Eusko Jaurlaritzaren Informatika Elkartea
Sociedad Informática del Gobierno Vasco

UDA - Utilidades de desarrollo de aplicaciones

Gestión de properties

Fecha: 02/02/2012 | **Referencia:**

EJIE S.A.
Mediterráneo, 14
Tel. 945 01 73 00*
Fax. 945 01 73 01
01010 Vitoria-Gasteiz
Posta-kutxatila / Apartado: 809
01080 Vitoria-Gasteiz
www.ejie.es

Control de documentación

Título de documento: Gestión de properties

Histórico de versiones

Código:	Versión:	Fecha:	Resumen de cambios:
	1.0.0	02/02/2012	Primera versión.

Cambios producidos desde la última versión

Control de difusión

Responsable: Ander Martínez

Aprobado por:

Firma: Fecha:

Distribución:

Referencias de archivo

Autor:

Nombre archivo:

Localización:

Contenido

1 Introducción

El presente documento pretende explicar y detallar la manera en la que se gestionan los ficheros de propiedades (tanto de aplicación como idiomáticos) en UDA a través de Spring MVC. De este modo, los usuarios podrán comprender su funcionamiento y acceder al contenido de dichos ficheros de manera rápida y sencilla sin problema alguno.

El acceso a los ficheros que nos ocupan se realiza a través de ciertos objetos concretos como se verá a continuación. Por ello se incluyen diversos apartados relacionados con la declaración y uso de dichos objetos de manera programática y declarativa.

2 Gestión mediante anotaciones (modo programático)

2.1 Declaración

Spring y por tanto UDA, permiten el uso de anotaciones para la declaración de objetos en un "repositorio de *beans*" de Spring. Cuando se desee utilizar alguno de estos objetos bastará con recuperarlo e invocar la función correspondiente a través de la Inyección de Dependencia (ID).

Existen ciertas anotaciones que permiten que Spring incluya ciertas clases Java directamente como objetos en el respositorio y estas son algunas de ellas:

```
@Controller
@Service
@Repository
@Component
```

El plugin de generación de código de UDA permite seleccionar el uso de anotaciones para la resolución de las dependencias. En caso de seleccionar esta opción, en cada controlador, servicio o DAO se incluirá su correspondiente anotación (para los DAO se utiliza la anotación @Repository).

Una vez las clases estén implementadas (con su anotación correspondiente), Spring tiene que cargarlos en su repositorio para poder ser utilizados a través de la ID. El fichero de configuración incluirá un tag que le indica el paquete desde el que tiene que recorrer los subpaquetes localizando las clases Java para incluir como objetos del repositorio aquellas anotadas correctamente.

- Controllers (WAR)
  - xxxNombreWar\WebContent\WEB-INF\spring\mvc-config.xml
  - `<context:component-scan base-package="com.ejie.x21a.control" />`
- Servicios (EAR)
  - xxxEARClasses\src\spring\service-config.xml
  - `<context:component-scan base-package="com.ejie.x21a.service" />`
- DAOs (EAR)
  - xxxEARClasses\src\spring\dao-config.xml
  - `<context:component-scan base-package="com.ejie.x21a.dao" />`

Adicionalmente, Spring comprobará las anotaciones internas de cada clase resolviendo las dependencias que encuentre como se explica en el siguiente apartado.

2.2 Inyección

La inyección de dependencia de los objetos de manera programática se realiza del siguiente modo:

```
@Autowired
private Properties appConfiguration;
```

Mediante esta anotación Spring buscará en su repositorio de *beans* algún objeto cuyo ID sea appConfiguration y enlazará la variable declarada con dicho objeto. Se podrá utilizar directamente el objeto appConfiguration sin tener que inicializarlo ya que Spring lo habrá realizado por nosotros.

3 Gestión mediante XML (modo declarativo)

3.1 Declaración

La declaración de objetos en Spring puede realizarse a través del fichero XML ya que hasta la versión 5 de Java no existía soporte para anotaciones.

