



 Escuela Técnica Superior de
Ingeniería Informática



Getting Started

Alumnos:

Javier Bonilla (javierbonillad@gmail.com)

Daniel Moreno (morenomdani@gmail.com)

Álvaro Valencia (alvarovalenciavp@gmail.com)

Tutor:

Rafael Corchuelo

Resumen

Sprouts es un framework para el desarrollo de sistemas de información web cuya finalidad es aumentar los beneficios y la productividad. Haciendo uso del mismo puede lograr ahorrar costes en el desarrollo, al requerir un menor tiempo para llevar a cabo el desarrollo. Además se consigue una mayor eficiencia y productividad, ya que el desarrollo con este framework hace que la implementación sea menos propensa a cometer errores. Esto se debe a que el desarrollador se debe centrar únicamente en implementar y puede olvidarse de las dificultades y problemas que suponen los ficheros de configuración.

El presente documento es una guía de introducción para que pueda dar sus primeros pasos con Sprouts Framework. Cuando termine esta guía, usted sabrá todo lo necesario para instanciar la plantilla de Sprouts y comenzar a crear casos de uso sencillos. En todo momento se hará referencia a la documentación oficial de cara a obtener información avanzada sobre cualquier aspecto del framework.

Veremos cómo preparar su workspace, cómo ha de instanciar y configurar la plantilla, cómo ejecutarla, cómo crear un modelo de dominio y cómo implementar casos de uso básicos como creación, actualización, eliminación, listado y muestra de información de entidades.

Índice

1. Introducción
2. Preparando el entorno de desarrollo
 - 2.1 Importando y configurando la plantilla
 - 2.2 Configurando el servidor de base de datos
 - 2.3 Ejecutando la plantilla
3. Modelo de dominio
4. Implementando casos de uso
 - 4.1 Creación de repositorios
 - 4.2 Creación de servicios
 - 4.3 Creación del controlador
 - 4.4 Creación de vistas (.jsp)
 - 4.5 Casos de uso de listado de entidades
 - 4.6 Casos de uso para mostrar información respecto a una entidad
 - 4.7 Casos de uso de creación de entidades
 - 4.8 Casos de uso de actualización de entidades
 - 4.9 Casos de uso de eliminación de entidades
 - 4.10 Casos de uso de mostrar información simple de entidades
5. Conclusión
6. Bibliografía

1. Introducción

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

Sprouts Framework está construido sobre Summer Framework, que a su vez está basado en Spring Framework 3. Sprouts hereda, por tanto, la filosofía principal de Summer Framework: lo provisto es inmutable, lo que implica que si el framework incluye algún archivo configurado por defecto, ese archivo no debe ser modificado. Gracias a esta filosofía, se reduce la posibilidad de cometer fallos durante el desarrollo del proyecto. Summer Framework, además, evita que el desarrollador tenga que realizar tareas repetitivas con alta probabilidad de errores. El valor añadido que Sprouts ofrece es versatilidad. Se ha llevado a cabo cambios y adaptaciones sobre la base de Summer para ofrecerle al usuario una gran versatilidad y flexibilidad. Con Sprouts, no habrá caso de uso que el desarrollador no pueda implementar. Y gracias a la base de Summer, el desarrollador no tendrá que repetir código costoso y con alta probabilidad de fallos.

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

En esta guía básica encontrará información suficiente para poner en marcha su plantilla de Sprouts Framework. Le enseñamos a montar su Workspace, a importar su proyecto en Eclipse, definir las variables de entorno necesarias, configurar su servidor de base de datos MySQL y ejecutar la plantilla.

También se presenta una guía sobre implementación de casos de uso básicos (creación, actualización, listado, borrado y muestra en pantalla de información de entidades), gracias a la cual podrá empezar a trabajar en su proyecto. **Puede encontrar información adicional acerca de detalles de la implementación o aspectos técnicos en el manual de la documentación oficial de Sprouts Framework.**

Lo que usted va a necesitar:

- Java JDK 7 update 11 o superior.^[14]
- Entorno de desarrollo integrado Eclipse Java EE Índigo (v3.7) o posterior.^[15]
- Base de datos MySQL versión 5.5 o superior.^[3]
- Servidor de aplicaciones Apache Tomcat 7 versión developer.^[4]
- Apache Maven integration for Eclipse.^[2]

Recomendamos el uso de una máquina virtual para trabajar con Sprouts Framework, con sistema operativo Windows XP o superior.

NOTA: para el desarrollo del Framework y de los proyectos de ejemplo que se presentan se ha hecho uso de una máquina virtual provista por la asignatura Diseño y Pruebas del departamento de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla.^[13]

Este documento va acompañado de dos proyectos a modo de ejemplo, y recomendamos que importe ambos en su Workspace, ya que los ejemplos presentados aquí son del proyecto Acme-Barter.

¡Adelante!

2. Preparando el entorno de desarrollo

Introducción

El primer paso para empezar a trabajar con Sprouts Framework es instanciar su entorno de trabajo, importar la plantilla que le proporcionamos y el proyecto de utilidades. Puede encontrar los tres elementos en la página web oficial de Sprouts Framework. El entorno de trabajo proporcionado incluye scripts útiles para trabajar con la base de datos y la configuración adecuada para el servidor Tomcat. Viene configurado para usar el protocolo HTTPS. La plantilla es el núcleo principal de framework, sobre la que desarrollará sus proyectos. Finalmente, el proyecto de utilidades provee herramientas para trabajar con su proyecto.

En este capítulo le enseñamos a instanciar y configurar su entorno de trabajo y a importar la plantilla y el proyecto de utilidades.

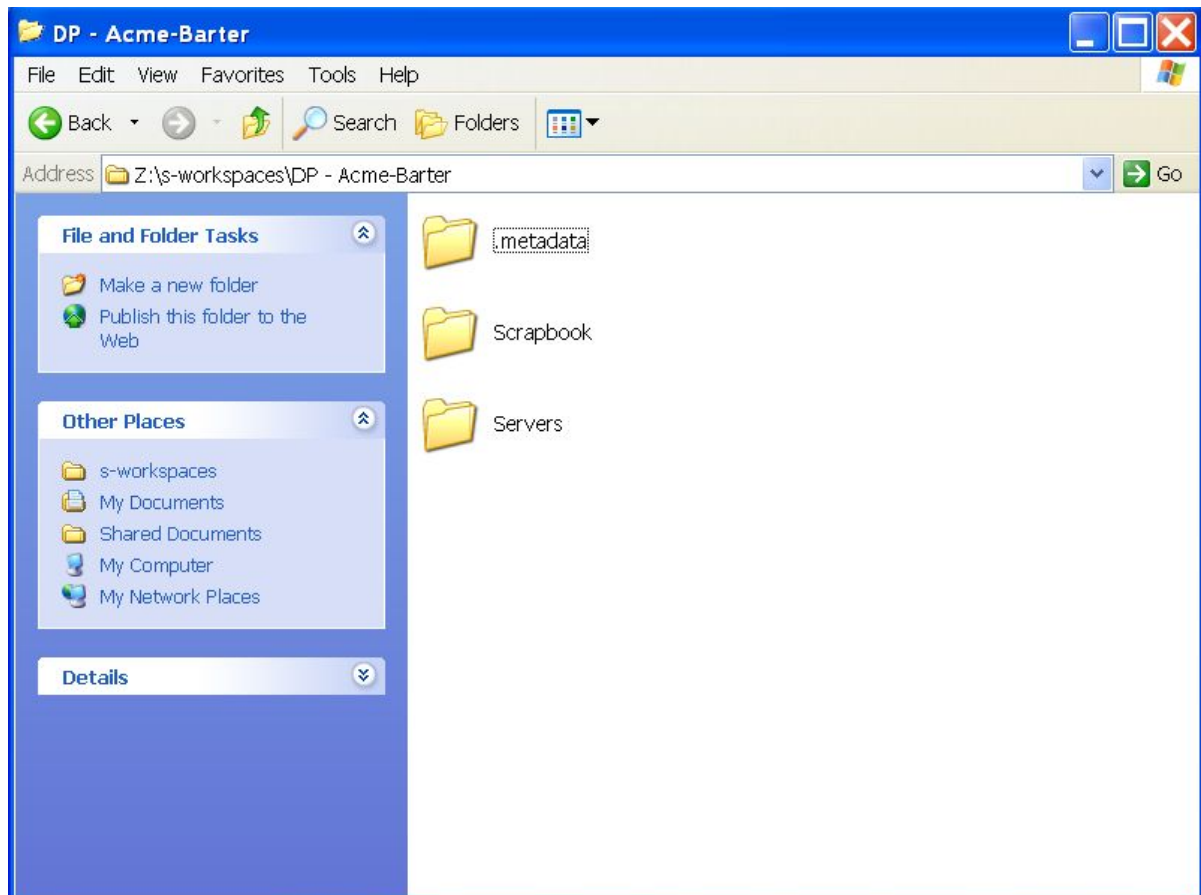
2.1 Importando y configurando la plantilla

2.1.1 Importando el workspace

Se proporciona un fichero .zip con la última versión del Workspace. Se deberá descomprimir y hacer uso de una copia de este cada vez que se quiera crear uno nuevo.



