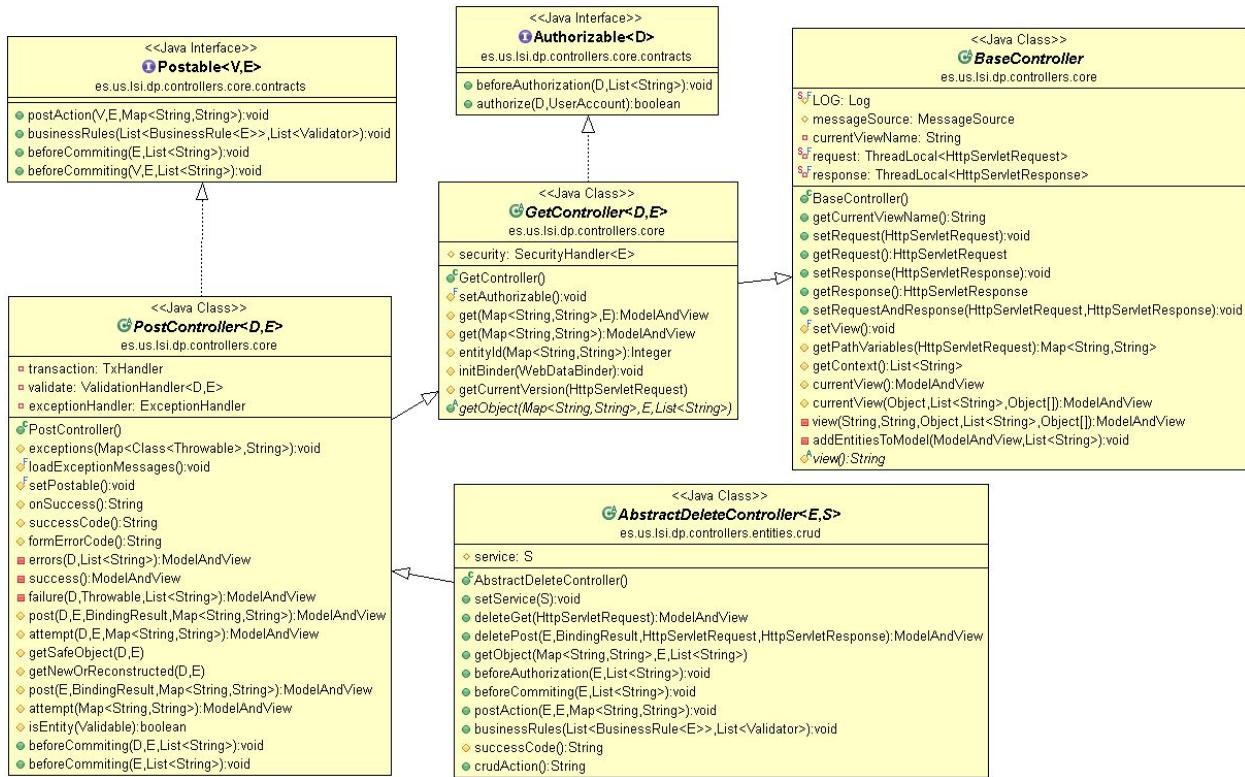
Los controladores destinados a la eliminación de entidades, deben extender a *AbstractDeleteController*. Debe recibir como parámetros genéricos una clase que extienda a *DomainObject*, es decir, una entidad o un objeto *form*, y el servicio que trabaje con dicha clase. Además de extender a esta clase, deben tener las etiquetas *@Controller*, que recibe como argumento un identificador único del controlador, y *@RequestMapping*, recibiendo la primera parte de la URI a la que responderá este controlador. Por convenio, la primera parte estará compuesta por el nombre de la entidad con la que trabaja el controlador y el rol que puede acceder a la vista. Sirva como ejemplo la cabecera de la clase *DeleteController* de *Folder* de Acme-Barter:

```

@Controller("folderDeleteAdministrator")
@RequestMapping("folder/administrator")
public class DeleteController extends AbstractDeleteController<Folder,
FolderService> {

```

Vamos a proceder a describir el funcionamiento de *AbstractUpdateController*, para entender cómo se gestionan los casos de uso de eliminación de entidades. Esta clase es muy parecida a *AbstractCreateController* y *AbstractUpdateController*, por lo que encontrará referencias a ambos apartados.



En primer lugar, al igual que en *AbstractCreateController* y *AbstractUpdateController*, tenemos el atributo *protected service*, lo que nos va a permitir acceder desde nuestro controlador al servicio de nuestra entidad.

```

protected S service;

```

Seguidamente tenemos los métodos *deleteGet* y *deletePost*. El primero se encarga de generar la vista donde se mostrará el objeto a eliminar, llamando al método *get* heredado de *GetController*. El segundo, se llamará cuando se ejecute la acción HTTP POST, y devolverá la vista informando del resultado de la operación. Para ello, llama al método *post* heredado de *PostController*.

Los métodos `deleteGet` y `deletePost`, tal como se especifica en sus etiquetas `@RequestMapping`, responden a las URIs que comienzan como se especifica en la etiqueta `@RequestMapping` de la cabecera de la clase, finalizando con “`delete.do`” o “`/{context}/delete.do`”. Si la petición HTTP es GET, se llamará a `deleteGet`, por el contrario, si el método es POST, se llamará al `deletePost`.

```
@RequestMapping(value = {Codes.DELETE_MAPPING_VALUE,
    Codes.DELETE_MAPPING_VALUE_PARAMS}, method = RequestMethod.GET)
public ModelAndView deleteGet(@final HttpServletRequest request) {
    return get(getPathVariables(request));
}

@RequestMapping(value = {Codes.DELETE_MAPPING_VALUE,
    Codes.DELETE_MAPPING_VALUE_PARAMS}, method = RequestMethod.POST)
public ModelAndView deletePost(@ModelAttribute(Codes.MODEL_OBJECT_NAME)
    final E domainObject, @final BindingResult bindingResult,
    final HttpServletRequest request, final HttpServletResponse response) {
    setRequestAndResponse(request, response);
    return post(domainObject, bindingResult,
        getPathVariables(request));
}
```

El siguiente método, `getObject`, presenta una finalidad y comportamiento idéntico al especificado en `AbstractCreateController` y `AbstractUpdateController`, por lo que omitiremos su descripción.

Los siguientes cuatro se explicaron en el apartado `AbstractCreateController` y `AbstractUpdateController`. Las diferencias son que el método `beforeAuthorization` llama a `beforeDeleting`, `beforeCommitting` a `beforeCommittingDelete`, `postAction` al método `delete` del servicio y `afterCommittingDelete`, y `businessRules` llama a `deleteBusinessRules`.

Al igual que en `AbstractCreateController` y `AbstractUpdateController`, `beforeAuthorization` es heredado de la interfaz `Authorizable`, y los demás de `Postable`. Todos son llamados en el método `post` de `PostController`, excepto `postAction`, que es llamado desde el método `attempt` de `PostController`.

```
@Override
public void beforeAuthorization(@final E domainObject, List<String>
    context) {
    service.beforeDeleting(domainObject, context);
}

@Override
public void beforeCommitting(E domainObject, List<String> context) {
    service.beforeCommittingDelete(domainObject, context);
}

@Override
```

```

public void postAction(final E domainObject, final E
duplicatedDomainObject, final Map<String, String> pathVariables) {
    service.delete(domainObject);
    service.afterCommittingDelete(domainObject.getId());
}

@Override
public void businessRules(final List<BusinessRule<E>> rules, final
List<Validator> validators) {
    service.deleteBusinessRules(rules, validators);
}

```

Los dos últimos métodos son *successCode* y *crudAction*. El funcionamiento y el propósito de ambos es idéntico al explicado en *AbstractCreateController* y *AbstractUpdateController*. El método *successCode* devuelve el código “*DELETE_SUCCESS_CODE*”, que representa la cadena “*delete.success.message*”. Es el código del mensaje de internacionalización que se mostrará cuando se elimina con éxito una entidad. Está definido en los ficheros “/resources/messages/messages.properties” y “/resources/messages/messages_es.properties”

```

@Override
protected String successCode() {
    return Codes.DELETE_SUCCESS_CODE;
}

@ModelAttribute("crudAction")
public String crudAction() {
    return Codes.CRUD_ACTION_DELETING;
}

```

Cuando el desarrollador quiera crear un controlador destinado a la eliminación de un tipo de entidad en concreto, debe extender a la clase *AbstractDeleteController*, indicando la entidad y el servicio con los que va a trabajar el controlador. Al hacerlo, el desarrollador deberá definir obligatoriamente los métodos *authorize* y *view*. La utilidad y el funcionamiento de ambos es idéntico al descrito en *AbstractCreateController*.

Otros métodos, cuya implementación no es obligatoria, pero en algunos casos de uso es necesaria, son *onSuccess* y *getObject*. El primero (que esta vez no está implementado en *AbstractDeleteController* si no en *PostController*), nos permitirá cambiar la URI por defecto a donde redirigirá la aplicación en caso de éxito al eliminar (por defecto hará que la aplicación redirija a “*URI declarada en @RequestMapping en la cabecera del controlador*”/list.do).

- i** La entidad con la que trabajamos se añade al modelo con la clave “*modelObject*”. Si queremos añadir más objetos al modelo, nuestro controlador tendrá que implementar la interfaz *AddsToModel*.

5.3.4 AbstractListController

Controlador a utilizar cuando se requiere llevar a cabo el listado de alguna entidad. Para ello se debe extender de la clase *AbstractListController*. Se necesita pasar como parámetros genéricos la entidad en cuestión y el servicio que involucre a esta entidad y donde se encuentren los métodos necesarios para realizar el listado de la misma. También será necesario que el controlador que extienda de este, tenga las etiquetas *@Controller* y *@RequestMapping*. La primera de ellas para dar un nombre a este controlador y que no haya errores en caso de haber dos controladores de listado de una misma entidad. La segunda para indicar la URI a la que debería responder este controlador cuando le envíen una petición. Tómese como referencia la siguiente cabecera:

```
@Controller("EntityListController")
@RequestMapping("entity/authorizedRole")
public class ListController extends AbstractListController<Entity,
EntityService>{
```

Destacar que a la URI que se ponga en la etiqueta *@RequestMapping* se le va a añadir */list.do*, al tratarse de un controlador de listado.

A continuación se detallan los métodos de los que dispone dicho controlador:

- *list*: recibe como parámetro la petición HTTP. Además tiene como *@RequestMapping* las correspondientes al listado, estas son */{context}/list* y */list*. Este método se encarga de devolver un *ModelAndView*. En él tan solo se llama al método *setRequest*, para hacer un set del atributo *request* del controlador padre, pasándole la *HTTPRequest* que recibe como parámetro. Finalmente, devuelve como *ModelAndView* el valor que retorna el método *currentView* del controlador padre (*BaseController*).
- *crudAction*: no recibe ningún tipo de parámetro. Tiene la anotación *@ModelAttribute("crudAction")*. Se encarga de injectarle a este atributo del modelo el valor correspondiente. En este caso la constante decidida para los casos en los que hay un listado: “*listing*”.
- *view*: no recibe parámetro alguno. Además no se implementa en esta clase, puesto que queda a la disposición de que el usuario que quiera hacer uso de un controlador de listado, sobreesciba este método e indique cuál es el nombre de la vista.



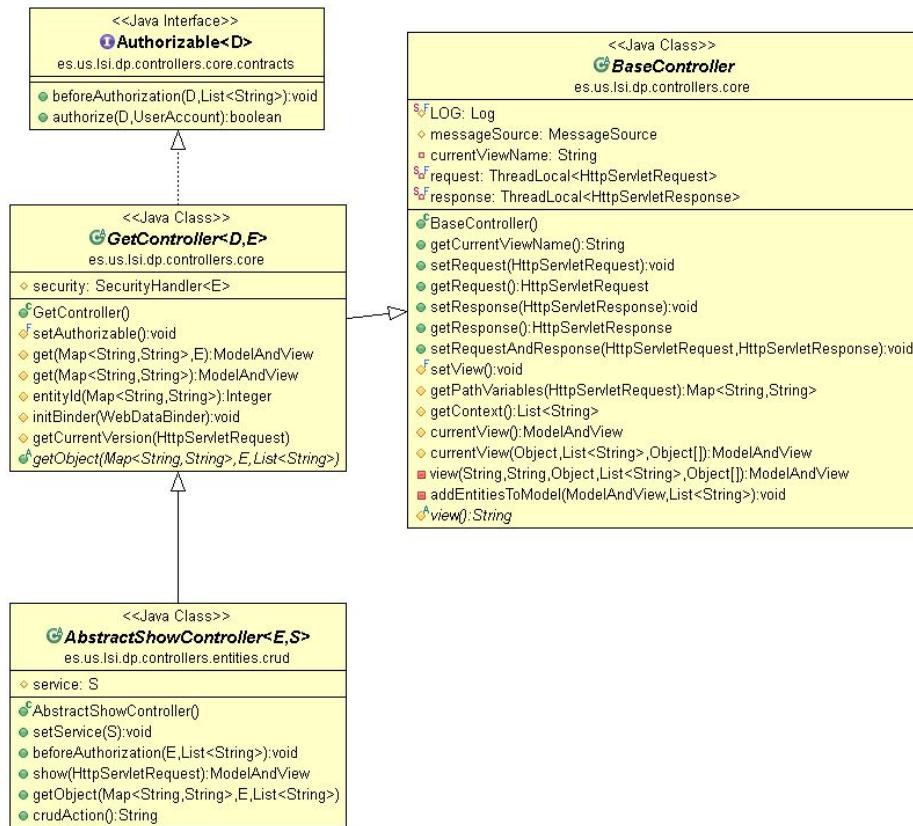
Si se quiere obtener una página en el controlador de listado, que no sea la página por defecto, se debe sobreescribir dicho método en el controlador que extienda de *AbstractListController*. El método en cuestión a sobreescribir es *Page<Barter> getPage*. Este método recibe como parámetros la página, el criterio de búsqueda y el contexto. A partir de los mismos el usuario debe poder seleccionar la página que crea conveniente de sus servicios.

5.3.5 AbstractShowController

Para aquellos casos de uso en los que necesitas mostrar información sobre una entidad, debemos extender a la clase *AbstractShowController*. Este recibe como parámetros genéricos la entidad y el servicio con el que va a trabajar el controlador. Además, deben tener las etiquetas `@Controller`, que recibe como argumento un identificador único del controlador, y `@RequestMapping`, recibiendo la primera parte de la URI a la que responderá este controlador. Por convenio, la primera parte estará compuesta por el nombre de la entidad con la que trabaja el controlador y el rol que puede acceder a la vista. Sirva como ejemplo la cabecera de la clase *ShowController* de *SocialIdentity* de Acme-Barter:

```
@Controller("SocialIdentityShowController")
@RequestMapping("home/user/socialIdentity")
public class ShowController extends
AbstractShowController<SocialIdentity, SocialIdentityService>
```

Describiremos a continuación el funcionamiento de la clase *AbstractShowController*. En primer lugar, presentamos el diagrama de clases UML en el que se ve implicada esta clase.



Notamos que *AbstractShowController*, a diferencia de los controladores anteriores, extiende de *GetController*. Esto es porque en este tipo de controladores no haremos peticiones HTTP POST.

AbstractShowController permite acceder en todo momento al servicio con el que trabaja por medio del atributo con modificador de acceso protected llamado service:

```
protected S service;
```

El método principal de este controlador es *show*. Este método, tal como vemos en la etiqueta *@RequestMapping*, responde a las peticiones HTTP GET dirigidas a las URLs que empiezan como se definió en la equita *@RequestMapping* de la cabecera de la clase, y terminando o bien en “/show.do” o bien “/{context}/show.do”. Recibe un objeto de tipo *HttpServletRequest*, llamado *request*. El método devuelve una vista, obtenida a partir de una llamada al método *get* de *GetController*. A este método se le pasa como parámetros las path variables, obtenidas a partir del objeto *request*.

```
@RequestMapping(value = {Codes.SHOW_MAPPING_VALUE,
    Codes.SHOW_MAPPING_VALUE_PARAMS}, method = RequestMethod.GET)
public ModelAndView show(@final HttpServletRequest request) {
    return get(getPathVariables(request));
}
```

A continuación, tenemos una implementación vacía del método *beforeAuthorization*. Esto es así porque en ningún momento se realizarán operaciones sobre el objeto a mostrar.

```
public void beforeAuthorization(E entity, List<String> context) {
}
```

El método *getObject* es el encargado de obtener el objeto que se mostrará en la vista. Como ya sabemos, será el método *get* de *GetController* quien acceda a este método. Por defecto, *getObject* obtiene la entidad a partir del id especificado en la primera posición del contexto. Es posible redefinir este método para obtener el objeto a enviar a la vista por otros medios.

```
@Override
public E getObject(@final Map<String, String> pathVariables, @final E
entity, List<String> context) {
    return service.findById(entityId(pathVariables));
}
```

Por último, tenemos el método *crudAction*. Este devuelve, por medio del atributo estático *CRUD_ACTION_SHOWING* de la clase *Codes*, la cadena “showing”. Esto, gracias a la etiqueta *@ModelAttribute*, se añadirá siempre al modelo bajo la clave *crudAction*, de manera que desde la vista podremos saber qué tipo de operación CRUD está llevando a cabo esa

vista. Esto permitirá, por ejemplo, ocultar el botón de “enviar” cuando el formulario solo debe mostrar información.

```
@ModelAttribute("crudAction")
public String crudAction() {
    return Codes.CRUD_ACTION_SHOWING;
}
```

Cuando el desarrollador quiera crear un controlador destinado a mostrar información de un objeto en concreto, debe extender la clase *AbstractShowController*, indicando la entidad y el servicio con los que va a trabajar el controlador. Al hacerlo, el desarrollador deberá definir obligatoriamente los métodos *authorize* y *view*. La utilidad y el funcionamiento de ambos es idéntico al descrito en los apartados anteriores.