Por cada objeto que se quiera declarar en el repositorio de *beans* de Spring deberá incluir una sentencia como la siguiente:

```
<bean id="NOMBRE_DEL_OBJETO" class="RUTA_DEL_OBJETO">
      <property name="xxa" ref="yyy" />
      <property name="xxb" value="zzz" />
      …
</bean>
```

- NOMBRE_DEL_OBJETO: Identificador con el que se va a registrar el objeto (se utilizará para reverenciarlo o recuperarlo)
- RUTA_DEL_OBJETO: Nombre completo de la clase que implementa el objeto (incluyendo paquete)
- xxa: Nombre de las propiedad del objeto donde se va a inyectar el objeto yyy.
- yyy: Objeto del repositorio de beans de Spring a inyectar en el objeto que se está declarando.
- xxb: Nombre de la propiedad a la que se le va a asignar el valor zzz.
- zzz: Valor que se asociadrá directamente a la propiedad xxx2

UDA declara los objetos empleados para la gestión de propiedades mediante XML como se verá más adelante. **A pesar de que los objetos se declaren en el XML, puede utilizarse la anotación @Autowired para inyectarlos allí donde sean necesarios**.

3.2 Inyección

La inyección de dependencia de los objetos de manera programática se realiza a través del método set de la variable.

```
private Properties appConfiguration;
public void setAppConfiguration(Properties appConfiguration) {
      this.appConfiguration = appConfiguration;

}
```

Cuando Spring esté registrando los objetos en su repositorio y detecte que un objeto define una propiedad que es una referencia a otro objeto, intentetará invocar el metodo set de esa propiedad.

En el siguiente ejemplo, Spring al cargar el objeto “example” implementado en el fichero “com.ejie.xxx.Foo” invocará el método setAppConfiguration para inyectarle el objeto. Se podrá utilizar directamente el objeto appConfiguration sin tener que inicializarlo ya que Sprin lo habrá realizado por nosotros.

```
<bean id="example" class="com.ejie.xxx.Foo">
      <property name="appConfiguration" ref="appConfiguration" />

</bean>
```

4 Propiedades de aplicación

Las aplicaciones desarrolladas con UDA contienen un fichero de propiedades de aplicación denominado **aaa.properties** (ej. x21a.properties) alojado en /config/aaa. Dicho fichero contiene valores que dependen del entorno en el que se trabaja pero no de la versión idomática. Un claro ejemplo puede ser la ruta en la que se alojan el contenido estático de la aplicación:

```
foo=bar
```

El acceso a este tipo de propiedades se realizará a través de la variable **appConfiguration** declarada en el fichero xxxEARClasses\src\spring\service-config.xml:

```
<bean id="appConfiguration"
class="org.springframework.beans.factory.config.PropertiesFactoryBean">
  <property name="location" value="classpath:x21a/x21a.properties" />
  <property name="fileEncoding" value="UTF-8" />
</bean>
```

- Acceso desde clase **Java**
  - Declarar la propiedad (consultar apartados 2.1 y 3.1)

    ```
    private Properties appConfiguration;
    ```

  - Acceso directo a la propiedad a recuperar

    ```
    appConfiguration.get("foo")
    ```

- Acceso desde **JSP**
  - Guardar propiedad en el Model (en el controller)

    ```
    model.addAttribute("foo", "bar");
    ```

  - Acceso medienta EL (Expression Language)

    ```
    ${foo}
    ```

5 Propiedades idiomáticas (i18n)

La gestión idiomática (i18n) se realiza a tarvés de diferentes ficheros de propiedades (uno por cada idioma de la aplicación). Un claro ejemplo puede ser el literal de la pantalla de inicio:

```
welcome.title=Hello EJIE!
```

Estos ficheros existen a varios niveles (EAR y WAR) y por tanto en diferentes localizaciones como veremos a continuación.