El aspecto interno de la carpeta es el siguiente:



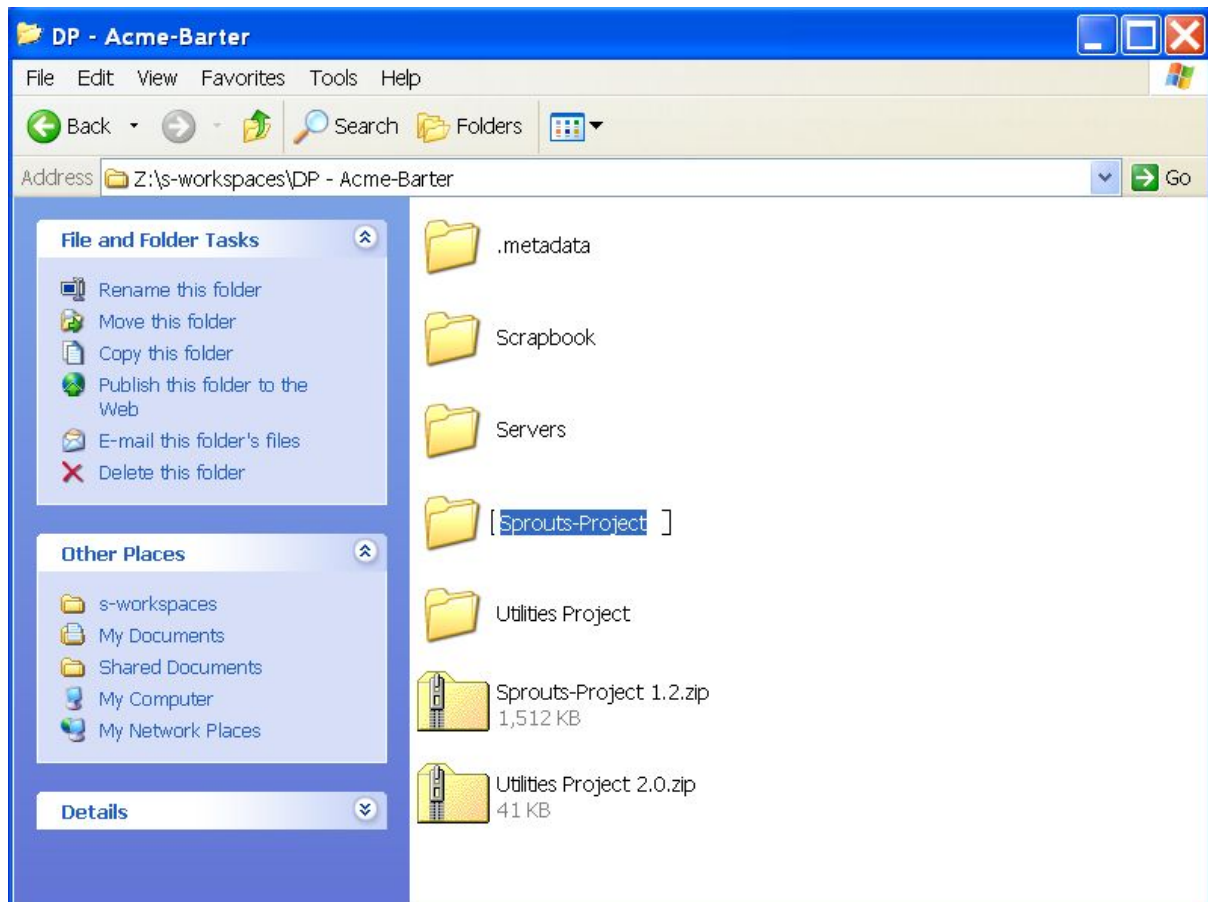
Contiene varias carpetas, una de ellas con el servidor de aplicaciones Tomcat 7 ya configurado para poder hacer uso del mismo en el proyecto. Además se ha añadido soporte para HTTPS, de forma que si se quiere se podría utilizar en el proyecto a desarrollar.

La carpeta Scrapbook incluye varias utilidades (scripts .sql) para facilitar la creación de la base de datos necesaria, así como crear y dotar de los permisos necesarios a los usuarios de la misma.

2.1.2 Importando los proyectos

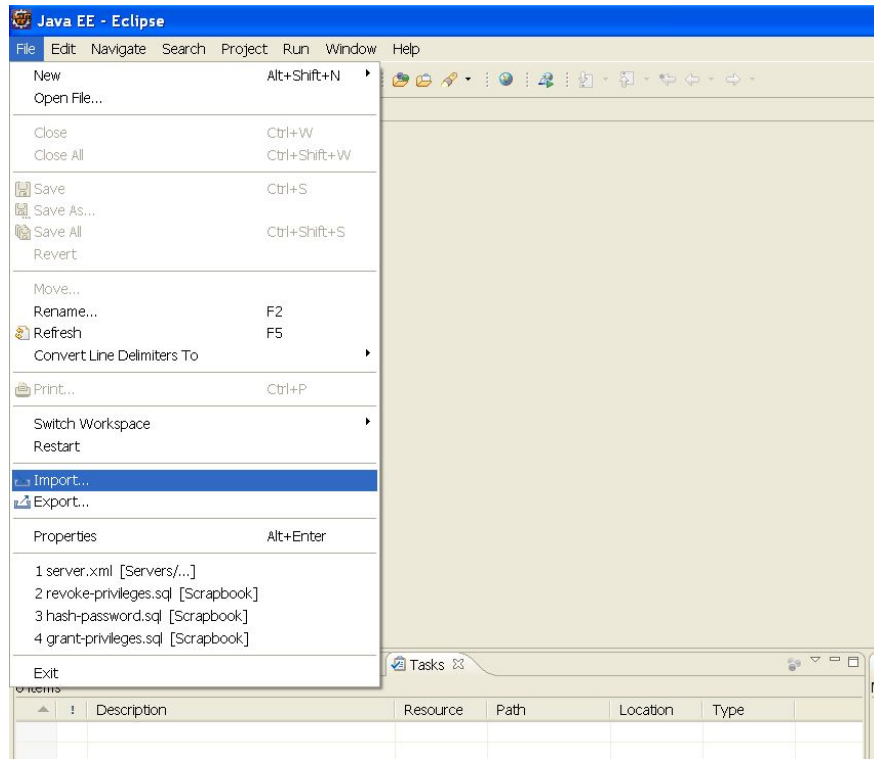
A continuación, procederemos a importar el proyecto. Para ello, dentro de nuestro workspace, pegamos los ficheros comprimidos "Sprouts-Project x.y.zip" y "Utilities-Project w.z.zip". El primero es la plantilla que ofrece Sprouts Framework, y el segundo es un proyecto que incluye utilidades que permiten poblar una base de datos, hacer consultas sobre ella, pasar una contraseña a hash o probar las búsquedas full-text.

Descomprimimos ambos proyectos:



Antes de abrir Eclipse, debemos cambiar el nombre a nuestro proyecto.

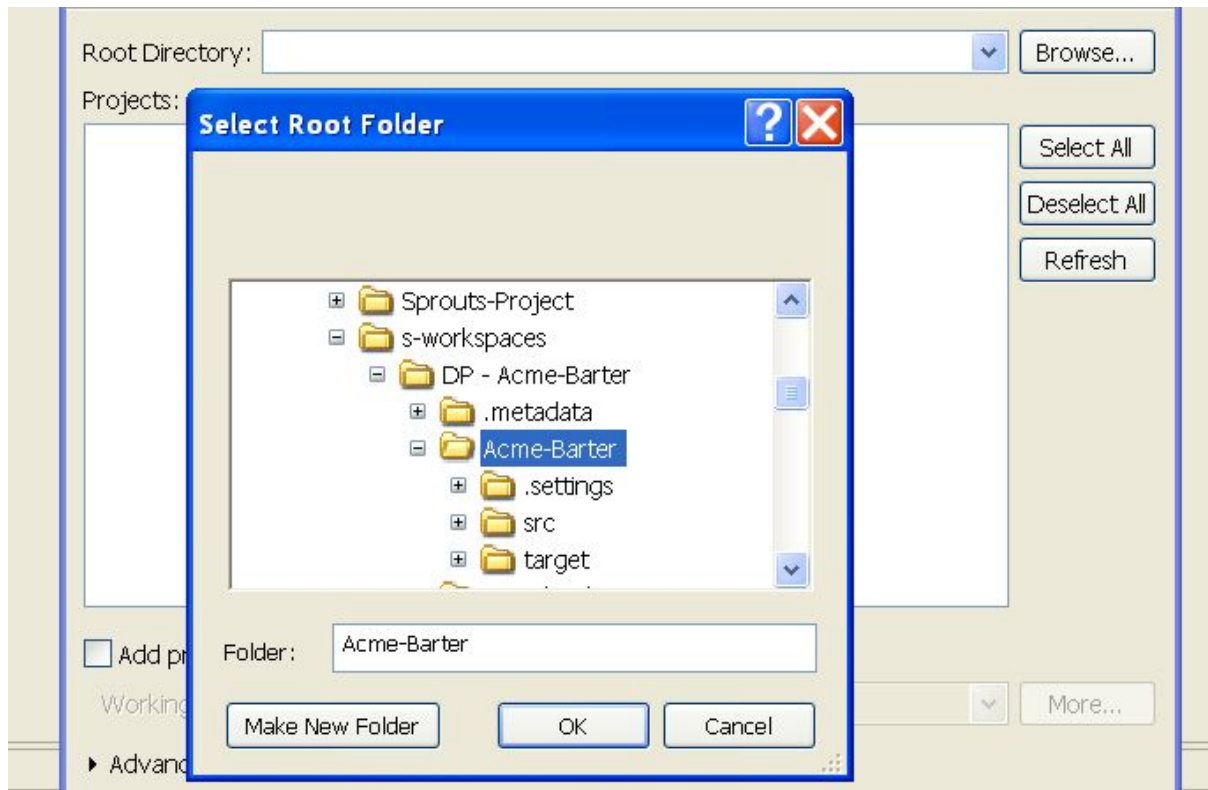
El siguiente paso es ejecutar Eclipse. Seleccionamos como Workspace el que acabamos de crear, e importamos ambos proyectos. Una vez cargado, nos dirigimos al menú File y pulsamos Import...