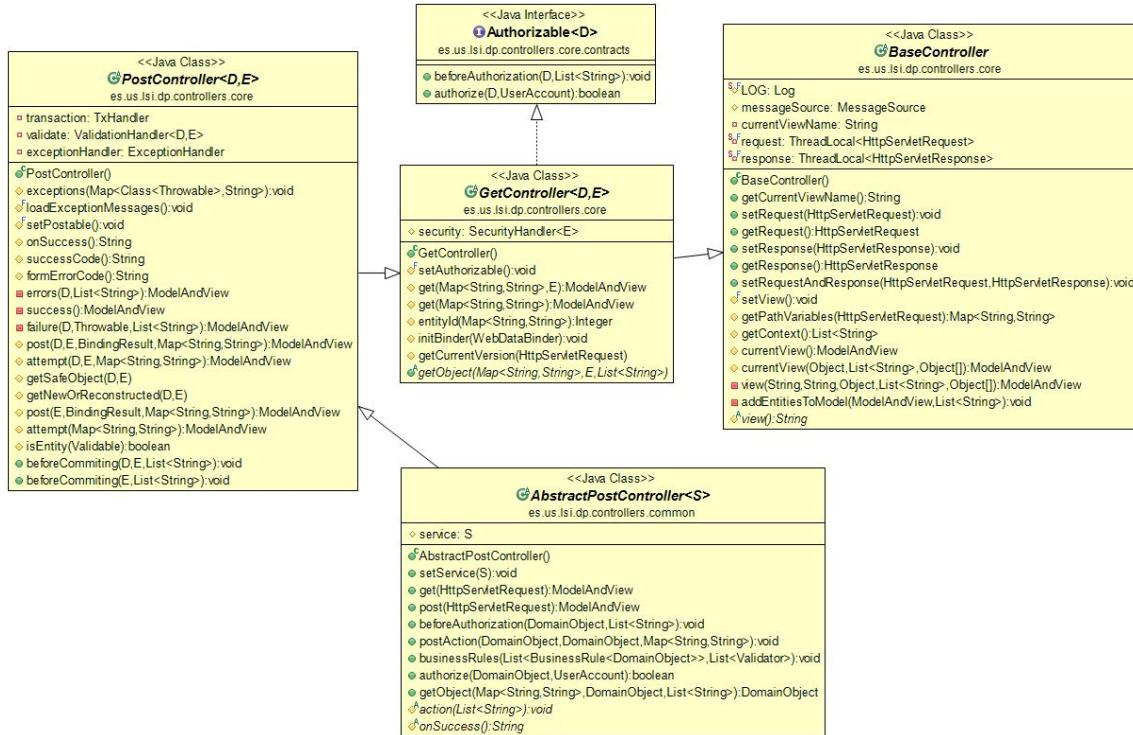
- i** La entidad con la que trabajamos se añade al modelo con la clave “modelObject”. Si queremos añadir más objetos al modelo, nuestro controlador tendrá que implementar la interfaz *AddsToModel*.

5.3.6 AbstractPostController para entidades

La clase *AbstractPostController* permite al desarrollador realizar operaciones POST de casos de uso más complejos que no se contemplan en *AbstractCreateController*, *AbstractUpdateController* o *AbstractDeleteController*. A continuación presentamos distintos casos en los que este controlador puede ser útil:

- Ejecutar un procedimiento, que realice una operación sobre una serie de objetos persistidos en la base de datos. Ejemplo: proyecto Acme-Barter, clase *CancelBartersController*.
- Actualizar un atributo de tipo boolean de una entidad. Ejemplo: proyecto Acme-Barter, *CancelMatchController*.
- Eliminar una imagen de un atributo de tipo colección de una entidad. Ejemplo: proyecto Acme-Six-Pack, *DeletePictureController*.

Para hacer uso de este controlador, hay que extender la clase *AbstractPostController*, que recibe como parámetros genéricos un servicio de una entidad. La cabecera de la clase debe tener las etiquetas *@Controller*, que debe recibir como parámetro una cadena que identifique al controlador de forma única, y *@RequestMapping*, donde se debe indicar la URI a la que responderá este controlador. Por convenio, esta seguirá la siguiente estructura: “*nombre_entidad implicada/rol/operación_a_realizar*”.



En primer lugar, la clase `AbstractPostController` tiene un atributo con modificador de acceso `protected` llamado `service`, desde el cual podemos acceder al servicio con el que estamos trabajando.

```
protected S service;
```

Se implementan a continuación los métodos `get` y `post`. Ambos poseen la etiqueta `@RequestMapping`, especificando que el primero responde a peticiones HTTP GET y el segundo a HTTP POST. El método `get` recibe un objeto de tipo `HttpServletRequest`, llamado `request`, y devuelve una vista, obtenida a partir del método `currentView` de `BaseController`.

De forma parecida, el método `post` llama a `attempt`, que ejecuta la acción `post`, tal como vimos en `PostController`, y devuelve una vista de éxito o una vista mostrando un fallo en caso de que la transacción no se complete con éxito. Este método, al igual que `get`, recibe un objeto de tipo `HttpServletRequest`, a partir del cual se obtienen las path variables, que se le pasa al método `attempt`.

```
@RequestMapping(method = RequestMethod.GET)
public ModelAndView get(@final HttpServletRequest request) {
    setRequest(request);
    return currentView();
}

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

```

public ModelAndView post(final HttpServletRequest request) {
    setRequest(request);
    return attempt(getPathVariables(request));
}

```

Se dan definiciones por defecto a los métodos *beforeAuthorization*, *businessRules*, *authorize* y *getObject*, de manera que estos métodos estarán vacíos o devolverán valores por defecto. Esto es así porque este controlador no trabaja con entidades, por lo que en ningún momento van a ser llamados.

```

@Override
public void beforeAuthorization(DomainObject object, List<String>
context) {
}

@Override
public void businessRules(final List<BusinessRule<DomainObject>> rules,
final List<Validator> validators) {
}

@Override
public boolean authorize(final DomainObject domainObject, final
UserAccount principal) {
    return true;
}

@Override
public DomainObject getObject(final Map<String, String> pathVariables,
final DomainObject entity, List<String> context) {
    return null;
}

```

Se redefine el método *postAction* heredado por *PostController*, de forma que llama al método *action*, declarado como abstracto en *AbstractPostController*, y que el desarrollador debe redefinir en la clase que la extienda. En el método *action* se ha de especificar la acción a realizar, llamando a un método del servicio. Recibe, además, el contexto, de manera que el desarrollador puede usarlo para obtener información de cara a la acción a ejecutar.

```

@Override
public void postAction(final DomainObject object, final DomainObject
entity, final Map<String, String> pathVariables) {
    action(ContextParser.parse(pathVariables));
}
protected abstract void action(List<String> context);

```

Por último, se define el método `onSuccess` como abstracto. El desarrollador debe redefinirlo en su clase, de forma que devuelva la URI donde redirigirá en caso de éxito.

```
@Override  
protected abstract String onSuccess();
```

Con todo, el desarrollador debe implementar los siguientes métodos al hacer uso de `AbstractPostController`:

- `action`: recibe el contexto y no devuelve nada. En este método se debe llamar al método del servicio que ejecute la acción POST.
- `onSuccess`: devuelve una cadena que representa la URI donde redirigirá la aplicación en caso de éxito al realizar la operación indicada en `action`.
- `view`: devuelve el nombre de la vista con la que trabajará este controlador. Es heredado de `BaseController`.

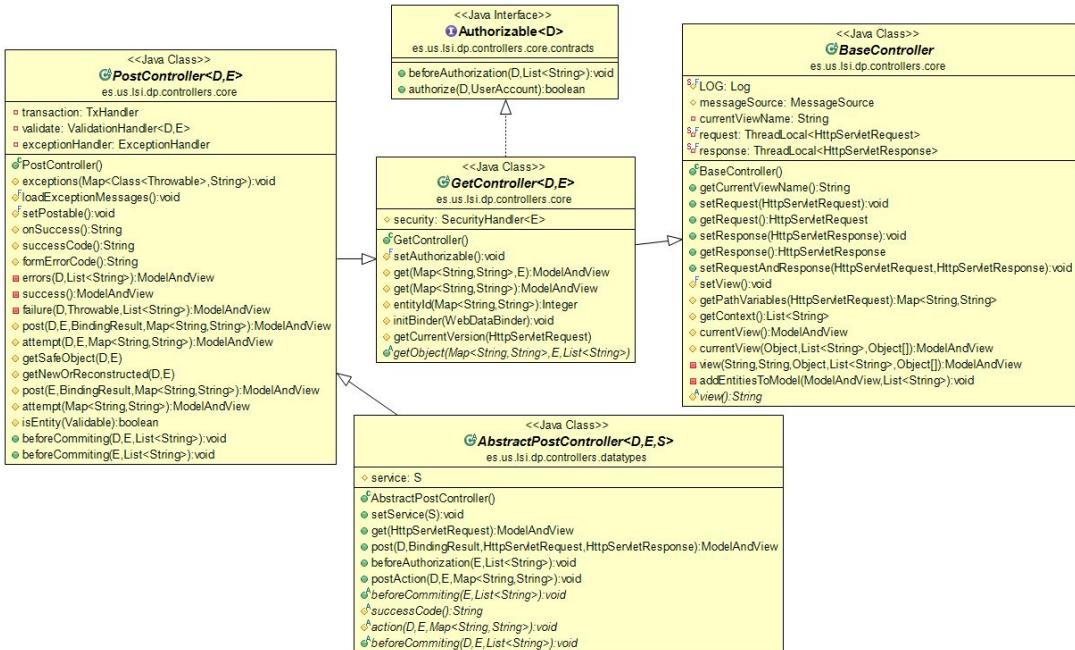
Además, el desarrollador puede redefinir el método `successCode` de `PostController`, para cambiar el mensaje de internacionalización usado en caso de éxito.

5.3.7 AbstractPostController para datatypes

Acabamos de ver la clase `AbstractPostController`, que nos permitía implementar casos de uso que no se contemplaban en los casos de uso anteriores para trabajar con entidades. Para operar con datatypes, podemos utilizar la clase `AbstractPostController` del paquete `es.us.lsi.dp.controllers.datatypes`. El uso de esta clase es similar al de `AbstractPostController` para entidades, pero en este tenemos la posibilidad de aplicar reglas de negocio a la entidad a la que pertenece el `datatype`, utilizar los métodos `authorize`, `getObject`, y `beforeCommitting`.

Para usarlo, nuestro controlador debe extender de `AbstractPostController` para `datatypes`, pasando como parámetros genéricos el `datatype`, la entidad a la que pertenece el `datatype`, y el servicio de esta entidad. La cabecera de la clase debe llevar las etiquetas `@Controller`, recibiendo como argumento un identificador único para la clase, y `@RequestMapping`, que recibe la URI a la que responderá este controlador. El formato será: `"nombre_datatype/rol/{0}/operación"`. `{0}` indica que la URI espera recibir un ID de una entidad. En este caso, es obligatorio que reciba un ID, ya que siempre se obtendrá la entidad a partir de su ID en este tipo de controladores.

```
@Controller("creditCardCustomerCreate")  
@RequestMapping("creditCard/customer/{0}/create")  
public class CreateController extends AbstractPostController<CreditCard,  
Customer, CustomerService> {
```



Al igual que en `AbstractPostController` para entidades, tenemos un atributo protected que nos permite acceder al servicio de la entidad que opera con el datatype:

```
protected S service;
```

Tenemos un método `get` y `post`. El método `get` responde a las peticiones HTTP GET dirigidas a la URI especificada en la cabecera de la clase. Recibe un objeto de tipo `HttpServletRequest` llamado `request`, a partir del cual se obtienen las path variables. Se obtiene la entidad cuyo ID fue indicado en el método en las path variables, llamando al método `findById` del servicio. Devolveremos la vista llamando al método `get` de `GetController`, que, esta vez sí recibe la entidad como parámetros. Recordamos que, en todos los casos anteriores, únicamente recibía las path variables, tal como se explicó en el apartado 5.2.4.

El método `post` responde a peticiones HTTP POST a la misma URI que el método `get`. Recibe el datatype que el usuario ha enviado en el formulario, un objeto de tipo `BindingResult`, otro `HttpServletRequest` y `HttpServletResponse`. Lo primero que hace es obtener el ID de la entidad con cuyo datatype estamos trabajando, a partir del objeto `HttpServletRequest` que recibimos como parámetro. A partir de ese ID, obtiene la entidad, llamando al método `findById` del servicio. Finalmente, devuelve la vista generada por el método `post` de `PostController`. En esta ocasión, le pasa como parámetros el datatype, la entidad, el objeto `BindingResult`, y las `pathVariables`.

i Tenga en cuenta que es obligatorio que, en este caso, la URI reciba el ID de la entidad a la que pertenece el datatype que queremos modificar. Por ejemplo,

si se trata de una tarjeta de crédito, y es un atributo de Customer, en la URI debemos indicar el ID del Customer cuya tarjeta de crédito queremos modificar.

```
@RequestMapping(method = RequestMethod.GET)
public ModelAndView get(@final HttpServletRequest request) {
    Map<String, String> pathVariables;
    E entity;

    pathVariables = getPathVariables(request);

    entity = service.findById(entityId(pathVariables));

    return get(pathVariables, entity);
}

@RequestMapping(method = RequestMethod.POST)
public ModelAndView post(@ModelAttribute(Names.MODEL_OBJECT_NAME) @final
D datatype, @final BindingResult bindingResult, @final HttpServletRequest
request,
        @final HttpServletResponse response) {
    Map<String, String> pathVariables;
    E entity;

    pathVariables = getPathVariables(request);

    entity = service.findById(entityId(pathVariables));

    setRequestAndResponse(request, response);

    return post(datatype, entity, bindingResult, pathVariables);
}
```

Tenemos el método *beforeAuthorization* definido por defecto como vacío. Esto es porque no participa en ninguna etapa de este controlador (recordamos que en el método *get* de *GetController*, solo llamaba a *beforeAuthorization* si la entidad recibida como parámetro era nula). Hay un método *beforeCommitting* definido por defecto como vacío: el que recibe una entidad. Existe otro, que recibe un *datatype* y una entidad, que está declarado como abstracto, de forma que el desarrollador debe redefinirlo en su controlador.

El método *postAction* está implementado igual que en *AbstractPostController* para entidades: llama al método *action*, que recibe, en esa ocasión, un *datatype*, la entidad de ese *datatype* y las path variables. *Action* está definido en esta clase como abstracto.

Por último, se define como abstracto el método *successCode*, de forma que el desarrollador puede especificar el código del mensaje de internacionalización mostrado en caso de éxito al realizar la operación con este controlador.

```

@Override
public void beforeAuthorization(E object, List<String> context) {
}

@Override
public void postAction(final D object, final E entity, final Map<String,
String> pathVariables) {
    action(object, entity, pathVariables);
}

@Override
public void beforeCommitting(E entity, List<String> context) {
}

@Override
protected abstract String successCode();

protected abstract void action(D object, E entity, Map<String, String>
pathVariables);

public abstract void beforeCommitting(D entityOrDatatype, E entity,
List<String> context);

```

A modo de resumen, presentamos el siguiente listado de métodos que el desarrollador debe implementar al usar *AbstractPostController* del paquete *es.us.lsi.dp.controllers.datatypes*:

- *action*: recibe el *datatype* enviado desde el formulario, la entidad persistida en la base de datos relacionada con ese *datatype*, y las path variables. Desde este método, debemos llamar, por ejemplo, al método *update* del servicio, para actualizar la entidad, habiendo llamado previamente al método *set* de la entidad, de forma que quede establecido en esta el nuevo *datatype*.
- *beforeCommitting*: recibe el *datatype*, la entidad y el contexto. Desde este método podemos, por ejemplo, actualizar el atributo del *datatype* que hemos modificado en la entidad.
- *businessRules*: es posible aplicar reglas de negocio a la entidad que tiene al *datatype* como atributo. Recordamos que este método es llamado desde *post* de *PostController*, después de aplicar el *beforeCommitting*, por lo que de esta manera podemos aplicar reglas de negocio sobre el atributo *datatype* de la entidad. Por ejemplo, si tenemos un datatype representando la tarjeta de crédito de un Customer, nos interesaría tener una regla de negocio que comprobara que esta no ha expirado.
- *authorize*: ya conocemos este método. Recibe la entidad que usa al *datatype* y el usuario principal. Comprueba que este tiene permiso para operar sobre la entidad.
- *successCode*: código del mensaje de internacionalización mostrado en caso de éxito al ejecutar la acción especificada en este controlador.
- *onSuccess*: URI a la que debe redirigir en caso de éxito al ejecutar la acción especificada en este controlador.

- *getObject*: definido en *GetController*, recibe las path variables y la entidad que trabaja con el *datatype* que vamos a tratar. Debe devolver el *datatype* que vamos a modificar. Por ejemplo, si nuestro controlador se dedica a actualizar tarjetas de crédito, devolverá una nueva tarjeta de crédito, obtenida a partir de su constructor. Si vamos a actualizar la tarjeta de crédito de un *Customer*, llamaremos al método *getCreditCard* de este.
- *view*: devuelve el nombre de la vista que usa este controlador.

Para que sirva como aclaración, mostramos el código de la clase *CreateController* de *CreditCard* del proyecto Acme-Six-Pack.