5.1 Propiedades del EAR

Los ficheros de propiedades del EAR se encuentran en la siguiente ruta:

- xxxEARClasses\resources
  - xxx.i18n_en.properties
  - xxx.i18n_es.properties
  - xxx.i18n_eu.properties
  - …

El acceso a las propiedades del EAR se realizará a través de la variable **appMessageSource** declarada en el fichero xxxEARClasses\src\spring\service-config.xml:

```
<bean id="appMessageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
  <property name="basename" value="x21a.i18n" />
  <property name="defaultEncoding" value="UTF-8" />
  <property name="useCodeAsDefaultMessage" value="true" />
  <property name="fallbackToSystemLocale" value="false" />
</bean>
```

- Acceso desde clase **Java**
  - Declarar la propiedad (consultar apartados 2.1 y 3.1)

```
private ReloadableResourceBundleMessageSource appMessageSource;
```

  - Acceso directo a la propiedad a recuperar

```
appMessageSource.getMessage("welcome.title", args, locale)
```

    - args: Array de argumentos de la propiedad (ej. {0}, {1}, … {n}). En caso de no tener se pasará null
    - La *locale* se puede obtener como variable directamente o como parámetro de un método:
      - `Locale locale = LocaleContextHolder.getLocale();`
      - `public void xxx (Locale locale) { ... }`

- Acceso desde **JSP**
  - Incluir el fichero TLD de Spring (se incluye en el fichero 'includeTemplate.inc')

```
<%@ taglib prefix="spring" uri="/WEB-INF/tld/spring.tld" %>
```

- Acceso a la propiedad mediante el tag de Spring

```
<spring:message code="welcome.title" />
```

UDA incluye automáticamente la TLD de Spring en el fichero 'includeTemplate.inc' en la ruta xxxNombreWar\WebContent\WEB-INF\ por lo que bastará con incluir dicho fichero:

```
<%@include file="/WEB-INF/includeTemplate.inc"%>
```

5.2 Propiedades del WAR

Los ficheros de propiedades del WAR se encuentran en la siguiente ruta:

- xxxNombreWar\WebContent\WEB-INF\resources
  - xxxNombreWar.i18n_en.properties
  - xxxNombreWar.i18n_es.properties
  - xxxNombreWar.i18n_eu.properties
  - …

Las propiedades definidas a nivel de WAR solo pueden ser accedidas desde el propio WAR, en cambio las propiedades definidas a nivel de EAR pueden ser accedidas desde cualquier capa (Presentación, Servicios, Acceso a datos, …).

La resolución de una propiedad desde el WAR sigue el siguiente orden. Primero se comprueba que la propiedad exista en sus ficheros de recursos y en caso afirmativo se recuperará. Si por el contrario no existe se buscará en los ficheros de recursos del EAR. De esta manera en caso de necesitar identicos literales a nivel de EAR y de los diferentes WAR bastará con definirlos únicamente en el fichero de recursos del EAR.

El acceso a las propiedades del WAR se realizará a través de la variable **messageSource** declarada en el fichero xxxNombreWar\WebContent\WEB-INF\spring\mvc-config.xml:

```
<bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
        <property name="parentMessageSource" ref="appMessageSource" />
        <property name="basename" value="/WEB-INF/resources/x21aPilotoPatrones.i18n" />
        <property name="defaultEncoding" value="UTF-8" />
        <property name="useCodeAsDefaultMessage" value="true" />
        <property name="fallbackToSystemLocale" value="false" />
</bean>
```

- Acceso desde clase **Java**
  - Declarar la propiedad (consultar apartados 2.1 y 3.1)

```
private ReloadableResourceBundleMessageSource messageSource;
```

  - Acceso directo a la propiedad a recuperar

```
messageSource.getMessage("welcome.title", args, locale)
```

    - args: Array de argumentos de la propiedad (ej. {0}, {1}, … {n}). En caso de no tener se pasará null
    - La *locale* se puede obtener como variable directamente o como parámetro de un método:

- `Locale locale = LocaleContextHolder.getLocale();`
- `public void xxx (Locale locale) { ... }`

- Acceso desde **JSP**
  - Incluir el fichero TLD de Spring (se incluye en el fichero 'includeTemplate.inc')

```
<%@ taglib prefix="spring" uri="/WEB-INF/tld/spring.tld" %>
```

  - Acceso a la propiedad mediante el tag de Spring

```
<spring:message code="welcome.title" />
```

UDA incluye automáticamente la TLD de Spring en el fichero 'includeTemplate.inc' en la ruta xxxNombreWar\WebContent\WEB-INF\ por lo que bastará con incluir dicho fichero:

```
<%@include file="/WEB-INF/includeTemplate.inc"%>
```