En la ventana que se nos ha abierto, seleccionamos Maven → Existing Maven Project.



Pulsamos el botón "Next", y en el apartado Root Directory seleccionamos el proyecto que deseemos importar.



Pulsamos "OK", y "Finish" para comenzar con la importación del proyecto.

Debemos esperar unos minutos a que Maven importe todas las dependencias que aparecen en el fichero pom.xml. Cuando Maven termine de importar el proyecto, repetiremos el mismo procedimiento para el proyecto Utilities-Project.

2.1.3 Configurando la plantilla

Una vez que se hayan importado los proyectos procederemos a configurarlos.

Sprouts-Project

En primer lugar, se debe abrir el fichero pom.xml, ubicado en la raíz del proyecto o pulsamos CTRL + Shift + R para buscar dicho fichero. Se debe modificar como mínimo el artifactId, por el nombre del proyecto. Opcionalmente, se puede modificar la versión, la URL descripción, y nombre y página web de la organización.

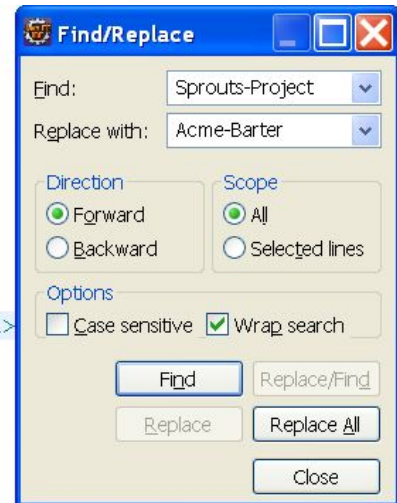
```
<!-- Artifact identification -->

<groupId>Design-and-Testing</groupId>
<artifactId>Acme-Barter</artifactId>
<version>1.0</version>
<packaging>war</packaging>

<!-- Indexing information -->

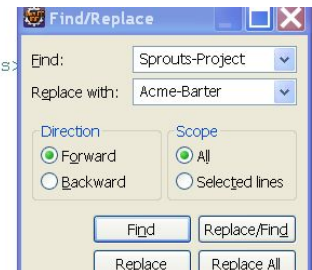
<name>Acme-Barter</name>
<url>http://www.Acme-Barter.com</url>
<description>This is the template project.</description>

<organization>
  <name>Design and Testing, Inc.</name>
  <url>http://www.dat.com</url>
</organization>
```



Por último, es necesario modificar el fichero de configuración web.xml, localizado en Webapp/WEB-INF/web.xml. Se debe modificar el nombre del servlet.

```
<servlet>
  <servlet-name>Acme-Barter</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value></param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>Acme-Barter</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```



Utilities-Project

Es necesario enlazar este proyecto de utilidad, con el proyecto principal, para ello hay que modificar el pom.xml de este proyecto. Lo encontramos en la raíz del proyecto o procedemos a buscarlo con Ctrl + Shift + R.

Debemos modificar las dependencias del proyecto. Deberemos indicarle el mismo groupId, artifactId y versión que se configuraron para el anterior proyecto para enlazarlo con este.

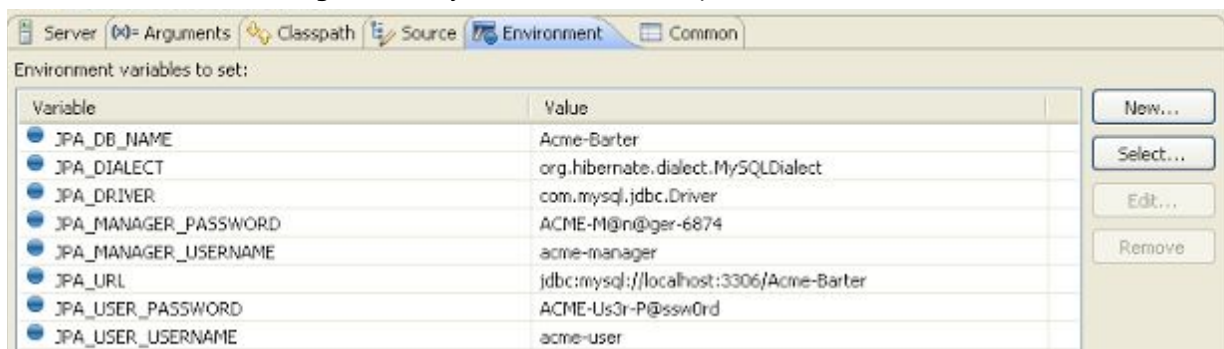