```

@Controller("creditCardCustomerCreate")
@RequestMapping("creditCard/customer/{0}/create")
public class CreateController extends AbstractPostController<CreditCard,
Customer, CustomerService> {

    @Autowired
    private IsNotExpiredCreditCard isNotExpiredCreditCard;

    @Override
    public void businessRules(List<BusinessRule<Customer>> rules,
List<Validator> validators) {
        rules.add(isNotExpiredCreditCard);
    }

    @Override
    public boolean authorize(Customer domainObject, UserAccount
principal) {
        boolean authorized = false;

        authorized =
domainObject.getUserAccount().equals(principal);

        return authorized;
    }

    @Override
    protected String successCode() {
        return "creditCard.create.successful";
    }

    @Override
    protected void action(CreditCard object, Customer entity,
Map<String, String> pathVariables) {
        service.update(entity);
    }

    @Override
    public CreditCard getObject(Map<String, String> pathVariables,
Customer entity, List<String> context) {
        return new CreditCard();
    }
}

```

```

@Override
protected String view() {
    return "creditCard/create";
}

@Override
protected String onSuccess() {
    return "/profile/customer/show.do";
}

@Override
public void beforeCommitting(CreditCard entityOrDatatype, Customer
entity, List<String> context) {
    entity.setCreditCard(entityOrDatatype);
}
}

```

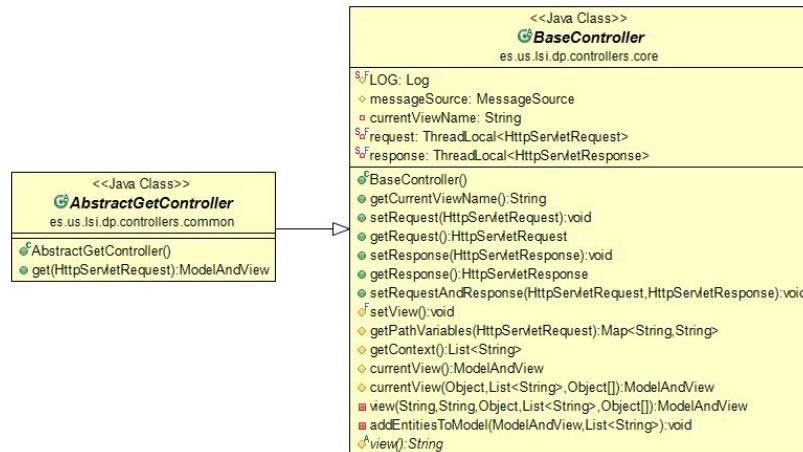
5.3.8 AbstractGetController

En caso de querer mostrar únicamente una vista para mostrar cualquier tipo de información (tenga o no que ver con una entidad o datatype), debemos heredar la clase *AbstractGetController*. Al hacer esto, no necesitamos especificar parámetros genéricos, y únicamente tendremos que redefinir el método *view* heredado de *BaseController*, especificando el nombre de la vista que usaremos. Como todos los controladores, debe tener presente las etiquetas *@Controller*, recibiendo un identificador único del controlador, y *@RequestMapping*, que recibe la URI a la que responderá este controlador.

```

@Controller("dashboardGetController")
@RequestMapping(value={"dashboard/administrator/list","dashboard/administ
rator/{context}/list"})
public class GetController extends AbstractGetController{

```



La clase *AbstractGetController*, implementa el método *get*. Este devuelve la vista llamando al método *currentView*.

5.3.9 Añadir atributos adicionales al modelo: interfaz *AddsToModel*

Si en algún controlador necesitáramos añadir atributos adicionales al modelo, por ejemplo, por queremos mostrar un menú desplegable en un formulario, dando la opción de elegir entre entidades persistidas en la base de datos, nuestro controlador tendrá que implementar la interfaz *AddsToModel*. Al hacer esto, debemos implementar el método *addToModel*, que recibe un objeto de tipo *Map<String, Object>* y el contexto. Debemos añadir en el mapa los atributos que deseemos que estén en el modelo. La clave será el nombre del atributo, y el valor el objeto a añadir.

En el siguiente ejemplo, estamos añadiendo una colección de actores para que el usuario pueda seleccionar en un dropdown el destinatario de un mensaje:

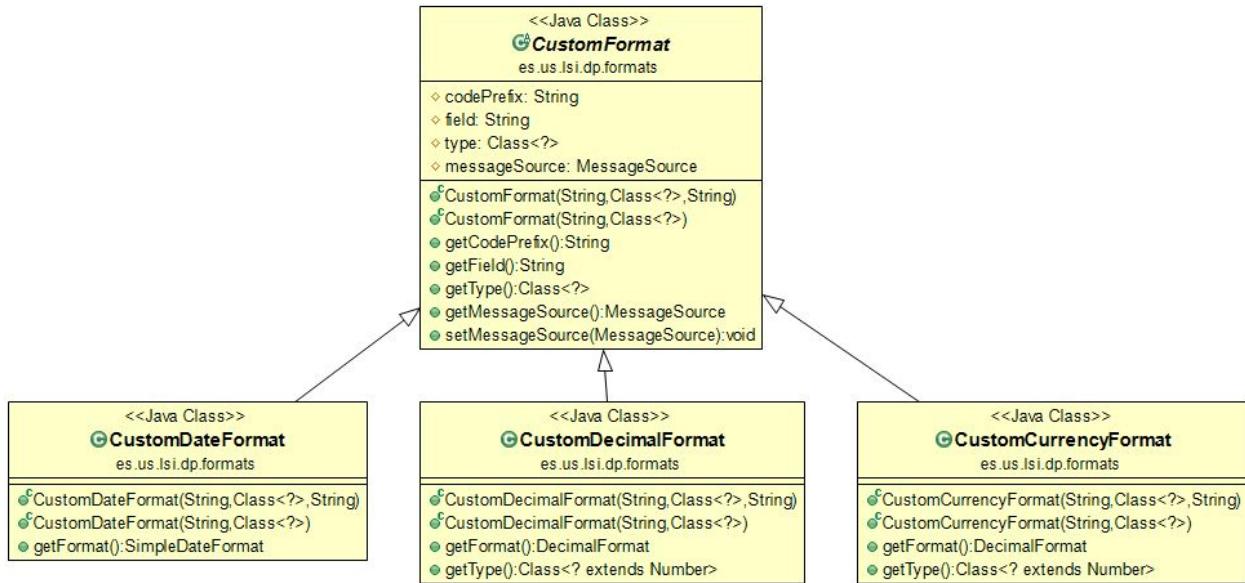
```
@Override  
public void addToModel(Map<String, Object> objects, List<String> context) {  
    Collection<Actor> actors;  
    actors = actorService.findAll();  
    objects.put("actors", actors);  
}
```

5.4. Formateadores y parseadores

Los formateadores y parseadores se usan para internacionalizar y adaptar al usuario los datos que provienen de la aplicación. La idea es, según el idioma con el que el usuario esté trabajando, mostrar ciertos tipos de valores con un formato u otro. Un claro ejemplo de esto son las fechas. Si la aplicación está en español, lo normal es que se muestre con el formato día/mes/año. En cambio, si está en inglés, es típico mostrarla con el formato mes/día/año. Esto mismo se pretende con los números decimales y las unidades monetarias. Esto se logra gracias a un formateador. De la misma manera, también se puede esperar que si el usuario tiene la aplicación en español, inserte la fecha con el formato día/mes/año, y si la tiene en inglés, en el formato mes/día/año. El trabajo de los parseadores es este, transformar la cadena del usuario en otra cadena que entienda la aplicación web. La misma idea se puede aplicar a números decimales y unidades monetarias.

Sprouts-Framework proporciona una arquitectura que permite hacer uso de una manera sencilla de formateadores y parseadores aplicados a distintos campos. Veamos cómo se implementan.

5.4.1 Implementación de formateadores y parseadores



Los formateadores y parseadores (de ahora en adelante, nos referiremos a ambos como formateadores) se definen mediante un objeto de una clase que extiende a *CustomFormat*, del paquete `es.us.lsi.dp.formats`. Esta clase tiene cuatro atributos:

```

protected String codePrefix;
protected String field;
protected Class<?> type;
protected MessageSource messageSource;
  
```

El primero es el prefijo del código de internacionalización donde se especifica el formato del campo. *Field* es el nombre del atributo al que le vamos a aplicar el formato, *type* es la clase que representa al tipo del atributo, y *messageSource* es un atributo que el usuario no tendrá que especificar. Sirve únicamente para que desde *GetController* se especifique el objeto que representa el conjunto de mensajes de internacionalización declarados en los *messages.properties*^[14].

Las clases que heredan de *CustomFormat* tendrán siempre dos constructores: uno que recibe los atributos *codePrefix*, *type* y *field*, y otro que simplemente recibe *codePrefix* y *type*.

Sprouts permite, por defecto, implementar tres tipos de formateadores: para fechas, números decimales y unidades monetarias. Permite, además, establecer formatos generales para todas las vistas o específicos para una en concreto. De la misma forma, permite definir distintos formateadores según el controlador.

Los formatos generales se especifican en el fichero `resources/messages/messages.properties` en inglés y `resources/messages/messages_es.properties` en español. Si queremos aplicar un formato

distinto a un campo concreto, lo especificaremos en el fichero de internacionalización de la carpeta de la vista en la que lo queramos usar. El código será siempre de la siguiente forma: “*nombre_de_la_vista*”.*“código_del_formato”*.

Por otra parte, para aplicar los formateadores existen dos posibles caminos. El primero es establecerlo por defecto para todos los controladores. Esto quiere decir que por defecto se aplicará el formateador que se especifique. Se declaran en la clase *DefaultFormats* del paquete formatters, dentro del método *getDefaultsFormats()*. Por defecto, Sprouts Framework incluye un formateador para todos los objetos de tipo *Date* y para todos los de tipo *Double*. El contenido del método es el siguiente:

```
public static List<CustomFormat> getDefaultsFormats() {  
    List<CustomFormat> result = new ArrayList<>();  
    result.add(new CustomDateFormat("", Date.class));  
    result.add(new CustomDecimalFormat("", Double.class));  
    return result;  
}
```

Los formateadores se deben añadir a la lista *result*. De esta manera, podrán ser aplicados por *GetController*, que es quien se encarga de llevar a cabo esta labor. Por defecto, se aplican dos formateadores: uno a todos los objetos de tipo *Date* (*new CustomDateFormat("", Date.class)*) y otro a todos los objetos de tipo *Double*, (*new CustomDecimalFormat("", Double.class)*). En los próximos apartados se explicará detalladamente cómo usar cada formateador y qué código hay que usar para especificar el formato.

La segunda forma de aplicar formateadores es declararlos en un controlador. Para ello, este debe implementar la interfaz **AddCustomFormat**. Al hacer esto, será necesario definir el método **addCustomFormats**, que recibe una lista de objetos de tipo **CustomFormat**. En esa lista debemos añadir, de manera similar que en el método *getDefaultsFormats*, los formateadores que deseemos que sean usados cuando se llame a ese controlador.

Como ejemplo, exponemos la clase *CreateController* de *FeePayments*. La cabecera de la clase quedaría de la siguiente forma:

```
public class CreateController extends  
AbstractCreateController<FeePayment, FeePaymentService> implements  
AddCustomFormat {...}
```

y el método que hay que redefinir queda así:

```
@Override  
public void addCustomFormats(List<CustomFormat> formats) {  
    formats.add(new CustomCurrencyFormat("",  
    BigDecimal.class, "fee"));
```

```
}
```

Aquí se está aplicando un formateador cuyo formato está definido en *resources/messages* (porque la cadena que indica el prefijo está vacía), aplicada a objetos de tipo *BigDecimal*, y a todos los atributos llamados *fee*.

- i** No es posible tener definido un formateador por defecto y otro específico. Es decir, si tenemos un formateador por defecto para objetos de tipo *Date* en la clase *DefaultFormats*, no podemos tener otro en un controlador aplicado a un campo en concreto. Si el desarrollador desea hacer esto último, debe definir todos los demás formatos de fecha en cada controlador donde aparezca un objeto de tipo *Date*.

5.4.2 Formateador de fechas: *CustomDateFormat*

Los formatos de fecha se especifican con los siguientes códigos de internacionalización:

- *date.format* para el formato de fecha con el que trabajará la aplicación.
- *datepicker.format* para el formato de fecha con el que trabajará el datepicker.

Por ejemplo, los formatos de fecha establecidos por defecto en Sprouts-Framework están definidos de la siguiente manera:

resources/messages/messages_es.properties

date.format	= dd/MM/yy HH:mm
datepicker.format	= dd/mm/yy hh:ii

resources/messages/messages.properties

date.format	= MM/dd/yy HH:mm
datepicker.format	= mm/dd/yy hh:ii

Para usarlo, es necesario crear una instancia de la clase *CustomDateFormat*. Esta clase tiene dos constructores. Uno recibe los atributos *codePrefix*, *type* y *field*, y el otro simplemente recibe *codePrefix* y *type*. Estos atributos ya se han explicado cuando se describió la clase *CustomFormat*.

Para usarlo, es necesario crear una instancia de la clase *CustomDateFormat*. A continuación se exponen un ejemplo de posibles implementaciones de formateadores de fecha, declarando un formateador para todos los atributos de tipo Date en la clase *DefaultFormats*:

```
public static List<CustomFormat> getDefaultFormats() {  
    List<CustomFormat> result = new ArrayList<>();  
    result.add(new CustomDateFormat("", Date.class));  
    return result;  
}
```

5.4.3 Formateador de números decimales: CustomDecimalFormat

Los formatos de números decimales se especifican con los siguientes códigos de internacionalización:

- *decimal-mark*: carácter a usar como separador decimal.
- *grouping-separator*: carácter a usar como separador de centenas
- *number-format*: formato del número. Se indica, mediante caracteres de tipo almohadilla (#). Indica a partir de qué dígito se debe usar el separador de centenas (carácter coma “,”) y el número de cifras decimales a usar (viene dado por el número de almohadillas tras el separador decimal “.”).

Por ejemplo, los formatos de cifras decimales establecidos por defecto en Sprouts Framework están definidos de la siguiente manera:

resources/messages/messages_es.properties

```
decimal-mark          = ,  
grouping-separator   = .  
number-format         = #,###.##
```

resources/messages/messages.properties

```
decimal-mark          = .  
grouping-separator   = ,  
number-format         = #,###.##
```

Para usarlo, es necesario crear una instancia de la clase *CustomDecimalFormat*. Esta clase tiene dos constructores, al igual que *CustomDateFormat*, uno que recibe los atributos *codePrefix*, *type* y *field*, y otro que simplemente recibe *codePrefix* y *type*. A continuación se expone un ejemplo de uso de formateadores de números decimales:

```
public static List<CustomFormat> getDefaultFormats() {  
    List<CustomFormat> result = new ArrayList<>();  
    result.add(new CustomDecimalFormat("", Double.class));  
    return result;  
}
```

5.4.4 Formateador de unidades monetarias: CustomCurrencyFormat

Los formatos de unidades monetarias se especifican con los siguientes códigos de internacionalización:

- *currency.prefix*: símbolo de unidad monetaria que se mostrará a la izquierda de la cifra. Si queremos mostrar un símbolo, se debe especificar el código fuente Java de dicho carácter. Por ejemplo, para el carácter Euro (€), el código sería \u20AC.
- *currency.suffix*: igual que *currency.prefix*, pero referido al símbolo de la unidad monetaria que se mostrará a la derecha de la cifra.
- *currency.decimal-mark*: tipo de separador decimal.
- *currency.grouping-separator*: tipo de separador de centenas
- *currency.number-format*: formato del número. Ya se explicó en el el formateador de números decimales (*CustomDecimalFormat*).

Por ejemplo, los formatos de unidades monetarias establecidos por defecto en Sprouts Framework están definidos de la siguiente manera:

resources/messages/messages_es.properties

```
currency.prefix          =
currency.suffix          = \u20AC
currency.decimal-mark   = ,
currency.grouping-separator= .
currency.number-format  = #,###.##
```

resources/messages/messages.properties

```
currency.prefix          = \u20AC
currency.suffix          =
currency.decimal-mark   = .
currency.grouping-separator= ,
currency.number-format  = #,###.##
```

Para usarlo, es necesario crear una instancia de la clase *CustomCurrencyFormat*. Esta clase tiene dos constructores, al igual que *CustomDateFormat* y *CustomDecimalFormat*, uno que recibe los atributos *codePrefix*, *type* y *field*, y otro que simplemente recibe *codePrefix* y *type*. Exponemos un ejemplo en el que se aplica el formato a un atributo específico, “fee”, en la clase *FeePayment*:

Cabecera de la clase:

```
public class CreateController extends
AbstractCreateController<FeePayment, FeePaymentService> implements
AddCustomFormat {...}
```

Método `addCustomFormats`:

```
@Override  
public void addCustomFormats(List<CustomFormat> formats) {  
    formats.add(new CustomCurrencyFormat("", BigDecimal.class,  
                                         "fee"));  
}
```

Como podemos ver, le pasamos como parámetro el nombre del atributo de la entidad `FeePayment` al que queremos aplicarle este formato.

Si quisiéramos especificar un formato distinto para este controlador en concreto, debemos definir ese nuevo formato en los ficheros de internacionalización de la vista de `FeePayment`. Se haría de la siguiente manera:

Fichero `resources/views/feePayment/messages_es.properties`

<code>feePayment.currency.prefix</code>	=
<code>feePayment.currency.suffix</code>	= USD
<code>feePayment.currency.decimal-mark</code>	= ,
<code>feePayment.currency.grouping-separator</code>	= .
<code>feePayment.currency.number-format</code>	= #,###.##

Fichero `resources/views/feePayment/messages.properties`

<code>feePayment.currency.prefix</code>	= USD
<code>feePayment.currency.suffix</code>	=
<code>feePayment.currency.decimal-mark</code>	= .
<code>feePayment.currency.grouping-separator</code>	= ,
<code>feePayment.currency.number-format</code>	= #,###.##

Método `addCustomFormats` de `CreateController` de `FeePayment`:

```
@Override  
public void addCustomFormats(List<CustomFormat> formats) {  
    formats.add(new CustomCurrencyFormat("feePayment",  
                                         BigDecimal.class, "fee"));  
}
```

i Aconsejamos el uso del tipo BigDecimal para las unidades monetarias. El motivo es que a los atributos de tipo Double se les aplica por defecto el formateador de cifras decimales, por lo que no sería posible aplicar otro tipo de formateador a las unidades monetarias de tipo Double.

i Lo que hemos visto en este apartado es válido para formularios, y vistas para mostrar información sobre una entidad. Para este último caso, es necesario usar etiquetas específicas para obtener el formato, y para el caso de las tablas, la implementación es completamente distinta. En los capítulos del componente Datatable y vistas, se explican estos casos de manera detallada.

5.5 Conclusión

Hemos visto la estructura interna de clases de Sprouts Framework para los controladores. Provee un exhaustivo control sobre las posibilidades de hacking y acceso no autorizado, y transacciones. Esta misma estructura le da al desarrollador una alta flexibilidad y le brinda posibilidades más allá de simples operaciones CRUD.

Sprouts Framework permite implementar controladores, teniendo que implementar tan solo dos métodos en la mayoría de casos. Los controladores de casos de uso más complejos, son sencillos de implementar, ya que, gracias a los controladores que Sprouts provee para ello, queda bien acotado el ámbito y el rol de cada método.

Así mismo, hemos aprendido la importancia que tiene saber dónde se llama cada método y en qué orden, de cara a implementar casos de uso más complejos.

6. COMPONENTE DATATABLES

6.1. Introducción

Este componente es un plug-in para la librería JavaScript de jQuery. Es una herramienta que ofrece cierta flexibilidad y permite llevar a cabo interacciones complejas con cualquier tabla HTML. Soporta paginación, búsqueda instantánea y ordenación por varias columnas. Para obtener los datos a mostrar en la tabla se hace uso de procesamiento en el lado del servidor. Esto es, la tabla se encarga de pedir los datos de forma dinámica y asíncrona al servidor. De esta forma, no es necesario traer todos los datos de la tabla al lado del cliente, enviándole solo los justamente necesarios para la página que esté visualizando. A continuación se van a explicar todas las clases de utilidad que han sido necesarias crear para dar soporte a Datatables, así como los controladores necesarios y cómo funciona el proceso de construcción de las tablas.

6.2. Clases de utilidad asociadas a DataTables

6.2.1. DatatableJson

Esta clase representa el objeto JSON que espera recibir el componente *DataTables* cuando realiza una petición de información al servidor. Tiene los atributos necesarios según lo exigido en la documentación del componente. Puede acceder a la sección *Reply from the server* de la web del mismo^[15] si quiere más información acerca del uso de cada uno de estos parámetros.

6.2.2. Table

La finalidad de esta clase es representar la estructura de la tabla. Sirve de utilidad para construir posteriormente de forma sencilla el *DatatableJson* que representa el JSON de respuesta que se enviará al cliente que solicitó los datos de una tabla. Para ello, el único atributo que tiene es una lista de *Column*, donde se almacenan cada una de las distintas columnas que se definen para una tabla dada.

6.2.3. Column

Esta clase tiene como objetivo representar el contenido de una columna de la tabla. Es por ello que se tienen diversos atributos, cada uno de ellos representando elementos de relevancia en la columna. Cada atributo tiene correspondencia directa con un atributo en la custom-tag, que se usa en los ficheros .JSP, *data-column*. En la sección correspondiente a las vistas se comentará cuál es la finalidad de cada uno de dichos atributos.

Además de estos atributos, también tiene dos referencias a la ordenación de dicha columna. Uno es *sortable* que es un *boolean* indicando si la columna se puede ordenar o no, y otro es un *Array* con el criterio de ordenación de dicha columna, que por defecto será el contenido.

En esta clase nos basta con saber que por cada uno de los atributos que se añade a la columna son necesarios tres métodos asociados al mismo: un getter, un setter y un método como el del siguiente ejemplo

```
public Column nullCode(final String nullCode) {  
    setNullCode(nullCode);  
    return this;  
}
```

El método debe devolver una columna y tiene como nombre el del propio atributo, además recibe por parámetros el valor del atributo. Dentro del método se llama al setter para asignar al atributo en cuestión el valor pasado por parámetro y se devuelve la columna.

6.2.4 TableBuilder

Es una interfaz, que únicamente tiene dos métodos. Esta interfaz proporciona únicamente la declaración de dos métodos, los cuales se deben declarar e implementar si se implementa dicha interfaz. Uno de estos métodos es *build*, que recibe como parámetros el nombre de una vista, el id de una tabla, una HTTP request y una HTTP response. Debe devolver una clase de tipo *Table* con sus correspondientes columnas inyectadas en ella. Además se necesita implementar el método *getRows*, que recibe como parámetros una lista de entidades del dominio, una *Table* y una cadena representando el *Locale* del usuario que ha hecho la petición.

6.3. Controladores asociados a DataTable

6.3.1. GetCollectionController

Este controlador no tiene las funciones típicas que se podría esperar de un controlador. Realmente sirve como una clase de utilidad para que el controlador que sí que tiene las funciones de recibir las peticiones de parte de los usuarios, pueda redirigirlas a una vista con unos ciertos datos, que son los que se van a construir con los métodos de esta clase.

En esta clase se dispone de una variable *tableBuilder*, en la que se inyecta mediante la anotación *@Autowired* el componente que se encarga de la construcción de las tablas. Todo lo referente a la construcción de las tablas puede consultarse en el apartado 6.3. *Construcción de las tablas*.

Además tiene los siguientes métodos que se comentan a continuación:

- *getData*: este método devuelve un objeto de tipo *DatatableJson*, clase que se comentó con anterioridad. Recibe como parámetros una serie de atributos que son imprescindibles para construir la tabla. A continuación una breve descripción de qué representa cada uno de ellos.
 - *HttpServletRequest request*: petición que hace el usuario al servidor.
 - *HttpServletResponse response*: respuesta que se va a enviar de parte del servidor.
 - *String viewName*: nombre de la vista en la que se muestra el listado.

- o *String tableIndex*: índice de la tabla, útil cuando existe más de una tabla en la misma vista.
- o *Map<String, String> params*: mapa con parámetros que se reciben de la petición del cliente, corresponde a las *PathVariables*.
- o El resto de parámetros son enviados al lado del servidor por parte del componente de la tabla cuando se hace cualquier petición. Se puede obtener más información acerca de los mismos en su documentación, en la sección *Parameters sent to the server*^[15].

Dentro de este método se construye el objeto de tipo *Table* mediante el método *build* del componente *tableBuilder* que se injectó. Se comentará más adelante dicho método cuando se hable acerca de la clase *DefaultTilesViewTableBuilder*.

Una vez que se ha construido esa *Table*, conteniendo la estructura de la tabla, se llama al método *getJson* (detalles a continuación) de esta misma clase, al cual se le pasa por parámetro entre otras cosas, la tabla que se acaba de construir. Este método construye el JSON de respuesta que se enviará al cliente. Finalmente se devuelve este *DatatableJson* que es el objeto que espera recibir el componente de la tabla.

- *getJSON*: este método devuelve un objeto de tipo *DatatableJson*. Recibe los mismos parámetros que el método *getData* y adicionalmente un objeto de tipo *Table* que representa la estructura de la tabla que se creó en el método *getData*.

En primer lugar, se prepara el objeto *Pageable* que se va a pasar al método del repositorio que recupere la página correspondiente. Para ello se llama al método *prepareRequest*, pasando los atributos necesarios para poder construir este *Pageable*.

Una vez se ha construido el objeto *Pageable*, el siguiente paso es obtener la *Page<Entity>* con los datos de la página que se han solicitado, que se hace llamando al método *getPage*.

Cuando se tiene la página que ha devuelto el repositorio, se recupera el listado de objetos que trae consigo. También se recupera el total de resultados que ha devuelto la consulta para poder construir posteriormente el JSON de respuesta, que necesitará dicho dato.

El siguiente paso es construir un *Array* de *String* de dos dimensiones que contiene los datos de cada una de las filas de esa página de la tabla. El primer índice del *Array* hace referencia a la fila y el segundo a una columna concreta dentro de esa fila. Para construirlo se llama al método *getRows* del componente *tableBuilder* que se injectó. Necesita como parámetros el listado de objetos que devolvió la consulta, el objeto *Table* y el *Locale* en el que está la petición.

Por último, falta construir el propio objeto *DatatableJson* que se devolverá al cliente. Para ello se llama a su constructor pasando por parámetros *sEcho* que se recibía como parámetro en este método, el número total de elementos a mostrar y el *Array* de dos dimensiones con los datos de la tabla.

- *prepareRequest*: el método devuelve un objeto de tipo *Pageable*, necesario para hacer la petición al repositorio solicitando una página concreta.

El primer paso es determinar cuál es el orden que van a seguir las páginas. Para determinarlo, se necesitan dos parámetros: el criterio de ordenación y la dirección. El primero de ellos se recupera del atributo de la columna referente al propio criterio de ordenación. Respecto a la dirección, se llama a un método auxiliar de esta propia clase, *sortDirection*. Con estos parámetros ya listos, se llama crea un objeto de tipo *Sort*.

Posteriormente, se calcula el número de la página a partir del índice del primer elemento a mostrar en la página y el número total de elementos mostrados por página.

Para finalizar, se construye el objeto *Pageable* a partir del número de página, el número total de páginas y el orden.

- *sortDirection*: dicho método lo único que realiza es un parseo del atributo *sortDirection* que envía como parámetro la tabla al servidor. Si es “asc” la dirección de la ordenación es ascendente, caso contrario será descendente.
- *getPage*: en esta clase está vacío, sin definir. Se hace así de esta forma para que en las clases que heredan de esta, las que se encargan de los controladores de listado en sí, se sobreesciba el método para así poder recuperar la página que sea de interés.

6.3.2. AbstractGetListDataController

Este controlador es el encargado de recibir las peticiones de los datos por parte de la tabla. Esta clase es el primer controlador en sí, es decir, que realiza las funciones propias de un controlador, en la jerarquía de controladores de listado.

En primer lugar, *AbstractGetListDataController* tiene un atributo llamado **service** con modificador de acceso *protected*, que representa el servicio con el que trabaja el controlador. Gracias a este atributo, en cualquier parte de nuestro controlador podremos acceder al servicio.

```
protected S service;
```

Los métodos que se pueden encontrar dentro de esta clase que representa al controlador son los siguientes:

- *data*: este método se encarga de devolver un objeto de tipo *DatatableJson*. Las URIS a las que se encarga de responder son del siguiente tipo:

```
entidad/rol/list/data.do  
entidad/rol/{context}/list/data.do
```

El componente *Datatables*, cuando se hace uso del mismo en un fichero .jsp, hace referencia por defecto a una URL que sigue este patrón para pedir los datos al lado del servidor.

Los parámetros que recibe el método y por tanto la petición del lado del cliente son los atributos que se esperan recibir por parte de la tabla cuando envía una petición al servidor. Se pueden consultar los mismos en la documentación oficial del componente. Además, recibe dos parámetros extra que son *_viewName* y *_tableIndex*. Estos serán de

utilidad posteriormente cuando se vaya a construir la tabla para el JSON de respuesta. El primero de los parámetros, hace referencia al nombre de la vista en la que aparece la tabla en cuestión que está enviando la petición, y el segundo referencia el índice de la tabla, esto es en caso de que haya más de una tabla en una misma vista, de parte de cuál de ellas viene la petición.

El método en sí, lo único que realiza es llamar al método *getData* de la clase padre de la que hereda, recordemos que es *GetCollectionController*. Le pasa los parámetros que recibe la petición, además de las *PathVariables* que recupera a través de la petición HTTP.

- *getPage*: se encarga de devolver un objeto de tipo *Page<Entity>*. Esta página, de la entidad que se está listando, es la que el componente de la tabla pide al servidor, aquella que el usuario desea visualizar. Se recibe por parámetros un objeto *Pageable* con la información de la página, el criterio de búsqueda y el contexto.

Por defecto, se devuelve la página que retorna el método *findPage* del servicio que se pasa por parámetro al controlador. Si se quiere modificar la página que se quiere devolver a la tabla, se podría hacer. Bastaría con sobreescribir este método desde el controlador hijo de *AbstractListController*, es decir, el controlador que creemos nosotros.

- *view*: este método devuelve la cadena que en el fichero *tiles.xml* correspondiente representa a la vista que se quiere mostrar. Como en el caso de este controlador, la única función que tiene es devolver de forma asíncrona los datos de la tabla al lado del cliente, este método devuelve *null*, puesto que no se corresponde a ninguna vista.

6.4. Construcción de las tablas

La construcción de las tablas se lleva a cabo mediante el componente ***DefaultTilesViewTableBuilder***. Esta clase implementa a la interfaz previamente definida, *TableBuilder*. Además, para que dicha clase sea un componente se hace uso de la etiqueta *@Component*, para indicarle a Spring que cuando realice el scan en busca de componentes, que esta clase sea uno de ellos.

Esta clase dispone de varios métodos, con la finalidad de construir las tablas a partir de una petición que se ha realizado desde el lado del cliente, y partiendo de los datos que tenemos presentes en los distintos ficheros *tiles.xml*. En concreto aquel que esté relacionados con la tabla en cuestión que necesitamos construir.

Los atributos presentes son:

- *messageSource*: se inyecta mediante la etiqueta *@Autowired*, al tratarse de un componente del que queremos hacer uso en esta clase. Su funcionalidad es permitirnos ciertas facilidades a la hora de recuperar mensajes internacionalizados, que se encuentren declarados en los ficheros *messages.properties*^[14].

- Una serie de constantes, que principalmente hacen referencia a etiquetas que se usan para parsear posteriormente ciertos campos.

Pasamos a la descripción de cada uno de los métodos que se encuentran en esta clase, ya que tienen cierta complejidad y es importante entender para qué sirven cada uno de ellos.

- *build*: este es uno de los métodos que se deben implementar de forma obligatoria, dado que está declarado en la interfaz *TableBuilder* que aquí se implementa. Como se comentó al hablar de la interfaz, dicho método tiene que recibir como parámetros el nombre de la vista, un índice indicando que tabla es (útil en caso de que en una vista hubiese más de una), una HTTP request y una HTTP response. El método devuelve como resultado un objeto de tipo *Table*, con la tabla ya construida.

Primero, se recupera un objeto de tipo *Definition*, que tiene una estructura similar a un mapa, con clave=valor. Dichos valores se recuperan a partir de un método *getTemplateDefinition* de la clase *TilesUtils* al que se le pasan los parámetros correspondientes al nombre de la vista, la *request* y la *response*. La definición obtenida tendrá una estructura similar a esta:

```
{name=entity/list, template=/WEB-INF/classes/views/template/master.jsp,
role=null, preparerInstance=null, attributes={title=entity.list.title,
body=../entity/list.jsp}}
```

A partir de estos valores de la instancia de *Definition* se obtiene la ruta al fichero .jsp que se necesita analizar, donde se habrá declarado la tabla y será necesario para ver cómo construirla. Para llevar a cabo esta tarea se utiliza de nuevo un método de *TilesUtils*, en concreto *getJspPath*, que recibe la definición, *request* y *response* como parámetros.

Cuando disponemos de la ruta de este fichero, ya se puede acceder al mismo para recuperar los valores que necesitemos. Por eso, el próximo paso es obtener los valores de los atributos de las columnas de la tabla. Esto es recuperar de nuevo una estructura similar a un mapa con atributo=valor. Estos atributos son cada uno de los atributos de las etiquetas *<sprouts:data-column>*. Se entrará en mayor detalle cuando comentemos el método *getUnparsedColumns* de esta misma clase. Ese método recibe la ruta al fichero .jsp y el índice de la tabla que queremos.

Una vez que tenemos para cada columna estos pares atributo=valor, se procede a parsear las columnas. Como resultado de esta operación, obtendremos una lista con la estructura de todas las columnas. Dichas columnas se crean como una clase de tipo *Column*, ya comentada con anterioridad.

Finalmente, cuando ya tenemos una lista con todas las columnas de las que dispone la tabla, ya habremos finalizado cuando se devuelva una *Table* con las mismas. Para ello se utiliza el constructor de esta clase que recibe como parámetros una lista de columnas y la construye a partir de la misma.

- *getRows*: este método es el otro que es de obligada redefinición al implementar la clase a *TableBuilder*. Como parámetros recibe: una lista con objetos de la entidad, la que se

esté tratando de listar, un objeto de tipo *Table* con la estructura de la tabla una vez construida y un String que representa el idioma de la petición que se ha realizado. El resultado a devolver es un *Array* de *String* de dos dimensiones, donde el primer índice representa a una fila de la tabla a la que se referencia y el segundo índice a una columna de esa misma fila.

Lo primero que se realiza en este método es obtener el tamaño de este *Array*. El tamaño del primer índice será el mismo que el listado de entidades que se tiene como parámetro y el del segundo será el tamaño de la estructura de la tabla + 1, la cual incluye el número de columnas de la misma. Cuando tenemos estos valores, se construye el *Array* con este tamaño.

El siguiente paso es recorrer ese *Array* que acabamos de construir para darle valores a cada una de sus filas y columnas. El procedimiento es el siguiente:

- o Cada vez que se empieza a recorrer una nueva fila de la tabla, se recupera el valor del objeto correspondiente a dicha fila a partir del listado que se recibía por parámetro.
- o Para cada columna, hay que transformar el valor que tenga el atributo del tipo que sea, en algo que sea una cadena de texto. Además, en esta parte debemos tener en cuenta las distintas opciones de formato que haya querido dar el usuario. Debido a ello, se realizan una serie de comprobaciones para ver si es necesario formatear el valor a mostrar en la columna en cuestión.
- o Pasamos a describir paso a paso el proceso que se sigue en cada columna:
- o Primero, se recupera de la *Table*, la columna por la que nos encontramos.
- o Segundo, se llama al método *resolveColumnValue* de esta misma clase. Este devuelve un valor dado una entidad y una columna que se pasan por parámetro. En general, dicho valor se corresponderá a la representación como cadena que se tenga por defecto del atributo de la entidad que se corresponda a esa columna. Posteriormente, cuando se comente este método se entrará en más detalles. Este valor que devuelve el método se almacena en la fila y columna correspondiente del *Array* de 2 dimensiones que estamos construyendo.
- o Una vez que se han realizado estos dos primeros pasos, si se tratase de una columna “simple”, a la cual el usuario hubiese decidido no darle ningún tipo de formato y dejarla con la representación como cadena que tenga por defecto, ya se habría finalizado con la construcción de una columna.
- o Sin embargo, para cada columna, es necesario realizar comprobaciones para ver si se le ha dado algún formato concreto para representarla. Pasamos a explicar cada una de las posibles casuísticas que se pueden dar.
 - Si se le ha dado valor al atributo ***toShow***, es que se quiere que el mensaje a mostrar sea una cadena de texto determinada, entre la cual se suele encontrar además el propio valor de la columna. Lo que se realiza dado el caso es sustituir la cadena correspondiente a esta columna por la que se haya pasado como valor del atributo *toShow*. Además, si se ha incluido la palabra clave “+PATH”, se sustituye dicha

cadena por el valor del atributo correspondiente a esa columna de esa entidad que se está listando.

- Las siguientes comprobaciones que se van a hacer son todas sobre el atributo **format**. Según el valor de la cadena de dicho atributo, significará que se quiere representar un tipo de dato u otro en la columna.
- El primero de los casos posibles: cuando **format** es **image**. Se quiere que en la columna correspondiente se muestre una imagen en lugar de una representación con texto. Dentro de este caso hay que verificar si ha decidido darle un tamaño concreto a la imagen. Para ello se comprueba si tiene valor el atributo **imgSize**. En caso afirmativo, hay que parsear el tamaño que ha dado el usuario. La forma de dar el tamaño es pasándolo como valor en el formato '*anchoXalto*'. Si no se pasa un tamaño concreto, se utiliza el que tenga por defecto la imagen redimensionada para que no deforme la tabla. El siguiente paso es comprobar si se ha dado valor al atributo **outFormat**. En caso de haberlo realizado, significa que la imagen a mostrar no es directamente el valor almacenado en el atributo de la entidad en cuestión. Entonces, se sigue un procedimiento igual que el comentado cuando se hacía uso de **toShow**, se usa el valor de una cadena que se pase como valor y se sustituye la subcadena "+*PATH*" por el valor del atributo. Una vez que se han realizado todas estas comprobaciones, lo último es sustituir el valor de la columna por una imagen con sus correspondientes etiquetas HTML. Dependiendo de los atributos opcionales que haya decidido completar, se creará de forma más personalizada o con los valores por defecto.
- Otro posible caso del atributo **format**: cuando se trata de una **URL**. Dentro de este caso, hay dos posibilidades. Una de ellas es que se quiera que el valor por defecto de esa columna sea el que se convierta en el enlace y el mensaje que se muestre en esa URL sea el mismo valor. El otro es que quiera que la URL sea la misma, sin embargo, el mensaje que se muestra en la pantalla es una cadena de texto distinta. En ese caso, el atributo **outFormat** deberá tener valor también. El procedimiento es el mismo que el comentado justo antes para una imagen. Finalmente, se sustituye el valor de la columna por una URL con sus etiquetas HTML correspondientes.
- El siguiente caso es cuando el atributo **format** toma el valor **date** y además el valor esa fecha de la columna correspondiente no es nulo. Se quiere que en lugar de mostrar la fecha con el formato por defecto, con el cual se recupera de la base de datos, se muestre con otro formato. Dentro de este caso, de nuevo debemos distinguir entre dos posibilidades. Una de ellas es que quiera que se muestre la fecha con el formato que haya definido por defecto en el fichero *messages.properties*, según el idioma del que se trate. Otra posibilidad

es que quiera un formato concreto para esa tabla, entonces deberá declararlo en un fichero *messages.properties* para cada idioma y pasárselo en el atributo ***outFormat***. El procedimiento en caso de tener que tratar con una fecha es algo más complejo. En primer lugar, tenemos que parsear la cadena con la fecha que se había recuperado de la base de datos. Una vez que tengamos un objeto de tipo *Date*, procedemos a recuperar el idioma con el que está tratando el usuario que ha hecho la petición. A partir del idioma y dependiendo del caso en el que nos encontremos, se recupera el formato propiamente dicho, (del tipo 'DD/MM/YYYY') declarado en los *messages.properties*. Para ello se hace uso del atributo que se comentó al inicio de esta sección: *messageSource*, que a partir del idioma y la cadena que estemos tratando de buscar, devuelve el valor que se guardase en el correspondiente mensaje de internacionalización. Finalmente, se crea un objeto de tipo *SimpleDateFormat* a partir de esa cadena con la representación del formato, y a partir de este objeto se formatea la cadena con la fecha. Para saber más detalles sobre cómo formatear y parsear valores, se recomienda acudir a la sección del manual correspondiente (5.4). Por último, de igual forma que los otros casos, se sobreescribe el valor que tenía almacenado el Array para esta columna.

- Otra posible casuística es que el atributo ***format*** tome el valor ***currency***. Se dará cuando se quiera representar el valor de una moneda concreta. Nos encontramos con dos posibles escenarios. Uno, cuando se quiere dar el valor por defecto que haya declarado. El otro cuando se quiere dar un formato concreto para esta tabla, diferente del que se ha dado por defecto. La única diferencia en ambos casos, es que en el segundo de estos, el usuario deberá pasar en el atributo ***outFormat*** el prefijo de los mensajes del *messages.properties* donde ha declarado los parámetros necesarios. Estos son: *currency.decimal-mark*, *currency.grouping-separator*, *currency.number-format*, *currency.prefix*, *currency.number-format*. De nuevo, si se quiere conocer más información sobre el formato, acudir a la sección del manual que corresponde (5.4). El proceso que se aplica para darle formato a una moneda es el mismo que el que se comenta ahí. Una vez que se ha formateado, se actualiza el valor de la columna con esa representación.
- El último caso posible es que ***format*** tenga el valor ***decimal***. En este caso, se trata de dar el formato de un número decimal. El proceso es idéntico al caso cuando de ***currency***. La única diferencia es que los parámetros de los que se necesita valor son: *decimal-mark*, *grouping-separator* y *number-format*. El resto del proceso es similar, y de igual forma, si se quieren saber más detalles, en la sección de los formatos se podrá consultar lo que sea necesario. Cuando se ha

- formateado, se actualiza el valor de la columna con esa representación.
- Una vez realizadas todas estas comprobaciones para cada una de las columnas de una fila determinada, en la última columna se almacena el valor del ID de la entidad.
- resolveColumnValue*: este método recibe como parámetros una columna y una entidad. Devuelve como resultado una cadena que representa el valor de dicha columna para la entidad que se pasa por parámetro.

Se hace uso de un objeto de tipo *ExpressionParser* y otro de tipo *Expression*, propios de Spring. El primero de ellos se utiliza para construir una *Expression* a partir del nombre del campo de la columna que queremos resolver. A partir de esta *Expression* y pasando la entidad como parámetro, se obtiene el valor del atributo, cuyo nombre coincide con el nombre de la columna. Hay varios posibles casos: en el caso general, se recupera el valor del atributo que interese de la entidad y ese es el resultado que se devuelve. Cuando el valor es nulo, se llama al método *getNullMessage* de esta misma clase. Cuando el valor es *true* o *false*, se hace lo propio llamando al método *getTrueMessage* o bien *getFalseMessage* respectivamente. Los valores que se devuelven en estos últimos casos son los que den como resultado dichos métodos.

- getNullMessage*, *getTrueMessage*, *getFalseMessage*: todos reciben por parámetro una columna. Devuelven como resultado el mensaje por defecto de dicha columna en caso de que el valor sea null, true o false. El funcionamiento de todos estos métodos es el mismo. La única diferencia es que en cada uno de los casos, el valor que se recupera será *column.getNullCode()*, *getTrueCode()* o *getFalseCode()*, respectivamente. Además de recuperar dicho código que es el correspondiente al mensaje de internacionalización, se necesita el idioma del usuario. Finalmente se recupera de los *message.properties* el valor por defecto que se ha decidido asignar a las columnas que sean nulas, sean un boolean con valor true o boolean con valor false, respectivamente.
- parseColumns*: recibe por parámetro un Array de String, con las columnas sin parsear. Cada elemento de este Array tiene los atributos que se le asignaron a dicha columna en el fichero .jsp. El método devuelve una lista de *Column*, donde cada columna tiene su estructura, que vendrá dada según se haya indicado en el correspondiente .jsp.

En este método, se recorre la lista de columnas sin parsear, y sobre cada una de ellas se realizan una serie de operaciones. Primero se crea un *Map<String, String>* a partir de los valores de la columna sin parsear, mediante el método *buildAttributesMap*. Posteriormente, se construye una *Column*, recuperando la key del mapa, cuyo nombre está definido en la constante *PATH*. Una vez realizado, se llama al método *setAttributes*, con la columna y mapa que se acaban de construir para darle. Finalmente se añade al listado de columnas a devolver, la columna que se está recorriendo, ya con los atributos con su valor correspondiente.

- getUnparsedColumns*: el método recibe como parámetros la ruta del fichero .jsp que se va a analizar y el índice de qué tabla se debe analizar (útil en caso de que hubiese más de una en el mismo fichero). Devuelve como resultado un Array de String. En cada

posición del mismo, se encuentran los atributos correspondientes a cada columna de la tabla.

En este método se comienza haciendo uso de una clase de utilidad de Spring, *FileCopyUtils*. En concreto se utiliza el método *copyToByteArray*. A dicho método se le pasa un *FileInputStream*, un objeto que obtiene bytes de entrada de un fichero del sistema. Este último se construye pasándole la ruta del fichero. Con este método *copyToByteArray* se obtiene un Array de bytes. Para poder tratar con el fichero, es necesario transformar dicho Array de bytes en una cadena que sea legible. Para lograrlo, se usa uno de los constructores de *String*, que recibe como parámetros un Array de bytes y el conjunto de caracteres con los que se quiere codificar. Cuando ya se dispone de las declaraciones que se habían hecho en el fichero .jsp como una cadena de texto, se aplican diversas operaciones sobre la misma. En primer lugar, se obtienen los fragmentos de código que contienen la declaración de tablas. Se utiliza el método de *StringUtils*, *substringsBetween*, al que se le pasa la cadena con el fichero .jsp, y las etiquetas de apertura y cierre de la tabla, definidas en las constantes. Este método devuelve todas las subcadenas que existan entre esas dos etiquetas.

Una vez obtenidas estas tablas, se necesitan saber ciertos datos de las mismas. En primer lugar, se deben recuperar los permisos que existen en las etiquetas *security*, para gestionar que se vea determinado contenido en función del rol del usuario logueado. También se hace uso del método *substringsBetween*, en este caso pasando como parámetros del Array de declaraciones de tablas obtenido antes, el que se encuentra en la posición que indica el índice que se recibe, así como las etiquetas de apertura y cierre de *security*. Este método devolverá un Array de *String* (en caso de que lo hubiera) con los fragmentos de código de la tabla que estén envueltos con etiquetas *security* para garantizar los permisos.

De igual forma, se debe seguir el mismo procedimiento para recuperar un Array de *String* que contenga tan solo los roles que son necesarios en cada una de las etiquetas *security*. Para ello, se buscan las subcadenas en la declaración de la tabla correspondiente que estén entre “*hasRole(“ “ y “)*”.

Finalmente, también es necesario recuperar un Array de *String*, en el cual en cada posición aparecerán los atributos de cada una de las columnas que están declaradas en la tabla en cuestión. El procedimiento es idéntico a los anteriores, el único cambio es que ahora las subcadenas que se buscan deben estar entre etiquetas de apertura y cierre de columna.

Cuando ya hemos obtenido todas estas subcadenas, tan solo falta un paso para finalizar. Hay que filtrar aquellas columnas, las cuales el usuario que actualmente se encuentra logueado y ha hecho la petición no tiene permiso para verlas. Para ello, tenemos que comprobar que el Array obtenido con los permisos del *security* no sea nulo (no se ha encontrado ninguna etiqueta). En caso de que así haya sido, entonces se recorre con un índice cada una de las subcadenas encontradas en dicho Array. Se debe comprobar mediante el *SignInService* que el usuario que está logueado tenga el permiso indicado en el Array de roles (en la misma posición del índice) y que además el campo que está dentro de la etiqueta *security* sea una columna y no un botón. Si se cumplen

dichas condiciones, entonces se recupera el contenido de la columna en cuestión y se elimina del *Array* a devolver como resultado, con las columnas sin parsear.

- *buildAttributesMap*: recibe como parámetro un *String* con una columna sin parsear. Este método se encarga de crear un patrón con una expresión regular que se almacena en un atributo como una constante. Dicha expresión regular, lo que realiza es encontrar pares atributo=valor en los datos de la columna sin parsear. De forma que mientras se encuentren pares en los datos de la columna sin parsear, se realiza una tarea de manera iterativa. Esta tarea consiste en añadir al *Map<String, String>* que se devuelve como resultado, clave y valor. Donde la clave es el del primer grupo del *matcher* obtenido a partir del *pattern* y de la columna sin parsear y la key es del segundo grupo del mismo.
- *setAttribute*: recibe como parámetro una columna y un mapa de *String* a *String* con los atributos de la columna. Lo que se realiza dentro de este método es, en primera instancia, eliminar de los atributos aquel que corresponde a la ruta del atributo. Una vez completado, se recorren los pares (valor,atributo) del mapa y por cada uno de ellos se llama al método *invokeMethod* pasando como parámetros la columna y el par.
- *invokeMethod*: recibe como parámetro una columna y un par (clave, valor) que son dos *String*.

Este método, obtiene la clase del objeto que representa la columna. Además también necesita un objeto de tipo *Method*, que se usará posteriormente. Se recupera el nombre del método, que no es más que la clave del par que se pasó como parámetro y los argumentos del método, que son el value del par pasado por parámetro. Se comprueba si los argumentos del método son como un *Array* (en caso de contener una coma).

En caso de ser un *Array*, se construye el *Array* a partir de los argumentos del método. El resto de operaciones son iguales, independientemente de que los argumentos sean de tipo *Array* o no. En primer lugar, se obtiene el método, llamando al objeto que representa la clase columna, pasándole como parámetros el nombre del método (obtenido anteriormente) y la el tipo de valor, *Array* de *String* en el primer caso o *String* en el segundo. Finalmente, se invoca al método en cuestión, haciendo uso del método *invoke*, que recibe como parámetros la columna con la que se está tratando y los argumentos.

6.5. Conclusión

El funcionamiento de cada una de las clases necesarias para soportar el uso del componente *Datatables* ha quedado claro. De igual manera que se cumplimenta la explicación de la jerarquía de controladores de listado, dando detalles de los controladores de la capa más interna relacionados directamente con el listado y en concreto con este componente *Datatables*. Además, ha quedado patente cómo hacer uso de las distintas alternativas de representación que se ofrecen con este componente.

7. VISTAS

7.1 Introducción

En este capítulo se explicará cómo configurar las vistas que se crean mediante los archivos de configuración de Apache *Tiles*^[20] destinados para ello, la configuración necesaria en los ficheros jsp para su correcto funcionamiento, así como la estructura de los ficheros de internacionalización. Además se explicarán todas las etiquetas disponibles para su uso en los ficheros jsp, y por último dos clases de utilidades que están relacionadas con las vistas.

7.2 Tiles

Los archivos *tiles.xml* son ficheros de configuración de vistas. En estos ficheros configuraremos cada una de las vistas. En *resources/views*, habrá una subcarpeta por cada tipo de entidad y en cada una de estas carpetas habrá un archivo *tiles.xml* que configurará las vistas relacionadas con esa entidad.

Todas las definiciones que vamos a explicar ahora deben ir entre las etiquetas *<tiles-definitions></tiles-definitions>*.

Los elementos comunes para todas las definiciones serán los siguientes: *name*, que es el nombre de la vista; *extends*, vista de la que extiende, por lo general será *template/master*; *title*, título de la vista que aparecerá en la cabecera de la misma, definida mediante códigos de internacionalización; *body*, que será el cuerpo de la vista, que puede ser un archivo .jsp o una extensión de la definición.

Listado

```
<definition name="barter/list" extends="template/master">
    <put-attribute name="title" type="string" value="barter.list.title" />
    <put-attribute name="body" type="template" value="../barter/list.jsp" />
</definition>
```

Por lo general, las vistas de listado tendrán por nombre “*entidad/list*”. El título será el definido en los códigos de internacionalización y el *body* será la ruta relativa a donde se encuentre el archivo .jsp correspondiente a la vista de listado.

Creación

```
<definition name="barter/create" extends="template/master">
    <put-attribute name="title" type="string"
        value="barter.create.title" />
```

```

<put-attribute name="body" value="barter/create/body" />
</definition>

<definition name="barter/create/body" template="../barter/edit.jsp">
    <put-attribute name="readOnly" type="string" value="false" />
    <put-attribute name="action" type="string" value="create.button" />
</definition>

```

Esta definición se divide en dos, ya que es necesario añadir una serie de atributos más. El nombre por lo general será “*entidad/create*”. El atributo *body* en este caso es un nombre identificativo para poder definirlo más adelante. Será “*entidad/create/body*”.

La siguiente definición llevará por nombre la misma que fue definida para el atributo *body* anteriormente. El atributo *template* es específico para este tipo de definiciones, y será la ruta relativa al archivo .jsp que contiene la vista de creación de una entidad. El atributo *readOnly* indicará si el formulario será solo de lectura o permitirá escribir, en este caso al ser de creación, será *false*. El atributo *action*, será el nombre del botón de submit en la vista y será definido mediante códigos de internacionalización.

Actualización

```

<definition name="folder/update" extends="template/master">
    <put-attribute name="title" type="string" value="folder.update.title" />
    <put-attribute name="body" value="folder/update/body" />
</definition>

<definition name="folder/update/body" template="../folder/show.jsp">
    <put-attribute name="readOnly" type="string" value="false" />
    <put-attribute name="action" type="string" value="update.button" />
</definition>

```

La definición es parecida al caso de creación, pero adaptando el nombre, por lo general será llamada “*entidad/update*”, y el atributo *body* será definido de esta forma: “*entidad/update/body*”.

En la siguiente definición, el nombre será el definido para el *body*, y el atributo *action* será definido mediante códigos de internacionalización acorde con la operación que se realizará para la vista definida.

Eliminación