```
<?xml version="1.0" encoding="UTF-8" ?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>Design-and-Testing</groupId>
    <artifactId>Sample-Utilities-Project</artifactId>
    <version>1.0</version>
    <name>Sample Utilities Project</name>
    <url>http://www.sample-utilities-project.com</url>
    <description>This is a sample utilities project.</description>
    <dependencies>
        <dependency>
            <groupId>Design-and-Testing</groupId>
            <artifactId>Sprouts-Project</artifactId>
            <version>1.0</version>
        </dependency>
    </dependencies>
</project>
```

Configurando las variables de entorno

En el fichero datasource.xml se hace uso de unas variables de entorno, para configurar los proyectos sin necesidad de tener que tocar dicho archivo. Para poder configurar las variables de entorno hay dos posibles formas de realizarlo:

La primera de ellas es configurarlas desde el propio Eclipse, se selecciona el proyecto → Run as → Run configurations y se selecciona la pestaña Environment



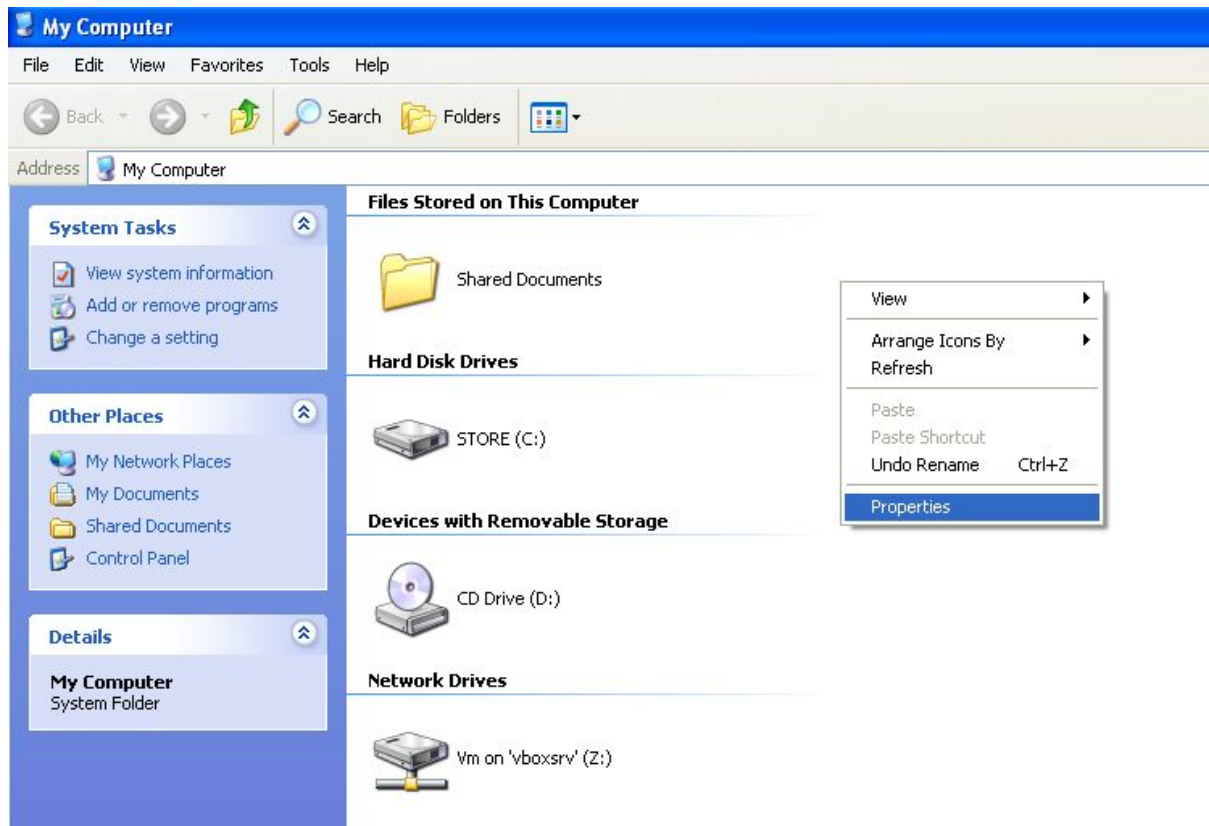
Las variables que son necesario configurar son las siguientes:

- JPA_DB_NAME: nombre de la base de datos
- JPA_DIALECT: org.hibernate.dialect.MySQLDialect
- JPA_DRIVER: com.mysql.jdbc.Driver
- JPA_MANAGER_PASSWORD: Contraseña del usuario manager de la BD
- JPA_MANAGER_USERNAME: Nombre usuario del manager de la BD
- JPA_URL: jdbc:mysql://localhost:3306/"DBNAME"
- JPA_USER_PASSWORD: Contraseña del usuario de la BD
- JPA_USER_USERNAME: Nombre de usuario de la BD

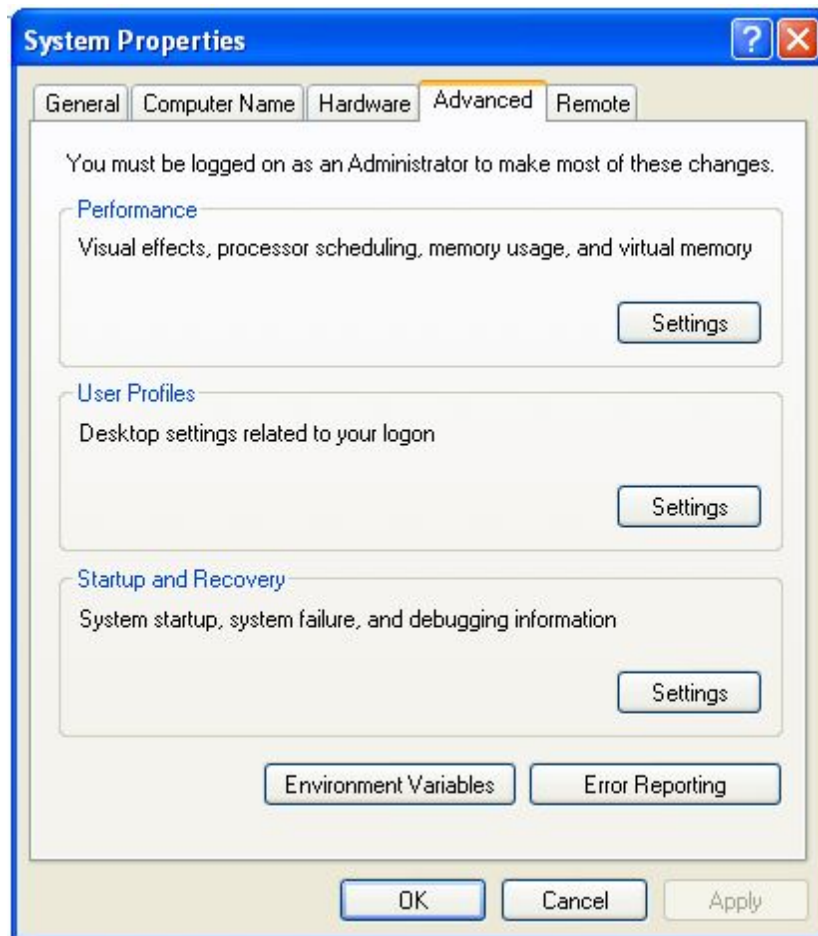
NOTA: si optamos por configurar las variables de entorno de esta forma, será necesario hacer esto mismo para ejecutar cada uno de los Scripts de Utilities-Project. Para ello, seleccionamos la utilidad que vayamos a configurar → Run as → Run configurations → doble clic sobre Java Application (en el lateral izquierdo) para añadir el archivo a la configuración y después seleccionar la pestaña Environment. Ahora se configuran las variables como se ha explicado anteriormente. Hay que repetir este proceso para cada uno de los scripts del proyecto de utilidades.

La otra posible forma de configurar las variables de entorno es desde el propio Sistema Operativo. Se mostrará como ejemplo los pasos a seguir en un SO Windows, aunque en los demás se hace de forma similar.

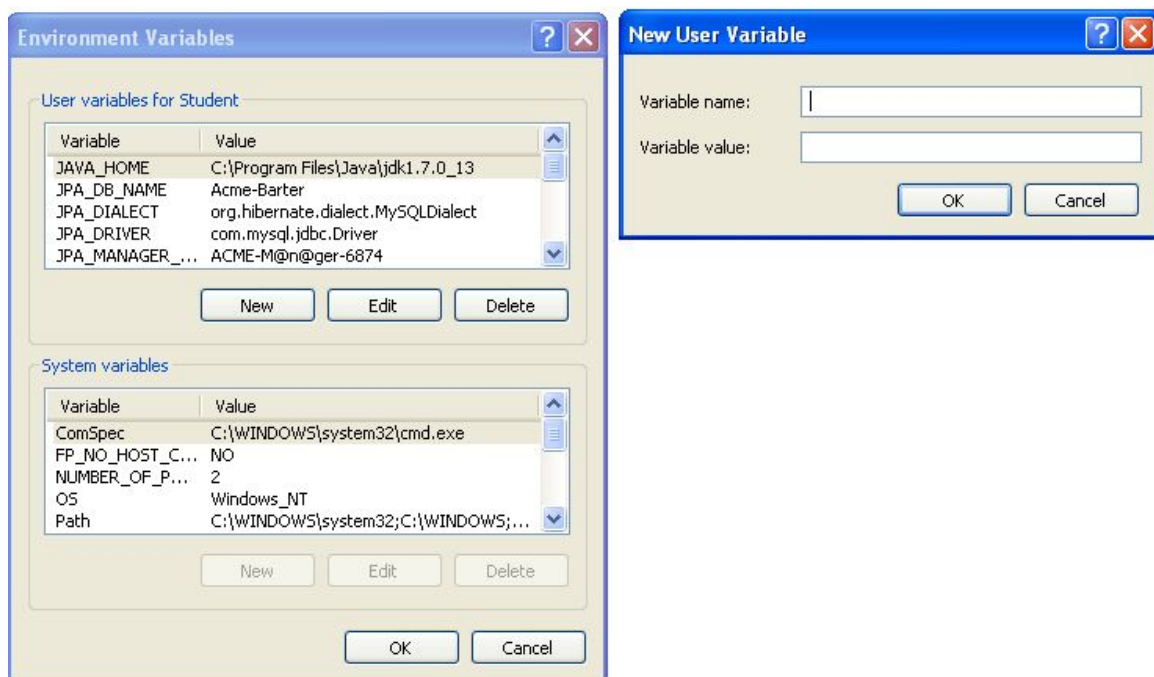
Se accede a Mi PC o Equipo y con el botón derecho se hace clic para seleccionar en el menú desplegable Propiedades.



Se selecciona el apartado de configuración avanzada y el botón de variables de entorno.



Ahí se deberán añadir cada una de estas variables que se han comentado anteriormente.



De esta forma se evita tener que configurar cada proyecto en cada Workspace y tener que añadir también esta configuración para cada una de las utilidades del Utilities Project.

Nuestra recomendación si usan este método es que definan aquí todas las variables que no van a variar y serán iguales para todos los proyectos y que en cada proyecto tan solo se tenga que definir cuál es el nombre de su base de datos, así como la URL.

2.2 Configurando el servidor de base de datos

Debemos crear una base de datos MySQL^[3] con el nombre que se crea oportuno, normalmente suele coincidir con el nombre del proyecto. También se conceden permisos a los usuarios `acme-user` y `acme-manager` de la siguiente manera:

```
drop database if exists `NombreBD`;  
create database `NombreBD`;
```

```
grant select, insert, update, delete  
on `NombreBD`.* to 'acme-user'@'%';
```

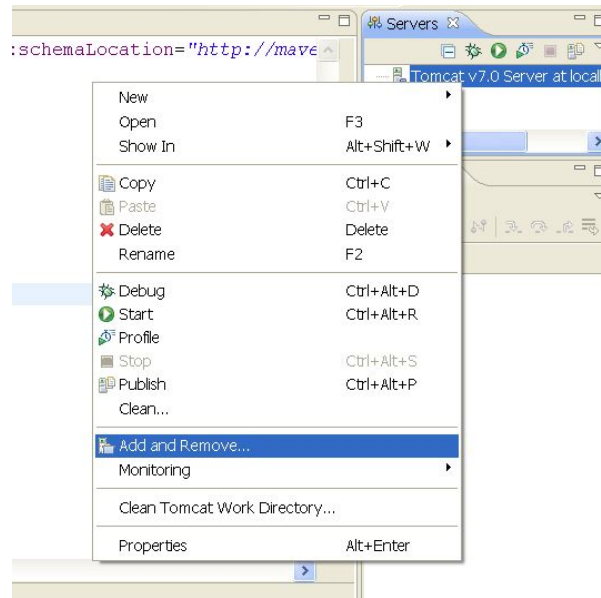
```
grant select, insert, update, delete, create, drop, references, index,  
alter,  
create temporary tables, lock tables, create view, create routine,  
alter routine, execute, trigger, show view  
on `NombreBD`.* to 'acme-manager'@'%';
```

En los ficheros que se proporcionan en el Workspace, en la carpeta Scrapbook están disponibles estas sentencias para ejecutarlas en la base de datos. Están especificadas en distintos ficheros `.sql`, en concreto en `create-database.sql` y `grant-privileges.sql`.

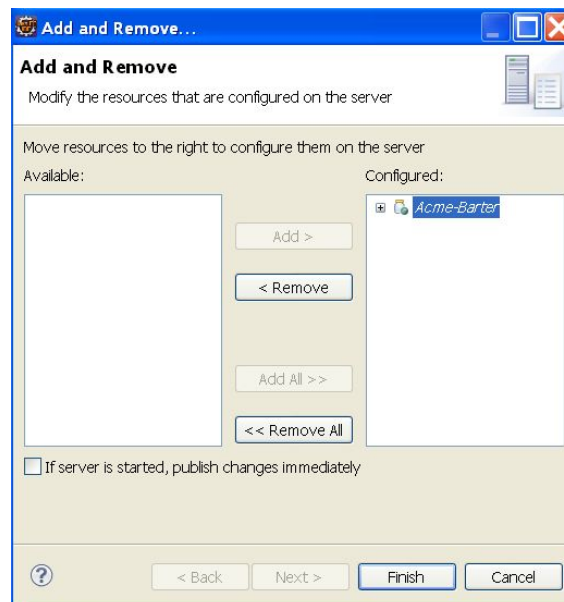
2.3 Ejecutando la plantilla

Antes de ejecutar el servidor, debemos poblar la base de datos. Para ello, basta con ejecutar el script `PopulateDatabase.java` del proyecto de utilidades. Una vez que haya poblado la base de datos, podremos ejecutar la plantilla. Tenga en cuenta que se deben definir las variables de entorno antes especificadas para ejecutar este Script, a menos que las haya creado en el sistema operativo.

En la zona de servers, pulsamos con el botón derecho sobre "Tomcat v7.0 Server at localhost" y seleccionamos "Add and Remove..."



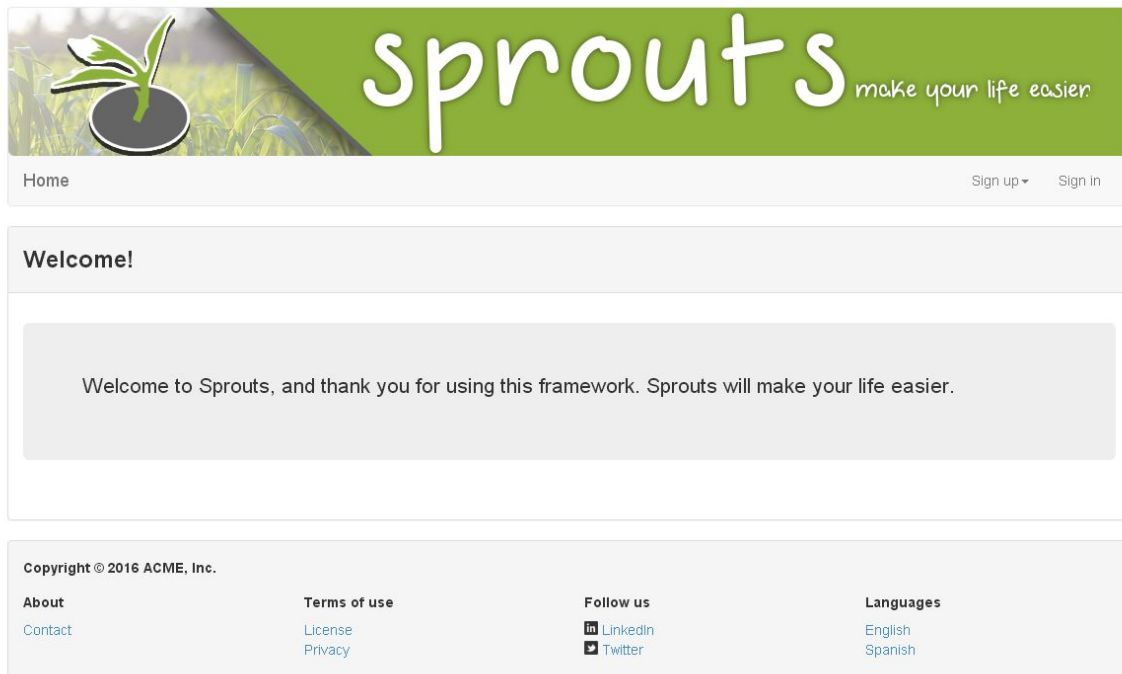
En la ventana que nos aparece, añadimos el recurso con el nombre que se le haya dado al proyecto en el fichero pom.xml al servidor, y pulsamos "Finish".



Ahora arrancamos el servidor pulsando el botón "Start the server":



Una vez arrancado el servidor, si todo ha ido bien, podremos ver nuestro proyecto ejecutándose en la siguiente URL: `localhost:8080/NombreDelProyecto`.



Sumario

Ya sabe cómo instanciar su entorno de trabajo, cómo importar y configurar la plantilla y el proyecto de utilidades, y cómo ejecutarlo todo. Tal y como ha podido comprobar, son pasos sencillos y no hay que configurar demasiadas cosas. Ahora ya está todo preparado para empezar a desarrollar su proyecto.

3. Modelo de dominio

Introducción

Una vez que tiene preparado su entorno de trabajo y la plantilla de Sprouts Framework, ya puede empezar a desarrollar su proyecto. El primer paso es crear el modelo de dominio Java. En este capítulo aprenderá a crear entidades y datatypes de una manera sencilla.

Creando una entidad de dominio o un datatype

Para crear una entidad de dominio del problema, debemos crear una clase que extienda a `DomainEntity`. Todas las entidades del dominio irán en el paquete `domain`. Esta clase añade los atributos `id`, `versión`, dos que representan la fecha de creación y de la última actualización de la entidad, y métodos de utilidad para controlar la concurrencia en la base de datos y reconstruir objetos.

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^[6] e Hibernate^[5], para una correcta configuración, validación de campos y generación de la base de datos..

Para la validación de elementos individuales que estén en una colección, se hace uso de etiquetas incorporadas por Sprouts Framework, incluyendo la librería validator-collection-2.1.6^[10]

Existe una etiqueta para colecciones por cada etiqueta de validación individual que provee Hibernate y Javax.

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

```
@ManyToMany
@NotNull
@EachNotNull
public Collection<Barter> getRequested() {
    return requested;
}
```

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.

NOTA: 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.

En caso de querer crear un Datatype, el procedimiento es exactamente igual que el mencionado anteriormente para una entidad, exceptuando el hecho de que la clase deberá implementar a Datatype si se quiere implementar el Datatype como un dato de tipo BLOB en la base de datos. Si se quiere mapear el Datatype haciendo que se embeban las columnas en la tabla de la base de datos de las entidades en las que aparezca, además se debe añadir la anotación @Embeddable. Para más información acerca de los Datatypes, acudir a la sección del manual correspondiente a *Alternativas en la implementación de Datatypes*.

Sumario

Ya ha sido capaz de crear las entidades y datatypes de su modelo de dominio, la base sobre la que se cimantan todos los demás componentes. Habrá comprobado que crear entidades de dominio es un proceso sencillo e intuitivo partiendo de un modelo de dominio UML.

4. Implementando casos de uso

Introducción

En este capítulo vamos a presentar los casos de uso más sencillos que puede implementar con Sprouts Framework. En primer lugar, le explicaremos cómo crear repositorios, servicios, controladores y vistas. Luego, explicaremos cómo crear casos de relacionados con operaciones CRUD. Las explicaciones irán en todo momento apoyadas en ejemplos basados en el proyecto Acme-Barter que ofrecemos junto a la plantilla.

4.1 Creación de repositorios

Lo primero de todo será crear un repositorio para cada una de las entidades implicadas en los casos de uso. Para ello, en el paquete *repositories* se deberá crear una interfaz que extienda de *PagingAndSortingRepository<Entity,ID>*, donde Entity será la clase de la entidad implicada y ID será el tipo de parámetro que representa al identificador de la entidad, generalmente será un *Integer* para todas las entidades. Además se debe añadir encima de la cabecera de la declaración de la interfaz la anotación *@Repository*.

```
@Repository("BarterRepository")
public interface BarterRepository extends PagingAndSortingRepository<Barter,
Integer>{
}
```

Ejemplo de la cabecera de un repositorio creado

4.2 Creación de servicios

Todos los servicios se crearán en el paquete *services*, deben tener las etiquetas *@Service* y *@Transactional* en la cabecera de la declaración de estos. Los servicios son clases que extienden, o bien a *AbstractService* o *AbstractFormService*, dependiendo de si se va a trabajar con una entidad o con un objeto de tipo Form. Más adelante aclararemos esto.

Por otra parte, los servicios pueden implementar varias interfaces, dependiendo de qué tipo de operaciones se vayan a realizar.

```
@Service
@Transactional
public class BarterService extends AbstractService<Barter, BarterRepository> {
}
```

Ejemplo de la cabecera de un servicio creado.

4.3 Creación del controlador

Se necesita crear un controlador que se encargue de recibir la petición del usuario de la aplicación y dirija la misma para que le devuelva una vista según la acción que ha solicitado.

Se crea una clase para el controlador en el paquete *controllers*, en el subpaquete correspondiente al rol que tiene permisos para solicitar dicha acción y a su vez en un subpaquete referente a la entidad con la que vamos a tratar.

El controlador debe llevar siempre las etiquetas `@Controller("nombreControlador")` y `@RequestMapping("entidad/rol")`. Esta última etiqueta sirve para que el controlador responda a la URI indicada seguida de un contexto (opcional) y la operación que realiza el controlador. Así pues, la estructura sería: "entidad/rol/{contexto separado por "," (opcional)}/acción.do". En el apartado de controladores de la documentación, se explica el uso del contexto.

Según las operaciones que se realicen en el controlador, extenderá de una u otra clase.

4.4 Creación de vistas (.jsp)

Todas las vistas deben incluir al comienzo del fichero .jsp la siguiente importación. Es un fichero jsp en el cual se realizan todas las importaciones de las librerías necesarias, para no tener que importarlás de nuevo.

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

La ruta donde deben ir alojados los ficheros .jsp de las vistas es en *resources/views/entityName*

En los ficheros .jsp, además de todas las etiquetas propias de Sprouts que se mencionan en este documento, también puede hacer uso de cualquiera de los componentes que proporciona Bootstrap^[11].

Debemos crear tres ficheros adicionales: *messages.properties*, *messages_es.properties*, y *tiles.xml*. Los dos primeros contienen los mensajes que se van a mostrar en la vista en inglés o en español, respectivamente. El tercero contiene los nombres de las vistas, relacionándolas con el fichero jsp correspondiente, gracias a la tecnología Apache Tiles^[7]. El formato de cada definición de vista dependerá del tipo de esta.

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

<tiles-definitions>
```

[Aquí van las definiciones de las vistas]
</tiles-definitions>

A los códigos de internacionalización que se utilizan en las distintas vistas se les da un valor en los `messages.properties` y `messages_es.properties`.

`match.create.title` = [Create a match](#)

4.5 Gestión de URIs: fichero `security-specific.xml`

Para que una URI sea accesible, debe estar declarada en el fichero `resources/spring/config/security-specific.xml`. Aquí se indica, además, el rol que debe tener el usuario que usa la aplicación para acceder a dicha URI. Este fichero se basa en el framework de autenticación y control de acceso de Spring Security^[8].