```
<definition name="folder/delete" extends="template/master">
    <put-attribute name="title" type="string" value="folder.delete.title" />
    <put-attribute name="body" value="folder/delete/body" />
</definition>

<definition name="folder/delete/body" template="../folder/show.jsp">
    <put-attribute name="readOnly" type="string" value="true" />
    <put-attribute name="action" type="string" value="delete.button" />
</definition>
```

La definición se realizará de forma parecida a las dos anteriores. Se debe adaptar de forma acorde a la eliminación. En el atributo *template* se debe poner lo mismo que en el de creación, ya que nos servirá para mostrar la información que va a ser eliminada. Para que no se pueda modificar esta información, el atributo *readOnly* deberá ser *true*.

Mostrar (Show)

```
<definition name="folder/show" extends="template/master">
    <put-attribute name="title" type="string" value="folder.display" />
    <put-attribute name="body" value="folder/show/body" />
</definition>

<definition name="folder/show/body" template="../folder/show.jsp">
    <put-attribute name="readOnly" type="string" value="true" />
    <put-attribute name="action" type="string" value="create.button" />
</definition>
```

La definición se hará de igual forma que para el borrado, pero adaptando el valor de los atributos para que se corresponda con una vista de mostrar.

Visualización (Display)

```
<definition name="barter/display" extends="template/master">
    <put-attribute name="title" type="string" value="barter.display" />
    <put-attribute name="body" type="template" value="../barter/display.jsp" />
</definition>
```

Se realiza de forma similar a los listados.

7.3 Ficheros JSP

Los ficheros con extensión .jsp son los que se utilizarán para realizar la creación de las vistas. Para cada entidad que exista en la aplicación y haya una vista relacionada con ella, se creará una carpeta específica para ello. Dentro de estas carpetas, se situarán los ficheros .jsp, así como los ficheros de internacionalización que se explicarán más adelante. Todos los ficheros deben tener en primer lugar la importación de todas las librerías necesarias para después escribir el código. Para ello, debemos incluir el fichero que importa todas estas librerías de la siguiente forma:

```
<%@ include file="../template/libraries.jsp" %>
```

La forma de darle cuerpo a estos ficheros para la creación de los diferentes tipos de vista se explica en el Getting-Started [16].

7.4 Ficheros de internacionalización

La estructura de carpetas para las vistas se compone de una carpeta por cada entidad. Dentro de estas carpetas habrá tantos ficheros de internacionalización como idiomas vaya a soportar la aplicación. Estos ficheros tendrán la extensión .properties y tendrán por nombre “messages” para el idioma por defecto y “messages_XX”, siendo XX el código del idioma que vaya a representar.

Estos ficheros se componen del código de internacionalización en la parte izquierda, un signo igual, y el mensaje que represente ese código. Todos los ficheros de una misma carpeta tienen que tener el mismo número de mensajes, y haber una correspondencia directa uno a uno, para que todos los mensajes aparezcan en todos los idiomas y la aplicación no dé conflictos.

7.5 Etiquetas

En este framework se proveen de etiquetas para usar en los ficheros jsp y facilitar la tarea de la creación de las vistas. Estas etiquetas están ubicadas en la carpeta webapp/WEB-INF/tags/template. No se deben modificar y si se quiere añadir etiquetas personalizadas, se hará en la carpeta custom ubicada en la misma ruta. Para hacer uso de las etiquetas, hay que escribir lo siguiente: <sprouts:nombre_etiqueta>.

Pasaremos ahora a describir todas las etiquetas, los parámetros que pueden contener y su uso.

7.4.1 Action-button

Esta etiqueta se usa junto con las tablas de listado para especificar las acciones que se pueden realizar sobre el elemento seleccionado en la tabla. Los parámetros de los que dispone esta etiqueta son: *code* y *url*.

- *Url*: parámetro obligatorio. Dirección a la que se accederá cuando se pulse el botón. En esta URL se le puede indicar el identificador del objeto que está siendo tratado en la vista mediante un número entre llaves (0 para el ID de la entidad principal), que será sustituido por el identificador de la entidad que se haya seleccionado en la tabla.
- *Code*: parámetro obligatorio. Código de internacionalización que determinará el mensaje que aparecerá en el botón.

7.4.2 Alert

Esta etiqueta se usa para mostrar un mensaje de alerta, ya sea para mensajes de éxito o mensajes de error. Los parámetros que recibe esta etiqueta son: *code*, *arguments* y *type*.

- *Code*: parámetro obligatorio. Código de internacionalización que contendrá el mensaje que aparecerá en el mensaje de alerta.
- *Arguments*: parámetro opcional. Permite modificar los delimitadores del mensaje que aparecerá en el cartel de alerta.
- *Type*: parámetro opcional. Indica si el cartel es de error (parámetro nulo) o si el cartel es de otro tipo, que será el pasado por este parámetro.

7.4.3 Button

Esta etiqueta se usa para mostrar un botón normal. Los parámetros que recibe esta etiqueta son: *code* y *url*.

- *Code*: parámetro obligatorio. Código de internacionalización que contiene el mensaje que se mostrará en el botón.
- *Url*: parámetro obligatorio. URI a la que se accederá tras pulsar el botón.

7.4.4 Cancel button

Esta etiqueta se usa para crear un botón que redirige al usuario a otra página. Lo más habitual es usarlo en formularios o en vistas de display para poder volver hacia atrás. Los parámetros que recibe esta etiqueta son: *code* y *url*.

- *Code*: parámetro obligatorio. código de internacionalización que contiene el mensaje que se mostrará en el botón.
- *Url*: parámetro obligatorio. URI a la que se redirige al usuario tras ser pulsado.

7.4.5 Checkbox

Esta etiqueta se usa para crear un checkbox. Los parámetros que recibe esta etiqueta son: *path*, *code*, *labelSize* y *boxSize*.

- *Path*: parámetro obligatorio. Nombre del atributo de la entidad que tomará el valor del checkbox.
- *Code*: parámetro obligatorio. Código de internacionalización del mensaje que aparecerá junto al checkbox.
- *LabelSize*: parámetro opcional. Es el ancho que ocupa el mensaje que aparecerá junto al checkbox.
- *BoxSize*: parámetro opcional. Es el ancho que ocupa el checkbox.

7.4.6 Data-column

Esta etiqueta se usa dentro de la etiqueta *data-table* para definir las columnas de esta. Los parámetros que recibe esta etiqueta son: *code*, *path*, *format*, *sortable*, *width*, *sortBy*, *nullCode*, *trueCode*, *falseCode*, *outFormat*, *toShow*, *imgSize*.

- *code*: parámetro obligatorio. Código de internacionalización que contendrá el mensaje a mostrar del título de la columna.
- *path*: parámetro opcional. Atributo de la entidad al que hace referencia la columna, y en general campo que se va a mostrar en la celda correspondiente a esta columna. Aunque sea opcional, realmente en la mayoría de ocasiones se va a necesitar.
- *format*: parámetro opcional. Indica qué tipo de formato se espera de la columna. Las posibles opciones son: ***image*** (para una columna con una imagen), ***url*** (para una columna que muestre un enlace), ***date*** (lo propio para las fechas), ***currency*** (columna con monedas), ***decimal*** (columna con números decimales). Se puede usar este atributo solo, o normalmente se pueden personalizar más las opciones de formato con otros adicionales.
- *sortable*: parámetro opcional. Indica si la tabla se puede ordenar en función de esa columna. El valor por defecto si no se indica nada es *true*.
- *width*: parámetro opcional. Indica el ancho que tendrá esa columna. En caso de no indicarse, el valor por defecto es *inherited*.
- *nullCode*, *falseCode*, *trueCode*: estos tres parámetros son opcionales. Su función es la misma: indicar el código de internacionalización que contiene el mensaje a mostrar en caso de que el campo de la columna sea nulo, falso y verdadero respectivamente.
- *outFormat*: parámetro opcional. Este parámetro no debe utilizarse solo. Siempre debe ir declarado cuando se haya indicado también el atributo *format*. Tiene distinta funcionalidad dependiendo del *format* con el que se acompañe. A continuación se detalla su utilidad según el caso:
 - Con *format='image'*: indica la URL de la imagen a mostrar, en caso de que no sea el propio valor del atributo. Si se introduce la cadena “+PATH”, esta se sustituirá por el valor del atributo. Por ejemplo: <http://imagenes.dominio/+PATH/imagen.jpg> se sustituiría por <http://imagenes.dominio/valorAtributo/imagen.jpg>

- o Con `format='url'`: mismo funcionamiento que en el caso de la imagen. En este caso, el mensaje que se pase al atributo `outFormat` es el que se va a mostrar en el enlace.
 - o Con `format='date'`: el valor que se pase por parámetro hace referencia a un código de internacionalización, donde se indica cuál es el formato que se quiere para mostrar la fecha en esa columna en cada idioma.
 - o Con `format='currency'`: el valor que se pasa por parámetro indica el prefijo de los distintos códigos de internacionalización que son necesarios para definir el formato de una moneda para esa columna.
 - o Con `format='decimal'`: mismo funcionamiento que `currency`. El valor indica el prefijo de los códigos de internacionalización para definir formato de un número decimal para esta columna.
- `toShow`: parámetro opcional. Utilizado para mostrar un mensaje “libre” en esa columna. Se puede pasar cualquier tipo de texto. Si se requiere que entre ese texto libre aparezca el valor del atributo de esa columna, utilizar la cadena “`+PATH`”, que al igual que en casos anteriores se sustituirá por dicho valor.
- `imgSize`: parámetro opcional. Usado en conjunto con `format = 'image'`. Se le indica el tamaño de la imagen a mostrar. El formato que se debe pasar es el siguiente ‘`anchoxalto`’. Por ejemplo: ‘`652x258`’. El tamaño que se indica es en píxeles.

7.4.7 Data-table

Esta etiqueta se usa para crear una tabla y listar entidades. Los parámetros que recibe esta etiqueta son: `data`, `tableId`, `idProperty`, `exportable`, `i18n`, `source` y `searchable`.

- `TableId`: parámetro opcional. Es el id que tendrá la tabla. Se genera uno por defecto si no se le indica.
- `IdProperty`: parámetro opcional. Indica el atributo que identifica a los datos que se muestran en la tabla. Si no se indica alguno, por defecto es el id.
- `Exportable`: parámetro opcional. Indica si los datos de la tabla se pueden exportar a otro tipo de formato. Por ejemplo: pdf, hoja de cálculo de Excel. Por defecto está a `true`.
- `I18n`: parámetro obligatorio. Indica un código de internacionalización, donde se debe hacer referencia al fichero, donde se definen el formato en cada idioma, para los distintos elementos que aparecen en la tabla. En la plantilla se puede hacer uso de “`datatables.language`”, que ya tiene definidos los elementos que aparecen tanto en español como inglés.
- `Source`: parámetro opcional. Sirve para indicar la fuente de los datos de la tabla. Se usa cuando hay más de una tabla en una misma vista y los datos se reciben desde otra URL. Todas estas url son del tipo ‘`/list/data.do`’.
- `Searchable`: parámetro opcional. Indica si la tabla va a tener un cuadro de búsqueda. Por defecto lo tendrá, a no ser que se le indique lo contrario.

7.4.8 Display-column

Esta etiqueta se usa en las vista de display para mostrar información variada. Los parámetros que recibe esta etiqueta son: *title*, *data*, *path*, *message* y *url*.

- *Title*: parámetro opcional. Cadena de texto que aparecerá como título del bloque de datos que se va a mostrar.
- *Data*: parámetro opcional. Cadena de texto que se va a mostrar. Puede ser algún atributo de la entidad o cualquier texto que se le quiera añadir. Los atributos de la entidad se especificarán de la siguiente forma: `${modelObject.atributo}`. Al hacer uso del atributo *data*, los datos no pasarán por los formateadores del GetController (véase el apartado 5.4).
- *Path*: parámetro opcional. Se le pasa como parámetro una cadena que representa la ruta del atributo de la entidad con la que estemos trabajando: `"modelObject.atributo"`. Al hacer uso de *path*, el valor de dicho atributo sí será formateado (véase el apartado 5.4).
- *Message*: parámetro opcional. Se puede usar junto con *data* o *url*. Si se usa con *data*, se le añadirá el contenido de *message* tras el contenido de *data*, pero si se usa junto con *url*, será el texto que se mostrará en el enlace.
- *Url*: parámetro opcional. URI que se abrirá si se clica en el enlace.

A continuación verá dos ejemplos de uso de esta etiqueta. En el primero, no se formatea el valor del atributo *gym.description*. En el segundo, sí se formatea el valor de *gym.fee*.