En primer lugar, debe haber una etiqueta indicando la primera parte de la URI, que será, por convenio, la entidad con la que estemos tratando:

```
<security:http pattern="/entidad/**" use-expressions="true" auto-config="true" >
[...]
```

Dentro de esa etiqueta, declaramos los roles que accederán a las URIs:

```
<security:intercept-url pattern="/rol/**"
    access="hasRole('Rol')" requires-channel="https"/>
```

Sirva como ejemplo la declaración de URIs de la entidad Barter:

```
<security:http pattern="/barter/**" use-expressions="true" auto-config="true" >
    <security:intercept-url pattern="/user/**"
        access="hasRole('User')" requires-channel="https"/>
    <security:intercept-url pattern="/administrator/**"
        access="hasRole('Administrator')"
requires-channel="https"/>
</security:http>
```

4.6 Casos de uso de listado de entidades

Repositorio

En caso de querer listar todas las entidades de este tipo que estén en la base de datos, no será necesario declarar ningún método en el repositorio. Sin embargo, si se quiere

listar determinadas entidades, sí será necesario crear un método que devuelva *Page<Entity>* y que reciba como uno de sus parámetros un *Pageable* para hacer uso de la paginación proporcionada por el repositorio. Encima de la declaración del método añadimos la consulta (con la anotación *@Query*) en JPQL con la que queremos extraer la información de la base de datos a mostrar en el listado.

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

Servicio

Se implementa un servicio como se indica al comienzo de esta sección que extienda de *AbstractService<Entity, EntityRepository>* y este servicio debe implementar *ListService<Entity>*.

Esto obliga a que en el servicio se tenga que definir el método *Page<Entity> findPage(Pageable page, String searchCriteria)*. Este método es el que utilizarán las tablas de listado por defecto cuando hagan una petición para cada una de las páginas de la misma.

En caso de que no hubiéramos creado ningún método en el repositorio, este *findPage* llamará al *findAll* pasándole el *Pageable* que recibe por parámetro.

```
@Override
public Page<Barter> findPage(int userId, Pageable page) {
    Page<Barter> result;

    result = repository.findBartersByUserId(userId, page);
    Assert.notNull(result);
    return result;
}
```

Controlador

En este caso, el controlador extiende de *AbstractListController<Entity, EntityService>*. Esto obliga a que se tenga que definir el método *view()*. En este hay que devolver el nombre de la vista en la cual se muestra la tabla. El nombre debe coincidir con el que se le ha dado a la misma en el fichero *tiles.xml* que se comentará a continuación.

```
@Controller("BarterListController")
@RequestMapping("home/barter")
public class ListController extends AbstractListController<Barter, BarterService>{

    @Override
    protected String view() {
```

```
        return "barter/list";
    }
}
```

Vista

Se debe crear un fichero .jsp en el cual se tendrá que incluir el componente *DataTables*^[9], el que se encarga del listado con paginación y la búsqueda de entidades. Para ello se añade la etiqueta `<sprouts:data-table>` y dentro de la misma se hará uso de `<sprouts:data-column>` para cada una de las columnas que se quieran mostrar y de `<sprouts:action-button>` para los botones de acciones que se podrán aplicar sobre una determinada fila de la tabla.

En `<sprouts:data-column>` los atributos más relevantes para mostrar una columna son *code* y *path*. El primero de ellos, hace referencia al código de internacionalización declarado en cada uno de los ficheros *.properties*, que se quiere mostrar en el título de dicha columna. El *path* hace referencia al atributo a mostrar en dicha columna.

Respecto a `<sprouts:action-button>`, los atributos a usar son: *url*, para definir la URI a la que se mandará al usuario al hacer clic sobre el botón (se puede hacer uso del parámetro {0} para inyectar en la URI el id de la entidad de cada fila); *code*, mismo uso que en `<sprouts:data-column>`.

```
<sprouts:data-table i18n="datatables.language" >
    <sprouts:data-column code="barter.username"
    path="user.userAccount.username" />
    <sprouts:data-column code="barter.moment" path="moment" sortable="true"
    format="date"/>
    <sprouts:data-column code="barter.title" path="title" />
    <sprouts:action-button url="home/user/profile/{0}/show.do"
    code="user.profile.viewDetails"/>
    <sprouts:action-button url="home/barter/{0}/show.do" code="show.button"/>
</sprouts:data-table>
```

Declaración de .jsp en tiles.xml

```
<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>
```

En el atributo *title* se define el título que llevará la vista a mostrar, definiendo el código que ahí se indique en los ficheros de internacionalización. En *body* se indica la ruta relativa al .jsp que se ha creado previamente.

List of barters

Show 5 items

Search:

View user's profile

Details

Creator	Date	Title	Requested item	Offered item
user1	05/05/16 18:56	Altavoces por objetos de marca	Objetos de marca	Altavoces LabTec
user1	04/26/16 15:00	Cambio chupa de cuero por camisa vaquera	Camisa vaquera	Chupa de cuero
user1	04/20/16 15:00	Ofrezco mi colección de sellos a cambio de televisor	Televisor FullHD	Colección de sellos
user1	03/31/16 15:00	Barter que NO debe ser cancelado2	Item barter 10	Item barter 10
user1	04/26/16 15:00	Busco entradas para Isla Mágica	Bonos descuento para restaurante	Entradas Isla Mágica

Showing items 1 - 5 (10 items in total)

First

Previous

1

2

Next

Last

Ejemplo de vista de listado.

Las tablas permiten formatear los valores de las celdas según el idioma del usuario. Puede obtener información al respecto en la documentación de Sprouts Framework.

4.7 Casos de uso para mostrar información respecto a una entidad

Como ejemplo, mostraremos información de un Barter, que será seleccionado en la lista usada como ejemplo en el apartado 3.6.

Repositorio

Para este caso de uso no es necesario declarar ningún método adicional, ya que la interfaz *PagingAndSortingRepository* provee por defecto de un método para encontrar entidades dado un identificador.

Creando el servicio

Debemos hacer que nuestro servicio implemente la interfaz *ShowService<Entity>*. Gracias a este paso, podremos hacer uso del método *findByld*, declarado en *ShowService* e implementado en *AbstractService*, que a partir de un identificador encuentra una entidad.

Siguiendo con el ejemplo de los Barters, *BarterService* implementará la interfaz *ShowService<Barter>*.

Creando el controlador

Debemos crear una clase, que llamaremos *DisplayController*, que extienda a la clase *AbstractShowController*, la cual recibe la entidad y servicio con los que va a trabajar. Al

extender esta clase, sucederán varias cosas. En primer lugar, esto hará que la URL a la que responderá este controlador (que indicamos en la etiqueta `@RequestMapping`) se le añadan parámetros, quedando estructurada de la siguiente forma:

`"home/entity/{identificador (opcional)}/show.do"`.

En segundo lugar, nos obligará a implementar los métodos `authorize` y `view`. El primero recibe como parámetros la entidad que va a tratar este controlador y el usuario que ha iniciado sesión. Sirve para comprobar que el usuario que accede a este controlador tiene permiso para ver la entidad a tratar. El segundo método no recibe nada, y devuelve el nombre de la vista que tratará el controlador.

En nuestro caso particular de los Barters, el método `authorize` devuelve true si es un administrador quien hace la petición o bien si el Barter no ha sido cancelado. Por otra parte, el método `view` devolverá `"barter/display"`.

```
@Controller("barterDisplayController")
@RequestMapping("home/barter")
public class DisplayController extends AbstractShowController<Barter, BarterService>
{
    @Override
    public boolean authorize(Barter domainObject, UserAccount principal){
        return SignInService.checkAuthority("Administrator") ||
        (domainObject.getCancelled() == false);
    }
    @Override
    protected String view() {
        return "barter/display";
    }
}
```

Creando la vista

Creamos un fichero jsp al que llamaremos `display.jsp`. Vamos a mostrar el título, el momento en el que se creó el Barter, un enlace al perfil del usuario que creó el Barter y fotos de los ítems que se piden y se ofrecen.

En primer lugar, declaramos las variables que contendrán el título de cada campo. El atributo `var` indica que nombre de la variable, y `code` el nombre que tiene el mensaje en los ficheros de internacionalización que creamos previamente.

```
<spring:message var="title" code="barter.title"/>
<spring:message var="moment" code="barter.moment"/>
<spring:message var="itemPictures" code="barter.item.pictures"/>
```


Con las siguientes etiquetas mostramos el contenido de los atributos del objeto con el que estamos tratando, contenido en la variable *modelObject*.

```
<sprouts:display-column title="${title}" data="${modelObject.title}"/>
<sprouts:display-column title="${moment}" path="modelObject.moment"/>
```

En el atributo *title* le pasamos el mensaje internacionalizado que hemos declarado, y en el atributo *data* le pasamos directamente el contenido de la propiedad que queramos mostrar. Si queremos que el dato aparezca formateado, como puede ser una fecha, utilizaremos el atributo *path*, pasándole una cadena indicándole la ruta a la propiedad que queremos formatear. En la documentación podrá encontrar más información acerca del formato de campos.