```
<sprouts:display-column data="${gym.description}" />
<sprouts:display-column title="${feeLabel}" path="gym.fee" />
```

7.4.9 Display-image-column

Esta etiqueta se usa en las vistas de display cuando se quiera mostrar una imagen. Los parámetros que recibe esta etiqueta son: *src*, *width*, *height*.

- *Src*: parámetro obligatorio. Es la dirección URL o ubicación relativa de la imagen en el proyecto.
- *Width*: parámetro opcional. Se usa para ajustar el ancho de la imagen.
- *Height*: parámetro opcional. Se usa para ajustar el alto de la imagen.

7.4.10 Form

Etiqueta que permite crear un formulario. Los parámetros que recibe esta etiqueta son: *modelAttribute*, *action*, *id* y *readOnly*.

- *modelAttribute*: parámetro obligatorio. Es el objeto que recogerá la información que se rellene o modifique en el formulario. Puede ser una entidad del dominio del problema o un formulario auxiliar.

- *Action*: parámetro opcional. URI en la que el controlador atenderá la petición cuando se pulsa el botón de submit.
- *Id*: parámetro opcional. Identificador que tendrá el formulario.
- *ReadOnly*: parámetro opcional. Indica si el parámetro es de lectura y escritura o solo lectura. Por defecto, es de lectura y escritura.

7.4.11 Hidden-field

Esta etiqueta se usa para ocultar un campo que no se vaya a modificar. El parámetro que recibe esta etiqueta es: *path*

- *Path*: parámetro obligatorio. Atributo de la entidad que se quiere ocultar.

7.4.12 Password-input

Se usa para incluir un campo de contraseña. Los parámetros que recibe esta etiqueta son: *path* y *code*.

- *Path*: parámetro obligatorio. Atributo de la entidad sobre la que se guardará la contraseña introducida en el campo.
- *Code*: parámetro obligatorio. Código de internacionalización que contiene el mensaje que aparecerá junto al campo.

7.4.13 Pictures-list

Se usa cuando se quiere mostrar una lista de imágenes a través de sus urls. Los parámetros que recibe esta etiqueta son: *code*, *pictures*, *size*, *width*, *height*.

- *code*: parámetro opcional. Es el mensaje que aparecerá en el botón que aparece junto a cada imagen.
- *Pictures*: parámetro obligatorio. Es la lista con las urls de las imágenes que se van a mostrar.
- *Size*: parámetro obligatorio. Número de imágenes que están contenidas en la lista anterior.
- *Width*: parámetro opcional. Ajusta el atributo width de cada imagen.
- *Height*: parámetro opcional. Ajusta el atributo height de cada imagen.

7.4.14 Protected

Se usa para proteger un campo de un formulario contra el hacking-post. Si un campo está marcado con esta etiqueta, si se modifica su valor, se producirá un error. Los parámetros que recibe esta etiqueta son: *path*.

- *Path*: parámetro obligatorio. Es el atributo de la entidad que se quiere proteger contra el hacking-post.

7.4.15 Select

Se usa para incluir un select en un formulario. Los parámetros que recibe esta etiqueta son: *path*, *code*, *items*, *itemLabel*, *readonly*, *id*, *onchange*, *labelSize* y *boxSize*.

- *Path*: parámetro obligatorio. Atributo de la entidad a la que se le asignará el valor del elemento seleccionado en el select.
- *Code*: parámetro obligatorio. Mensaje de internacionalización que contiene el mensaje que aparecerá junto al select.
- *Items*: parámetro obligatorio. Lista que contiene los posibles elementos a seleccionar en el select.
- *ItemLabel*: parámetro obligatorio. Atributo de la entidad que aparecerá representando al elemento en cuestión.
- *Readonly*: parámetro opcional. Atributo que indica si el select es de solo lectura y por lo tanto no se puede seleccionar nada.
- *Id*: parámetro opcional. Atributo de la entidad que realmente se manda como valor de esa selección al servidor.
- *Onchange*: parámetro opcional. Atributo que se usa para poder llamar a una función javascript cuando sea necesario.
- *LabelSize*: parámetro opcional. Ajusta el ancho del texto que aparece junto al select.
- *BoxSize*: parámetro opcional. Ajusta el ancho del campo select.

7.4.16 Social-account-sign-in

Se usa para incluir los botones de inicio de sesión con redes sociales o vincularlas con una cuenta local existente. Los parámetros que recibe esta etiqueta son: *twitter*, *google*, *isSignIn*.

- *Twitter*: parámetro opcional. Cuando el parámetro sea verdadero, indica que se quiere mostrar el botón relativo a Twitter.
- *Google*: parámetro opcional. Cuando el parámetro sea verdadero, indica que se quiere mostrar el botón relativo a Google.
- *IsSignIn*: parámetro opcional. Cuando el parámetro sea falso, indica que se está haciendo uso de los botones para vincular una cuenta local con la red social. En caso que el parámetro sea verdadero, indica que se va a hacer uso de él para iniciar sesión directamente con la red social.

7.4.17 Submit-button

Se usa para incluir el botón de envío en un formulario. El parámetro que recibe esta etiqueta es: *code*.

- *Code*: parámetro obligatorio. Código de internacionalización que contiene el mensaje que aparecerá en el botón.

7.4.18 Submit-or-cancel

Se usa para incluir el botón de envío en un formulario, además de un botón de cancelar que redirige a la url especificada. Los parámetros que reciben esta etiqueta son: *submitCode*, *name*, *backUrl*.

- *SubmitCode*: parámetro obligatorio. Código de internacionalización que contiene el mensaje que aparecerá en el botón.
- *Name*: parámetro opcional: Indica el nombre que se le asignará al botón.
- *BackUrl*: parámetro opcional. Indica la URI a la que se redirige cuando se pulsa este botón.

7.4.19 Textarea-input

Se usa para incluir un área de texto en un formulario. Los parámetros que reciben esta etiqueta son: *path*, *code* y *readonly*.

- *Path*: parámetro obligatorio. Atributo al que se le asignará el contenido del área de texto.
- *Code*: parámetro obligatorio. Código de internacionalización que contiene el mensaje que aparecerá junto al área de texto.
- *Readonly*: parámetro opcional. Indica si el área de texto es de solo lectura.

7.4.20 Textbox-input

Se usa para incluir un campo de texto en un formulario. Los parámetros que reciben esta etiqueta son: *path*, *code*, *disabled* y *readonly*.

- *Path*: parámetro obligatorio. Atributo al que se le asignará el contenido del campo de texto.
- *Code*: parámetro obligatorio. Código de internacionalización que contiene el mensaje que aparecerá junto al campo de texto.
- *Readonly*: parámetro opcional. Indica si el campo de texto es de solo lectura.
- *Disabled*: parámetro opcional. Indica si el campo de texto está deshabilitado.



Las etiquetas que están presentes en la plantilla del framework y que no han sido explicadas anteriormente, se mantienen por compatibilidad con versiones anteriores del framework, pero actualmente no están en uso.

7.5 Clases relacionadas con las vistas

Existen dos clases que merece la pena mencionar, son clases de la que se hacen uso en niveles superiores relativos a las vistas. Estas clases son *TilesUtils* y *ViewParser*.

Pasamos a describir brevemente el funcionamiento de los métodos de la clase *ViewParser*.

- *getDeclaredFields*: este método se encarga de extraer los atributos que se declararon en el fichero .jsp. Para ello hace uso del método de *TilesUtils*, *getTemplateDefinition* para obtener un objeto de tipo *definition* que contiene información acerca de la ruta del fichero. Haciendo uso del método *getJspPath* de *TilesUtils* se obtiene la ruta absoluta del fichero .jsp para poder ser analizado. Se obtiene también con el método de *TilesUtils* *isReadOnly* a través del atributo *definition* anterior si era de solo lectura el fichero. Con estos datos podemos llamar al método *getFieldsByTags* que parsea el fichero para obtener los atributos declarados.
- *getProtectedFields*: es similar al método anterior, pero en esta ocasión se llama a otra variante del método *getJspPath* que ya se encarga de crear el *definition* y con ello realizar la misma operación que en el método anterior, por lo tanto es indiferente usar un método u otro. La principal diferencia está en que ahora se llama al método *getFieldsByTags* pasándole un patrón diferente para detectar los campos que fueron declarados con la etiqueta *<sprouts:protected>*.
- *getFieldsByTag*: este método recibe por parámetros un *path*, *openingTag* y *closingTag*, y tiene por objeto devolver los atributos que se definieron en el fichero que está en la ruta que indica el parámetro *path*. Para poder tratar con este fichero, es necesario transformar en una cadena que sea legible. Para lograrlo, se usa uno de los constructores de *String*, que recibe como parámetros un Array de bytes y el conjunto de caracteres con los que se quiere codificar. Después de esto, es necesario extraer el texto que estuviese entre las etiquetas de apertura y cierre, definidas por los parámetros *openingTag* y *closingTag*, haciendo uso del método de *StringUtils*, *substringsBetween*. Una vez obtenidos los campos que fueron declarados, se añaden a un conjunto y se devuelve.

Métodos de la clase *TilesUtils* que son utilizados para determinadas tareas en lo que respecta a la construcción de tablas:

- *getJspPath*: este método recibe por parámetros una *Definition*, una HTTP request y una HTTP response. Devuelve una cadena que representa la ruta absoluta al fichero .jsp que se necesita. Primero, se necesita obtener un *ClassLoader*, que es un objeto responsable de cargar las clases. A continuación, se necesita el atributo del cuerpo, para lo cual se obtiene de la *Definition* que se pasa por parámetros. De este atributo del cuerpo, lo único que realmente se requiere es la representación como cadena. De esta representación como cadena se elimina el prefijo “..”. Una vez que se ha realizado, se puede obtener la ruta relativa, a partir de la constante en la que se almacena el nombre de la carpeta de las vistas y concatenando “/” y el cuerpo con el prefijo anteriormente eliminado. A partir de esta ruta relativa es sencillo obtener la ruta absoluta. Para ello se hace uso del método *getResource* del *classLoader* que se obtuvo al comienzo. Cuando ya se ha conseguido la ruta absoluta, tan solo quedan dos pasos más: eliminar la “/” del comienzo de la misma, y por último decodificarla con el conjunto de caracteres UTF-8, porque de lo contrario aparecerían símbolos extraños en la ruta.
- *getTemplateDefinition*: este método tiene como parámetros el nombre de la vista, una HTTP request y una HTTP response. Su cometido es devolver una *Definition* con los datos necesarios referentes a una vista. En primer lugar, se llama al método

getViewDefinition de esta misma clase. Como resultado, se obtiene una *Definition*, que tiene una estructura similar a la de un mapa con distintos pares clave=valor. Habrá tantos pares como sea necesario para que la estructura de la vista quede claramente definida. Posteriormente, se obtiene el atributo correspondiente al cuerpo de esa *Definition* que se tenía antes. De este cuerpo, lo único con lo que interesa quedarse es con el valor que devuelva como cadena. Finalmente, se realiza una comprobación. Si la representación como cadena del cuerpo no acaba en ".jsp", se llama al método *getViewDefinition*, si no fuese así, se devuelve directamente la *Definition* que se consiguió en un primer momento.

- *getJspPath*: este método recibe por parámetros un string *viewName*, una HTTP request y una HTTP result. Este método es parecido al que se explicó anteriormente, pero en esta ocasión se obtiene el *Definition* mediante el método *getTemplateDefinition* y *viewName*. Una vez obtenida la *definition*, se llama al método *getJspPath* anteriormente descrito, devolviendo así la ruta absoluta al fichero cuyo nombre de la vista era el pasado por parámetro.
- *isReadOnly*: este método tiene como parámetros *definition*, y tiene como objeto devolver un booleano indicando si la vista es de solo lectura. Para ello, se busca en el atributo *definition* el atributo *readOnly* para obtener su valor. Después de esto, se devolverá true cuando el atributo tuviese el valor true, y false en caso contrario.
- *getViewDefinition*: este método tiene como parámetros *viewName*, HTTP request y HTTP response y tiene por objeto devolver un objeto de tipo *Definition* de un determinado archivo tiles. Para ello es necesario obtener el objeto *appContext* a través de la *request*. Después de esto, obtenemos todos los ficheros tiles haciendo uso del objeto anterior. A continuación se crea un objeto de tipo *ServletRequest* pasándole como parámetros *appContext*, *request* y *response*. Este objeto será necesario para poder crear el objeto *definition*. Por último, se crea el objeto de tipo *definition* del archivo tiles donde estuviese definida la vista que se ha pasado por parámetro. Para ello, a través del objeto que contenía todos los archivos tiles, el objeto *container*, se invoca al método *getDefinition* con el nombre de la vista y el objeto *ServletRequest*.

7.6 Conclusión

Tras esta sección, se conoce todo lo relacionado con las vistas, la organización que estas deben tener, la configuración que hay que realizar. También se conoce con detalle todas y cada una de las etiquetas que están disponibles de forma predefinida en el framework para facilitar la creación de las vistas. Además, también se conocen los detalles de implementación de las clases que se encargan de parte de la gestión de los ficheros relacionados con las vistas.

8 INTEGRACIÓN CON OTROS COMPONENTES

8.1 Introducción

En el siguiente apartado vamos a explicar cómo se ha realizado la integración con componentes externos para poder hacer un uso rápido de ellos y darle un valor añadido al framework, explicando además cómo hacer uso de estos de forma sencilla. Los componentes que se han integrado son: Spring Social, Drools y Hibernate-search.

8.2 Integración con Spring Social

Para realizar la integración con Spring Social, en primer lugar se han añadido las siguientes librerías al fichero *pom.xml*: *spring-social-core 1.1.4*, *spring-social-web 1.1.4*, *spring-social-config 1.1.4*, y *spring-social-security 1.1.4*. Para realizarla con Twitter y Google como es el caso, se han añadido además: *spring-social-twitter 1.1.0*, *httpclient 4.3.6* y *spring-social-google 1.0.0*.

La integración se ha realizado de forma manual, es decir, se ha adaptado la clase *ProviderSignInController* para hacer uso de los propios repositorios en lugar del que proveía Spring Social.

En primer lugar, se modifica la url a la que se redirige al usuario cuando la operación se realiza correctamente o en caso de fallo. Esto se hace mediante la incorporación de dos atributos llamados *signInUrl* y *signInUrlAction*. Estos atributos se obtienen de la clase de configuración *SocialConfig* (que explicaremos más adelante). Tanto en los métodos *oauth1Callback* como *oauth2Callback*, se modifica la url a la que se redirige en caso de fallo, haciendo uso para ello de los atributos anteriormente mencionados.

El método *handleSignIn* es el que ha sufrido la mayor parte de las modificaciones para adaptarlo al uso de los repositorios propios, ya que es el encargado de gestionar que ocurre cuando la autenticación se ha realizado correctamente en la red social y esta nos devuelve los datos del perfil. Pasamos a describir el funcionamiento del método:

En primer lugar se recuperan la cuenta social asociada a esa red social en el sistema, un objeto de tipo *UserProfile* que contendrá toda la información de la cuenta del usuario en la red social y si hay algún usuario logueado en los instantes que se realiza la operación. Una vez obtenidos estos datos, se distingue entre varios casos.

- Si el usuario principal obtenido antes no es nulo, es decir, hay alguien logueado, quiere decir que está vinculando una red social con su cuenta local. Puede darse dos casos a su vez:
 - En el caso en que la social account anteriormente obtenida no sea nula y esté entre las cuentas sociales del usuario logueado, quiere decir que la cuenta ya

estaba vinculada, por lo tanto se actualiza mediante la información contenida las variables *principal*, *profile* y *connection*.

- o En el caso en el que no estuviese entre las cuentas del usuario, se añade una nueva cuenta social.

Una vez realizada una de las dos opciones anteriores, se redirige al usuario a la página principal de la aplicación.

- Si el usuario principal obtenido antes es nulo, es decir, no hay nadie logueado, quiere decir que está iniciando sesión con la red social, y pueden darse dos casos:
 - o Si la cuenta social es nula, es que es la primera vez que accede al sistema con esa cuenta, por lo tanto hay que crear un nuevo usuario y su correspondiente cuenta social con los datos de las variables *connection* y *profile*. Una vez realizado esto, se autentica al usuario y se le redirige a la página principal de la aplicación.
 - o Si la cuenta social no es nula, es que no accede por primera vez con esa cuenta, por lo tanto, se actualiza por si cambió algún dato desde la última conexión. Una vez realizado esto, se redirige al usuario a la página principal de la aplicación.

Todo lo explicado anteriormente, han sido las modificaciones que se han hecho para integrar Spring Social de manera manual, utilizando repositorios propios, ahora vamos a explicar cómo configurar Spring Social e integrar nuevas redes sociales. Para ello, se dispone de la clase *SocialConfig*, ubicada en el paquete social.

Esta clase, es una clase de configuración de Spring y como tal, lleva la anotación *@Configuration*, además de *@EnableSocial*, etiqueta que se usa para importar la configuración de *SocialConfiguration*. En primer lugar, esta clase dispone de varios atributos estáticos al comienzo de esta. Estos atributos son las claves y claves privadas provistas por la red social. También, hay dos atributos que permite configurar las urls de las que hará uso *ProviderSignInController*.

Esta clase tiene una serie de métodos:

- *connectController*: es un método que se encarga de crear un controlador del tipo *ConnectController*, necesario para realizar la conexión con las redes sociales que se integran.
- *addConnectionFactory*: en este método se añaden las factorías que se encargan de manejar la red social. Por lo tanto, hay que añadir en este método tantas factorías como redes sociales se quieran integrar. Cada una de estas factorías se suelen crear mediante dos claves, una de ellas privada, provistas por la red social cuando se crea la aplicación.
- *getUserIdSource*: método que asigna un identificador al usuario actual.

- *getUsersConnectionRepository*: con este método se configura que se usará un repositorio en memoria y no se persistirán los datos, para así poder hacerlo de forma manual con los repositorios propios en lugar de los provistos por Spring Social.

Para hacer uso de Spring Social en la aplicación se debe haber configurado en la clase *SocialConfig* las redes sociales que se quieran integrar, añadiendo para ello las claves y claves privadas proporcionadas por la redes sociales. Luego, hay que mostrar un botón para permitir el inicio de sesión mediante una red social. Para ello, ya existe la etiqueta *social-account-sign-in* con los botones para twitter y google. Si se quieren añadir más botones para otras redes sociales o hacerlo manualmente, hay que incluir incluir un formulario con un botón de submit. Ese formulario debe tener como acción la siguiente uri: */signin/ nombreRedSocial.do*.

```
<form id="tw_signin" action="" method="POST">
    <button type="submit" class="btn btn-twitter">
        <i class="fa fa-twitter"></i> | <jstl:out value="${twittermsg}" />
    </button>
</form>
```

Para los añadir los botones para iniciar sesión con las distintas redes sociales, se ha hecho uso de una librería, *Social Buttons for Twitter Bootstrap 3*^[21]. Está añadida al proyecto, en caso de necesitar añadir más botones.

8.3 Creación de reglas de negocio con Drools

Este framework está integrado con Drools, que permite controlar reglas de negocio bajo demanda. Por defecto está deshabilitado, pero se puede habilitar fácilmente. Para ello seguiremos los siguientes pasos:

- Buscaremos el fichero *pom.xml* situado en la raíz del proyecto, o lo buscamos con el atajo de teclado CTRL + Shift + R. Procedemos a buscar la parte comentada perteneciente a los seis componentes de Drools, situados al final del documento, y los descomentamos.
- Ahora habrá que actualizar el proyecto mediante Maven para que incluya las librerías que acabamos de descomentar. Para ello seleccionamos el proyecto, hacemos clic derecho sobre él. Vamos hasta el apartado Maven y seleccionamos *Update Project...*. Saldrá una ventana emergente en la que hay que seleccionar *Force Update of Snapshots/Releases* y pulsamos aceptar. Tardará un poco en descargar las librerías.
- Una vez que disponemos de las librerías de Drools en el proyecto, buscaremos el archivo *kmodule.xml* situado en *src/main/resources/META-INF* y procederemos a descomentarlo. En este archivo podemos cambiar la carpeta donde estarán situadas las reglas de negocio, así como el nombre de la sesión, en este caso en la carpeta *rules* y *KSession*.

```
<kbase name="KBase" packages="rules">
    <ksession name="KSession" />
</kbase>
```

Ahora procederemos a buscar el archivo *RulesConfig.java* y a descomentarlo. Este archivo se encarga de configurar *KieContainer*, que será necesario más adelante.

Una vez realizados los pasos anteriores, Drools está totalmente habilitado para su utilización bajo demanda. El siguiente paso es crear las reglas de negocio en ficheros *.drl* y situarlos en la carpeta *resources/rules*. Recordamos que la carpeta en la que se especifican las reglas de negocio se puede modificar en el fichero *kmodule.xml*. Para la creación de las reglas de negocio, les remitimos a la documentación oficial de Drools^[4].

Las reglas de negocio usan objetos almacenados en memoria. Cuando detectan que existen objetos en memoria que cumplen las condiciones de alguna de las reglas, entonces entra en acción los efectos de la regla en cuestión. Estos objetos son incluidos en memoria por nosotros mismos en los servicios. Vamos a proceder a explicar cómo realizar este proceso.

- En el servicio dónde se vaya a usar Drools, tendremos un atributo del tipo *KieContainer*, anotado con la etiqueta *@Autowired*.

```
@Autowired
private KieContainer kieContainer;
```

- También tendremos un atributo *KieSession*, que instanciaremos mediante el anterior *KieContainer* y utilizando la *KieSession* configurada en el archivo *kmodule.xml*. Lo realizaremos de la siguiente forma:

```
KieSession kieSession = kieContainer.newKieSession("KSession");
```

- Con el objeto *KieSession* ya podemos añadir todos los objetos que necesitemos a memoria. Para ello utilizaremos el método *insert* de *KieSession*. Lo realizaremos de esta forma:

```
kieSession.insert(customer);
```

- Una vez hayamos añadido todos los objetos que hagan falta después en las reglas de negocio, procederemos a ejecutarlas usando para ello el método *fireAllRules* de *KieSession*.

```
kieSession.fireAllRules();
```

Estos pasos lo realizaremos en todos los servicios que sea necesario, incluyendo los objetos que hagan falta en cada caso personalizado.

8.4 Búsquedas full-text

Para la integración con Hibernate-Search ha sido necesario incluir o actualizar las dependencias de ciertas librerías en el fichero *pom.xml*. Ha sido necesario incluir la librería *Hibernate-search-orm 5.3.0*, *Hibernate-core 4.3.10* y actualizar la librería *Hibernate-entity-manager* a la versión *4.3.10*.

La integración con dicho componente para dar soporte a las búsquedas full-text se ha realizado en la clase *AbstractService* mediante el método *fullTextSearch*. Pasaremos a explicar el funcionamiento del mismo:

Los parámetros que se reciben son: uno de tipo *Pageable*, necesario para devolver la consulta paginada; *searchCriteria*, que es la cadena de texto que el usuario introduce para realizar la búsqueda; *luceneQueryStr*, que es la consulta lucene para realizar un filtrado a la consulta; *clazz*, siendo la clase del objeto sobre la que se realizará la consulta; *fields*, son los campos sobre los que se realizará la búsqueda. El funcionamiento interno del método es el siguiente:

- En primer lugar, si *searchCriteria* está vacío, se devuelven los resultados de una consulta por defecto, que serán todos los elementos disponibles en la base de datos. En otro caso, se procede a comenzar el proceso. Para ello se crea una sesión de Hibernate a través del objeto *EntityManager* declarado al inicio de la clase, necesario para poder crear un objeto de tipo *FullTextSession*, que se utiliza para crear un objeto de tipo *SearchFactory*.
- Una vez obtenido el objeto *searchFactory*, se procede a crear un objeto de tipo *MultiFieldQueryParser* pasándole como parámetros los campos donde se realizará la búsqueda y un analizador de la clase, usando para ello el objeto *SearchFactory* anteriormente obtenido. Este objeto nos permitirá crear la consulta que realizará la filtración por la condición impuesta en el parámetro *luceneQueryStr*.
- Antes de proceder con la creación de las consultas, es necesario generar los índices de lucene para que estas funcionen bien, para ello se crea un objeto de tipo *FullTextEntityManager* pasándole como parámetro el *entityManager* declarado al principio de la clase. Una vez creado el objeto, podemos llamar al método *createIndexer().startAndWait()* para generar los índices de lucene.
- A continuación, se crea un objeto de tipo *QueryBuilder* para la clase pasada por parámetro. Este objeto servirá para combinar la consulta normal y la filtrada. Después, creamos un objeto de tipo *TermMatchingContext*, que será un objeto auxiliar al que se le indican los campos sobre los que se realizará la búsqueda en cuestión. Después, es hora de obtener la consulta que filtra por la condición pasada por parámetro, haciendo uso del objeto *MultiFieldQueryParser* creado anteriormente. Además, es necesario obtener la consulta que buscará aquellos objetos donde el *searchCriteria* coincida en alguno de los campos especificados, por lo tanto, mediante el objeto auxiliar de tipo

TermMatchingContext creamos una consulta que devolverá aquellos objetos que cumplen con las palabras claves en alguno de los campos especificados.

- Una vez obtenidas las dos consultas, la que filtra por el criterio de búsqueda pasado en el parámetro *luceneQueryStr* y la consulta que filtra por coincidencia de las palabras claves contenidas en el parámetro *searchCriteria* en alguno de los campos especificados, es necesario unirlas para obtener la intersección de los resultados de ambas. Para ello es necesario el objeto *QueryBuilder* generado anteriormente que unirá ambas consultas en una.
- A partir de esta consulta, hay que crear la consulta definitiva, que será del tipo *FullTextQuery*, que será la que devuelva los resultados.
- A continuación, con los resultados de la búsqueda y el objeto pageable que se pasó por parámetro, se crea un objeto de tipo *Page* para devolver los resultados paginados.

Para hacer uso de las búsquedas full-text es necesario incluir las anotaciones necesarias en la clase sobre la que se realizará la búsqueda o sus clases relacionadas. Para conocer qué anotaciones hay que incluir, diríjase a la documentación de Hibernate Search^{[18][19]}.

Una vez que esté configurada la clase sobre la que se realizará la búsqueda, es necesario llamar desde el servicio que lo requiera a uno de estos dos métodos: *fullTextSearch(Pageable pageable, String searchCriteria, Class<?> clazz, String... fields)* en cuyo caso no se realizará ningún filtrado ó *fullTextSearch(Pageable pageable, String searchCriteria, String luceneQueryStr, Class<?> clazz, String... fields)* donde sí se realizará un filtrado. Tenga en cuenta que la cadena que representa el filtrado, debe seguir el lenguaje de consultas de Lucene^[17].

En el siguiente ejemplo se ilustra la aplicación de búsquedas full-text aplicadas a objetos de tipo *Barter* que no estén cancelados. Puede encontrar dicho ejemplo en el proyecto Acme-Barter, en la clase *BarterService*:

```
@Override  
public Page<Barter> findPage(Pageable page, String searchCriteria) {  
    Page<Barter> result;  
    result = fullTextSearch(page, searchCriteria, "cancelled:false",  
        Barter.class, "title", "offered.name", "offered.description",  
        "requested.name", "requested.description");  
    return result;  
}
```

8.5 Conclusión

Después de haber leído esta sección, conocerá los detalles de la integración con estos componentes, el potencial que pueden tener así como las limitaciones. Además, lo más importante, sabrá cómo utilizar estos componentes sin ningún esfuerzo.

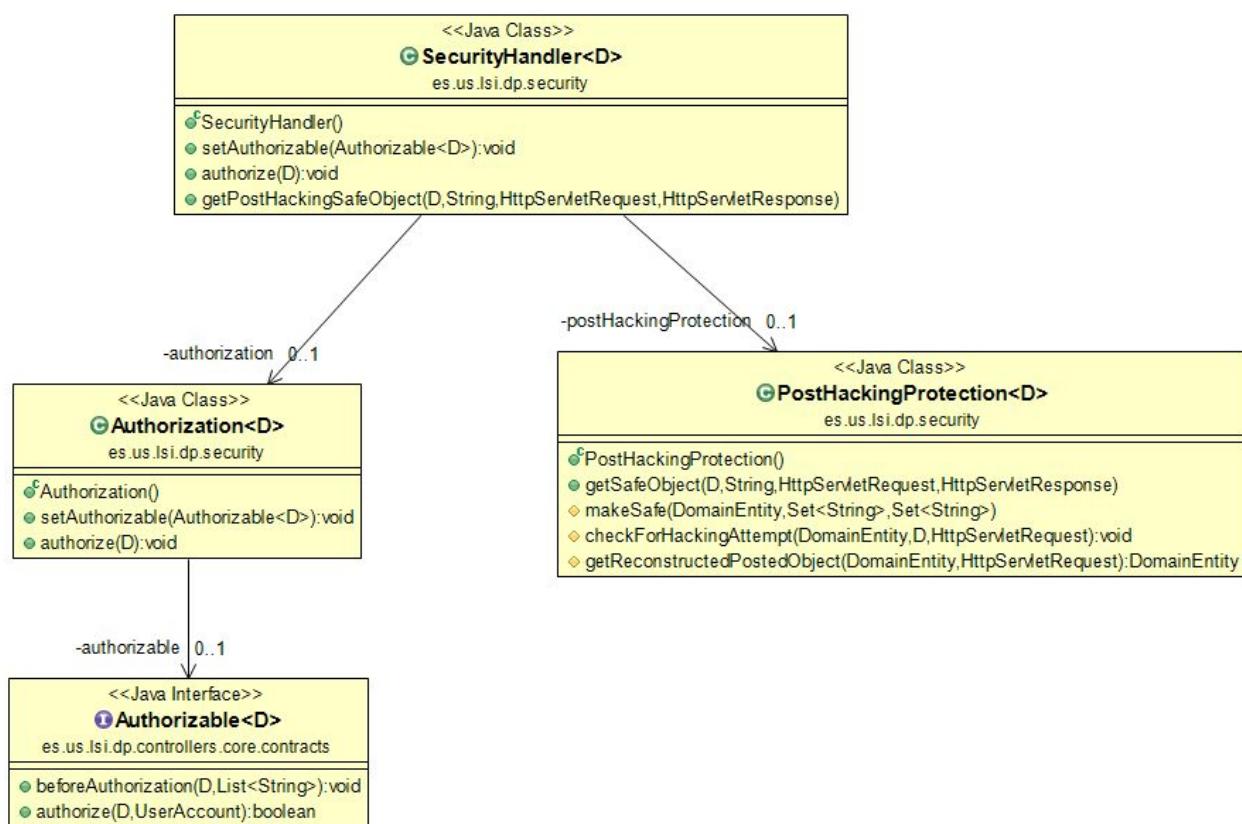
9 SEGURIDAD Y PREVENCIÓN DE HACKING

9.1 Introducción

En el presente capítulo, hablaremos sobre los mecanismos de seguridad y prevención de hacking que ofrece Sprouts Framework. En concreto, ofrece mecanismos que permiten proteger del hacking post y de que el usuario acceda a sitios donde no está autorizado. Gracias a esto, el programador no tendrá que preocuparse por implementar ningún mecanismo para proteger su aplicación de ambos tipos de hacking, simplemente tendrá que redefinir los métodos e implementar las etiquetas en la vista JSP que se le indique.

9.2 Estructura de clases

Presentamos la estructura de clases que hace posible todo esto:



Como podemos observar, la clase que se encarga de coordinar las tareas de protección es **SecurityHandler**. Esta clase está presente con modificador de acceso *protected* en **GetController**. Expliquemos cómo utiliza el framework dicha clase para la protección contra el acceso no autorizado y protección contra el hacking post.

9.2 Protección contra el acceso no autorizado

Esta protección es posible gracias a que *GetController* implementa la interfaz *Authorizable*. Al hacer esto, será necesario redefinir los métodos *beforeAuthorization* y *authorize*. Para la protección contra el acceso no autorizado, nos interesa analizar el método *authorize*. Está definido en la clase *Authorization*. Lo que hace es, llamar al método *authorize* que el desarrollador ha implementado en su controlador, es decir, el heredado por la interfaz *Authorizable*, implementada por *GetController*. Si ese método devuelve false, lanzará una excepción de tipo *HttpForbiddenException*.

9.3 Protección contra el hacking post

La protección contra el hacking post se basa en que, si el programador marca como protegidos ciertos cambios en el fichero JSP relacionado con la vista, el objeto que se persistirá en la base de datos conservará el valor original de esos campos, aunque el usuario los modifique.

Para ello, en el método *post* de *PostController*, se trabajará con un objeto seguro (*safeObject*), que se obtiene llamando al método *getPostHackingSafeObject*, al que se le pasa como parámetros el objeto que el usuario envía desde el formulario, el nombre de la vista, y dos objetos *HttpServletRequest* y *HttpServletResponse*. A este método se le llama a partir del atributo *SecurityHandler* que declara *GetController* (recordemos que *PostController* hereda de *GetController*).

El método *getPostHackingSafeObject*, llama al método *getSafeObject* de la clase *PostHackingProtection*, pasándole todos los argumentos que *getPostHackingSafeObject* recibe. Este método, obtiene, a partir del nombre de la vista y los objetos *HttpServletRequest* y *HttpServletResponse*, todos los campos declarados y los marcados como *protected*. Para ello, se llama a los métodos *getDeclaredFields* y *getProtectedFields*, respectivamente. Ambos devuelven un conjunto de *String*. A continuación, se llama a *makeSafe* de la misma clase *PostHackingProtection*. Recibe como parámetros el objeto que el usuario envió por el formulario, y los dos conjuntos de campos declarados y campos con la etiqueta *protected*. En este método se obtienen los campos “mutantes”, aquellos que no están marcados como *protected*. Con este nuevo conjunto, se le aplica al objeto de tipo *DomainEntity* que el usuario envió por el formulario el método *reconstruct* de *DomainEntity*, que recibe como parámetro el conjunto de campos mutantes. Este método reconstruye el objeto, obteniendo el que ya estaba persistido, y recorriendo cada campo mutante y cambiando su valor por el que el usuario especificó en el formulario.

Finalmente, se devuelve este último objeto, que conserva el valor original de los campos marcados como *protected*.

Por otra parte, en el caso de que necesitemos modificar el valor de un atributo que el usuario no deba modificar, por ejemplo, establecer un identificador de otra entidad, podemos utilizar el método *beforeAuthorization*. Este método, de la interfaz *Authorizable*, es

redefinido en todos los controladores de casos de uso, y, como ya se explicó tanto en el capítulo de controladores como de servicios, llaman a los métodos *beforeCreating*, *beforeUpdating* o *beforeDeleting*, dependiendo del tipo de servicio, que el usuario debe implementar en el servicio. Como ya sabemos, el método *beforeAuthorization* es llamado al obtener un objeto de la base de datos para mostrarlo en una vista y antes de persistir el objeto cuando el usuario envía el formulario.

- i Tenga en cuenta que, si en los métodos *beforeCreating*, *beforeUpdating*, *beforeDeleting*, *beforeCommitingCreate*, *beforeCommitingUpdate*, y *beforeCommitingDelete*, se modifican atributos que en el formulario estaban marcados como protegidos, los cambios aplicados sobre ambos quedarán sin efecto, puesto que se recuperará el valor que tenía ese atributo en la base de datos.

De esta manera, y siguiendo estos consejos, cualquier intento de hacking post queda completamente frustrado.

9.4 Conclusión

En este apartado hemos visto los mecanismos que ofrece Sprouts Framework para evitar el post-hacking y el acceso no autorizado. Gracias a estos, estaremos siempre protegidos contra ambos tipos de hacking, siendo necesaria la implicación del desarrollador en ambos casos, ya que para evitar el acceso no autorizado, es necesario que en los controladores implemente de forma correcta el método *authorize*, y en los ficheros JSP de las vistas, debe usar de forma correcta la etiqueta *protected*. Más allá de esto, el desarrollador puede despreocuparse de la seguridad de su aplicación web.

CONCLUSIÓN

Con Sprouts-Framework es más sencillo realizar todos los casos de uso que se puedan presentar en los proyectos de las empresas, ya que se precisa de muy poco tiempo para implementarlos respecto del framework Spring por las facilidades que ofrece. De igual forma, reduce la probabilidad de cometer errores en el código.

Además, es bastante sencillo, rápido y ágil empezar a usar este framework para una persona que no ha trabajado nunca con este framework o con Spring, ya que la documentación es bastante extensa, está bien estructurada y se explica de forma clara cada uno de los componentes de este.

Por lo tanto, teniendo en cuenta la finalidad con la que está construido este framework y la documentación que este posee, contribuirá a mejorar su productividad en entornos profesionales y en gran parte de los proyectos que quiera realizar. De esta forma será más eficiente y se podrán ahorrar costes puesto que se necesitarán invertir menos horas en cerrar un proyecto.

BIBLIOGRAFÍA

- [1] *Spring Framework Reference Documentation*. (2016). *Docs.spring.io*. Junio 2016, a partir de <http://docs.spring.io/spring/docs/3.2.17.RELEASE/spring-framework-reference/htmlsingle/>
- [2] *Spring Social Reference*. (2016). *Docs.spring.io*. Junio 2016, a partir de <http://docs.spring.io/spring-social/docs/1.1.4.RELEASE/reference/htmlsingle/>
- [3] *Hibernate Search*. (2016). *Docs.jboss.org*. Junio 2016, a partir de http://docs.jboss.org/hibernate/search/5.3/reference/en-US/html_single/
- [4] *Drools Documentation*. (2016). *Docs.jboss.org*. Junio 2016, a partir de http://docs.jboss.org/drools/release/6.4.0.Final/drools-docs/html_single/index.html
- [5] *javax.validation.constraints (Java(TM) EE 7 Specification APIs)*. (2016). *Docs.oracle.com*. Junio, 2016 de <https://docs.oracle.com/javaee/7/api/javax/validation/constraints/package-summary.html>
- [6] *javax.validation (Java(TM) EE 7 Specification APIs)*. (2016). *Docs.oracle.com*. Junio 2016, a partir de <https://docs.oracle.com/javaee/7/api/javax/validation/package-summary.html>
- [7] *javax.persistence (Java(TM) EE 7 Specification APIs)*. (2016). *Docs.oracle.com*. Junio 2016, a partir de <http://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html>
- [8] *org.hibernate.validator.constraints (Hibernate Validator 4.1.0.Final)*. (2016). *Docs.jboss.org*. Junio 2016, a partir de <https://docs.jboss.org/hibernate/validator/4.1/api/org/hibernate/validator/constraints/package-summary.html>
- [9] *jirutka/validator-collection*. (2016). *GitHub*. Junio 2016, a partir de <https://github.com/jirutka/validator-collection>
- [10] *CDI Dependency Injection @PostConstruct and @PreDestroy Example* · *Burak Aktas*. (2016). *Buraktas.com*. Junio 2016, a partir de <http://buraktas.com/cdi-dependency-injection-postconstruct-predestroy-example/>
- [11] *PostConstruct (Java Platform SE 7)*. (2016). *Docs.oracle.com*. Junio 2016, a partir de <https://docs.oracle.com/javase/7/docs/api/javax/annotation/PostConstruct.html>
- [12] *ThreadLocal (Java Platform SE 7)*. (2016). *Docs.oracle.com*. Junio 2016, a partir de <https://docs.oracle.com/javase/7/docs/api/java/lang/ThreadLocal.html>
- [13] *Spring MVC Validator with @InitBinder and WebDataBinder.registerCustomEditor Example*. (2015). *Concretepage.com*. Junio 2016, a partir de <http://www.concretepage.com/spring/spring-mvc/spring-mvc-validator-with-initbinder-webdatabinder-registercustomeditor-example>

- [14] *MessageSource (Spring Framework 4.3.0.RELEASE API)*. (2016). *Docs.spring.io*. Junio 2016, a partir de <http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/MessageSource.html>
- [15] *DataTables - Usage*. (2016). *Legacy.datatables.net*. Junio 2016, a partir de <http://legacy.datatables.net/usage/server-side>
- [16] Sprouts Framework - *Getting Started*
- [17] *Apache Lucene - Query Parser Syntax*. (2016). *Lucene.apache.org*. Junio 2016, a partir de https://lucene.apache.org/core/2_9_4/queryparsersyntax.html
- [18] *Getting started with Hibernate Search - Hibernate Search*. (2016). *Hibernate.org*. Junio 2016, a partir de <http://hibernate.org/search/documentation/getting-started/>
- [19] *Hibernate Search*. (2016). *Docs.jboss.org*. Junio 2016, a partir de https://docs.jboss.org/hibernate/search/4.2/reference/en-US/html_single/
- [20] *Apache Tiles - Framework - Tiles Concepts*. (2016). *Tiles.apache.org*. Recuperado Junio 2016, a partir de <https://tiles.apache.org/framework/tutorial/basic/concepts.html>
- [21] *Social Buttons for Twitter Bootstrap 3*. (2016). *Adamneumann.co*. Recuperado Junio 2016, a partir de <http://adamneumann.co/bootstrap-social-buttons/3/>