Para mostrar un botón, usamos la etiqueta *<sprouts:button />*, indicando la URL a la que nos enviará el botón y el mensaje que aparecerá en el mismo, representado por un código de internacionalización.

```
<sprouts:button url="home/user/${modelObject.user.id }/show.do"
code="user.profile"/>
```

El aspecto de lo que hemos hecho hasta ahora es el siguiente:

Title
Ofrezco mi colección de sellos a cambio de televisor

Date
04/20/16 15:00

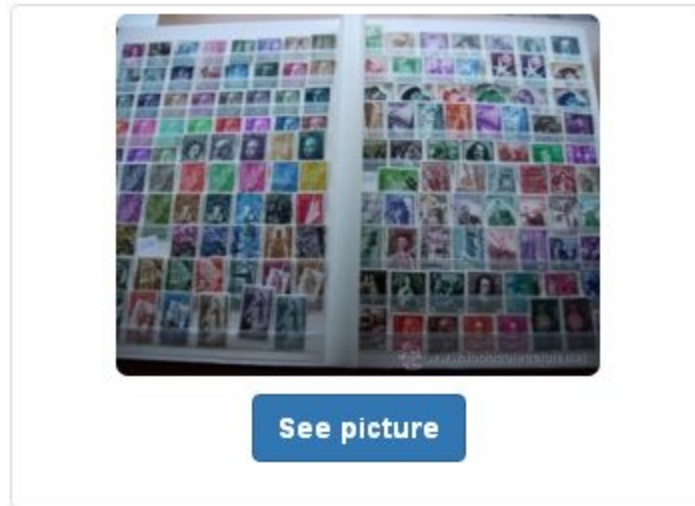
User's profile

Por último, para mostrar imágenes usaremos la etiqueta *<sprouts:pictures-list />*. Esta etiqueta recibe la colección de URIs de las imágenes por medio del atributo *pictures*, y el tamaño de dicha colección por medio del atributo *size*.

```
<sprouts:pictures-list pictures="${modelObject.requested.pictures }"
size="${modelObject.requested.pictures.size()}"/>
```

El aspecto es el siguiente:

Pictures



Declaración de .jsp en tiles.xml

Debemos añadir las siguientes líneas al fichero *tiles.xml* de la carpeta barter:

```
<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>
```

4.8 Casos de uso de creación de entidades

Repositorio

Dado que los repositorios extienden de *PagingAndSortingRepository*, este repositorio ya podrá realizar las operaciones CRUD.

Servicio

Para que el servicio esté habilitado para la creación, es necesario que implemente la interfaz *CreateService<Entity>*. Esto, nos obligará a implementar una serie de métodos:

- *getEntityClass()*: se indica la clase sobre la que se realizará la operación de crear.
- *createBusinessRules(List<BusinessRule<Entity>> rules, List<Validator> validators)*: se añade al parámetro *rules*, las reglas de negocio que previamente hayamos creado e instanciado en la clase mediante la anotación *@Autowired*. En la documentación oficial se especifica cómo se crean las reglas de negocio.

```
@Override
```

```
public void createBusinessRules(List<BusinessRule<Match>> rules, List<Validator>
validators) {
    rules.add(mustNotBeCancelled);
    rules.add(isUserMatch);
    rules.add(isNotMatched);
}
```

- *beforeCreating(Entity validable, List<String> context)*: se modifican los parámetros que sean necesarios del parámetro *validable* antes de enviarlo a la vista.

```
@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);
}
```

- *beforeCommittingCreate(Entity validable, List<String> context)*: se harán las modificaciones que sean necesarias al objeto que se recibe en la vista justamente antes de persistirlo en la base de datos.

```
@Override
public void beforeCommittingCreate(Match validable, List<String> context) {
    validable.setMoment(Moment.now());
    validable.setCancelled(false);
}
```

- *afterCommittingCreate(int id)*: cuando sea necesario realizar operaciones tras persistir el objeto en la base de datos porque afecte a otras entidades, se realizarán en este método, pudiendo hacer uso del identificador de la entidad antes persistida, que se pasa como parámetro.

Controlador

Para que un controlador realice las operaciones de creación, tiene que extender de *AbstractCreateController<Entity, EntityService>*. Opcionalmente, puede implementar la interfaz *AddsToModel* para añadir variables al modelo.

Tenemos que implementar dos métodos:

- *authorize(Match domainObject, UserAccount principal)*: mediante los objetos recibidos por parámetro, se debe realizar las comprobaciones que sean necesarias, para autorizar al usuario a realizar la operación o denegársela.

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

- *view()*: se debe devolver un String con el nombre de la vista de creación que se configuró en el *tiles.xml*

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

Si hemos implementado la interfaz *AddsToModel*, será necesario redefinir el método *addToModel(Map<String, Object> objects, List<String> context)*. Este método añade todas las variables que van a hacer falta en la vista. Para ello, hay que añadirlas al parámetro *objects*.

```
@Override
public void addToModel(Map<String, Object> objects, List<String> context) {
    Collection<Barter> offered;
    offered = barterService.findNotMatchedBartersByUserId();
    objects.put("offereds", offered);
}
```

Vista

Dentro de la carpeta en la ruta correspondiente a la entidad se crea un fichero *edit.jsp*. En el archivo *tiles.xml* configuraremos el nombre de la vista y el archivo que responderá a él. También especificaremos el título de la vista, si es de lectura solamente y el nombre del botón que enviará los datos, mediante códigos de internacionalización:

```
<definition name="match/create" extends="template/master">
```

```
<put-attribute name="title" type="string" value="match.create.title" />
<put-attribute name="body" value="match/create/body" />
</definition>

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

Ahora procederemos a crear el formulario de creación de la entidad en el archivo *edit.jsp*.

En primer lugar, siempre irán las importaciones de las librerías, y algunos de los atributos configurados en el *tiles.xml*.

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

<tiles:importAttribute name="readOnly" toName="readOnly" />
<tiles:importAttribute name="action" toName="action" />
```

Después creamos un formulario.

```
<sprouts:form modelAttribute="modelObject" readOnly="${readOnly}">
</sprouts:form>
```

A continuación especificamos los campos. Aquellos que el usuario no tenga que rellenar, se tienen que poner como campos ocultos, y los campos que se quieran proteger contra el hacking-post, se especificarán con la etiqueta *<sprouts:protected />*.

```
<form:hidden path="cancelled"/>
<form:hidden path="moment"/>

<sprouts:protected path="cancelled"/>
<sprouts:protected path="moment"/>
```

Después, se añaden los campos que sean necesarios rellenar, que pueden ser campos de texto, áreas de texto, desplegables...

```
<sprouts:select items="${offereds}" itemLabel="title"
  code="match.barter.offered" path="offered"/>
<sprouts:select items="${requesteds}" itemLabel="title"
  code="match.barter.requested" path="requested"/>
```

Por último, se le especifican los botones que estarán disponibles en ese formulario, siendo habitual, el botón de enviar y el de volver.

```
<sprouts:submit-button code="${action}" name="${action}"/>
<sprouts:cancel-button code="return.button" url="home/match/list.do" />
```

El aspecto del formulario de creación sería el siguiente:

Crear un emparejamiento

Trueque ofrecido

Trueque solicitado

Texto legal

Crear

Volver atrás

4.9 Casos de uso de actualización de entidades

Vamos a usar como ejemplo el caso de uso de actualización de una identidad social o `SocialIdentity`.

Repositorio

Basta con tener una interfaz que extienda a `PagingAndSortingRepository`. No es necesario definir ningún método.

Servicio

El servicio extenderá la clase `AbstractService<Entity, EntityRepository>` e implementará la interfaz `UpdateService<Entity>`. Al hacer esto, será necesario definir los métodos `beforeUpdating`, `beforeCommitingUpdate`, `updateBusinessRules` y `afterCommitingUpdate`. El funcionamiento de estos métodos se explicó en el apartado *Casos de uso de actualización de entidades*. La única diferencia es que estos métodos actúan al actualizar una entidad.

Para el caso de uso de actualizar una `SocialIdentity`, bastaría con dejar estos métodos vacíos, ya que no necesitamos realizar ninguna operación adicional:

```
@Override
public void beforeUpdating(SocialIdentity validable, List<String>
context) {
}

@Override
public void beforeCommitingUpdate(SocialIdentity validable, List<String> context) {
}
```

```
@Override
    public void updateBusinessRules(List<BusinessRule<SocialIdentity>> rules,
    List<Validator> validators) {
    }

@Override
    public void afterCommittingUpdate(int id) {
    }
```

Controlador

Nuestro controlador se llamará *UpdateController*. Como hemos hecho anteriormente, debemos especificar el nombre del controlador en la etiqueta *@Controller* y la ruta en la etiqueta *@RequestMapping*.

Los controladores relacionados con la actualización de entidades extienden la clase *AbstractUpdateController<Entity, EntityService>*. Esto hará que este controlador responda a la siguiente URI: *entity/rol/{contexto (opcional)}/update.do*.

Es necesario redefinir los métodos *authorize* y *view*. Ambos se explicaron en los casos de uso anteriores. Opcionalmente, se puede redefinir el método *onSuccess*, que devuelve una cadena que representa la URI a la que redireccionará la aplicación en caso de que la operación se complete con éxito. En caso de no redefinir este método, por defecto se hará una redirección a *entity/rol/list.do*.

El código para el controlador de actualización de *SocialIdentity* es el siguiente:

```
@Controller("updateUserSocialIdentityController")
@RequestMapping("socialIdentity/user")
public class UpdateController extends AbstractUpdateController<SocialIdentity,
SocialIdentityService>{
    @Override
    public boolean authorize(SocialIdentity domainObject, UserAccount principal)
    {
        return domainObject.getUser().getUserAccount().equals(principal);
    }
    @Override
    protected String view() {
        return "socialIdentity/update";
    }
    @Override
    protected String onSuccess() {
        return "/profile/user/show.do";
    }
}
```

Vista

Crearemos la vista en el fichero show.jsp de la carpeta socialIdentity. En el apartado Casos de uso de creación de entidades ya se explicaron las etiquetas básicas para la creación de formularios, por lo que simplemente mostraremos a modo de ejemplo parte del formulario de actualización de una SocialIdentity:

```
<tiles:importAttribute name="readOnly" toName="readOnly" />
<tiles:importAttribute name="action" toName="action" />
<sprouts:form modelAttribute="modelObject" readOnly="${readOnly}">
    <sprouts:hidden-field path="id"/>
    <sprouts:hidden-field path="version"/>
    <sprouts:hidden-field path="user"/>

    <sprouts:protected path="id"/>
    <sprouts:protected path="version"/>
    <sprouts:protected path="user"/>

    <sprouts:textbox-input code="socialIdentity.nick" path="nick" />
    <sprouts:textbox-input code="socialIdentity.network" path="socialNetwork" />

    <jstl:if test="${crudAction != 'showing'}">
        <sprouts:submit-button code="${action}" name="${action}" />
    </jstl:if>
    <sprouts:cancel-button code="return.button" url="profile/user/show.do" />
</sprouts:form>
```

En las dos primeras líneas se importan los atributos readOnly y action del fichero tiles.xml de socialIdentity. El botón de envío del formulario se mostrará sólo si la vista no se usa para mostrar la entidad. Esto se averigua gracias al atributo del modelo crudAction. El framework se encarga de dar valor a este atributo.

La definición de la vista en el tiles es la siguiente:

```
<definition name="socialIdentity/update" extends="template/master">
    <put-attribute name="title" type="string" value="socialIdentity.update.title" />
    <put-attribute name="body" value="socialIdentity/update/body" />
</definition>
<definition name="socialIdentity/update/body" template="../socialIdentity/show.jsp">
    <put-attribute name="readOnly" type="string" value="false" />
    <put-attribute name="action" type="string" value="update.button" />
</definition>
```


Como se puede apreciar, hay que especificar, por un lado, el título de la vista, y por otro lado el cuerpo de la vista. El formulario no será de solo lectura tal como se especifica en el atributo `readOnly`, y la acción será `update.button`.

Update social identity

Nick

@BeatrizJimenez

Social network

Twitter

Home page

https://twitter.com/beatrizjimenez

Picture

https://abs.twimg.com/sticky/default_profile_images/default_profile_0_200x200.png

Update

Return

Ejemplo de caso de uso de actualización

4.10 Casos de uso de eliminación de entidades

Repositorio

No es necesario añadir ningún método al repositorio de la entidad en cuestión ya creado. Al extender de *PagingAndSortingRepository* ya tiene métodos encargados de realizar las operaciones de borrado.

Servicio

Se crea un servicio como se especifica al inicio del apartado. Debe implementar *DeleteService<Entity>*. Como consecuencia de implementar dicha interfaz, se deben sobrescribir los siguientes métodos en caso de ser necesario.

```

@Override
public void beforeDeleting(SocialIdentity validable, List<String> context) {
}
@Override
public void beforeCommittingDelete(SocialIdentity validable, List<String> context)
{
}
@Override
public void deleteBusinessRules(List<BusinessRule<SocialIdentity>> rules,
List<Validator> validators) {
}
@Override
public void afterCommittingDelete(int id) {
}

```

Como se puede observar, los métodos son los mismos que en la creación y actualización de entidades. Su comportamiento es idéntico, solo que en este caso se refieren al borrado de una entidad.

Controlador

Es necesario crear un controlador en el paquete correspondiente que extienda de *AbstractDeleteController<Entity, EntityService>*. Una vez realizado, se deben implementar los siguientes métodos:

```
@Override
public boolean authorize(SocialIdentity domainObject, UserAccount principal)
{
    return domainObject.getUser().getUserAccount().equals(principal);
}
@Override
protected String view() {
    return "socialIdentity/delete";
}
@Override
protected String onSuccess() {
    return "/profile/user/show.do";
}
```

Como se puede observar, son los mismos que en el caso de actualización o creación de una entidad y el funcionamiento es por tanto idéntico.

Vista

El fichero .jsp para la eliminación de una entidad será por lo general el mismo que se utiliza para la creación y edición de la misma. La única diferencia es que los campos aparecerán como *readOnly*, no se podrán editar.

Respecto al fichero de configuración tiles.xml, en el contenido del mismo se deberá indicar el título y el cuerpo del mismo, añadiendo a este que los campos son *readOnly* y que el botón de acción se va a encargar de borrar.

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

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

Delete social identity

Nick

@BeatrizJimenez

Social network

Twitter

Home page

https://twitter.com/beatrizjimenez

Picture

https://abs.twimg.com/sticky/default_profile_images/default_profile_0_200x200.png

Delete

Return

Ejemplo de caso de uso de eliminación

4.11 Casos de uso de mostrar información simple de entidades

Repositorio

Tan solo es necesario crear el repositorio asociado a la entidad, sin añadir ningún método adicional.

Servicio

En el servicio que se encarga de gestionar esta entidad, es necesario implementar la interfaz *ShowService<Entity>*.

Controlador

Se creará el controlador como se mencionó en el apartado Controladores. Este controlador tiene que extender de *AbstractShowController<Entity, EntityService>*, y esto nos obligará a implementar dos métodos:

- *authorize(Entity domainObject, UserAccount principal)*: se realizarán las operaciones pertinentes en función de los parámetros recibidos para comprobar que puede realizar la operación.
- *view()*: debe devolver el nombre de la vista que se configuró en el *tiles.xml*.

Vista

Dentro de la carpeta en la ruta correspondiente a la entidad se crea un fichero *show.jsp*. En el archivo *tiles.xml* configuraremos el nombre de la vista y el archivo que responderá a él. También especificaremos el título de la vista, especificaremos que solo es de lectura al tratarse de mostrar información simple y el nombre del botón.

```
<definition name="socialIdentity/show" extends="template/master">
```

```
<put-attribute name="title" type="string"
               value="socialIdentity.show.title" />
<put-attribute name="body" value="socialIdentity/show/body" />
</definition>


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

Ahora pasaremos a darle cuerpo al fichero show.jsp

Dado que se trata de un formulario deshabilitado para la modificación de los parámetros, la manera de crear esta vista, es análoga a la que se explicó en el apartado de *Casos de uso de eliminación de entidades, Vista*.

La vista quedaría de la siguiente forma:

Identidad social



<https://twitter.com/BillGates>

Nick

Social network

Sumario

Ahora ya sabe cómo realizar los casos de uso más comunes y ha podido apreciar todo el potencial de este framework. Habrá comprobado lo fácil y rápido que es desarrollar casos de uso con él y cómo esto le va a hacer ahorrar costes en desarrollo. En la documentación oficial encontrará todo lo necesario para aprender a implementar casos de uso más complejos.

5. Conclusión

Como habrá podido comprobar, haciendo uso de esta plantilla proporcionada por Sprouts-Framework la complejidad de gran parte de los casos de uso típicos de un proyecto se ve reducida de forma considerable. No solo eso, sino que además el tiempo necesario para realizarlos es mucho menor que si se intentase implementar la misma funcionalidad con el framework Spring directamente. Asimismo, la probabilidad de aparición de bugs y errores se ve reducida considerablemente.

Es por ello que el uso de Sprouts-Framework 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.

6. Bibliografía

- [1]. (2016). Spring Framework Reference Documentation. [online] Available at: <http://docs.spring.io/spring/docs/3.2.17.RELEASE/spring-framework-reference/htmlsingle/>
- [2]. Siveton, J. (2016). Maven – Maven Getting Started Guide. [online] Maven.apache.org. Available at: <https://maven.apache.org/guides/getting-started/index.html>
- [3]. (2016). MySQL :: MySQL 5.5 Reference Manual. [online] Available at: <http://dev.mysql.com/doc/refman/5.5/en/>
- [4]. McClanahan, C., Maucherat, R. and Shapira, Y. (2016). Apache Tomcat 7 (7.0.69) - Documentation Index. [online] Tomcat.apache.org. Available at: <https://tomcat.apache.org/tomcat-7.0-doc/index.html>
- [5]. Ferentschik, H. and Morling, G. (2016). Hibernate Validator. [online] Docs.jboss.org. Available at: http://docs.jboss.org/hibernate/validator/4.3/reference/en-US/html_single/
- [6]. (2016). javax.validation.constraints (Java(TM) EE 7 Specification APIs). [online] Available at: <https://docs.oracle.com/javaee/7/api/javax/validation/constraints/package-summary.html>
- [7]. (2016). *Apache Tiles - Framework - Getting started*. [online] Available at: https://tiles.apache.org/framework/getting_started.html

- [8]. (2016). *Spring Security Reference*. [online] Available at:
<http://docs.spring.io/spring-security/site/docs/3.2.9.RELEASE/reference/htmlsingle/>
- [9]. Datatables.net. (2016). *Manual*. [online] Available at:
<https://datatables.net/manual/index>
- [10]. GitHub. (2016). *jirutka/validator-collection*. [online] Available at:
<https://github.com/jirutka/validator-collection>
- [11]. Mark Otto, a. (2016). Getting started · Bootstrap. [online] Getbootstrap.com.
Available at: <http://getbootstrap.com/getting-started/>
- [12]. Summer Framework
<https://summer.meteor.com/>
- [13]. Directorio de la asignatura Diseño y Pruebas del Grado en Ingeniería Informática - Ingeniería del Software de la Universidad de Sevilla.
<ftp://postgrado.lsi.us.es/DT/>
- [14]. Oracle.com. (2016). *Java SE Development Kit 7 - Downloads | Oracle Technology Network | Oracle*. [online] Available at:
<http://www.oracle.com/technetwork/es/java/javase/downloads/jdk7-downloads-1880260.html>
- [15]. Eclipse.org. (2016). *Eclipse IDE for Java EE Developers | Packages*. [online]
Available at:
<http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/mars2>