6 Uso de propiedades en clases de utilidades

El acceso y explotación de propiedades desde las clases de las capas de presentación (controllers), servicio (services) o acceso a datos (daos) es directo ya que UDA deja todo configurado directamente. Y dependiendo de cómo se generara el código a través del plugin de UDA se deberá realizar lo siguiente:

- Anotaciones
  - Incluir la anotación @Autowired delante del objeto
  - Asegurarse de que el paquete en el que se encuentra la clase está en el scan (por defecto sí)
- XML
  - Incluir el método set de la aplicación
  - Configurar la propiedad del objeto en el fichero XML correspondiente (tag property)

El problema puede existir cuando se utilizan clases propias de ayuda (paquetes de utilidades) que al no configurarse correctamente pueden inducir a error y generar problemas.

A continuación se muestra un ejemplo detallando esta casuística. Se creará una clase de utilidades cuya función recupere un valor de una propiedad de aplicación (para i18n sería idéntico) y lo escriba en el log:

```
package com.ejie.xxx;

//imports

public class Foo {

	private static final Logger logger = ...;

	//Declaración de appConfiguration (ver apartado 6.1)

	public void do(){
		logger.info(appConfiguration.get("bar"));
	}


}
```

6.1 Declaración del objeto a consumir

A continuación se detalla como debería declararse la variable appConfiguration dependiendo del tipo de variable (de instancia o de clase que dependerá del tipo de método a implementar) y de si se usan anotaciones o XML:

- Anotaciones
  - Variable de instancia

```
@Autowired
private Properties appConfiguration;
```

  - Variable de clase

```
private static Properties appConfiguration;

@Autowired
public void setAppConfiguration(Properties appConfiguration){
   Foo.appConfiguration = appConfiguration;
```

```
}
```

- XML
  - Variable de instancia

```
private Properties appConfiguration;
public void setAppConfiguration(Properties appConfiguration) {
      this.appConfiguration = appConfiguration;
}
```

  - Variable de clase

```
private static Properties appConfiguration;
public void setAppConfiguration(Properties appConfiguration){
      Foo.appConfiguration = appConfiguration;

}
```

NOTA: En caso de que la variable se declare de clase, habrá que modificar el método para añadirle el modificador **static.**

6.2 Configuración de la clase para la inyección de dependencia

En el caso de que utilicen anotaciones como la clase se encuentra fuera de los paquetes generados y escaneados por UDA (control, service y dao) se deberán realizar dos modificaciones:

1. Se deberá anotar la clase con el modificador @Component

```
@Component
public class Foo ...
```

2. Se deberá incluir el escaneo del paquete en el fichero xml correspondiente (dependiendo de su localización en el mvc-config [WAR], service-config [EAR] o dao-config [EAR]). Por ejemplo si lo incluyeramos en el mvc-config (ya que la clase la desplegaremos en el WAR):

```
<context:component-scan base-package="
      com.ejie.x21a.control,
      com.ejie.xxx"
/>
```

Si por el contrario se utiliza XML para la configuración de objetos, se deberá incluir la definición del bean en el xml para que Spring inyecte la dependencia a través del setter:

```
<bean id="foo" class="com.ejie.xxx.Foo">
      <property name="appConfiguration" ref="appConfiguration" />
</bean>
```

6.3 Consumo de la clase de utilidades

Una vez declarada la clase de utilidades con su objeto de acceso a las propiedades, será turno de realizar la invocación, por ejemplo desde un controlador.

En el caso de que el método de utilidades sea **estático** bastará con invocarlo de la siguiente manera:

```
@RequestMapping(method = RequestMethod.GET)
public String getXXX(Model model) {
	Foo.do();
}
```

NOTA: Podría inyectarse el objeto como variable del controller e invocar el método sobre esa instancia del objeto pero daría un warning por la manera de invocar el método por lo que se descarta.

En el caso de que el método de utilidades sea **dinámico**, podría invocarse de alguna de las siguientes maneras:

1. Inyectando el objeto (mediante anotación o xml) e invocando el método directamente

   - XML

     ```
     private Foo foo;
     public void setFoo (Foo foo) {
     	this.foo = foo;
     }

     @RequestMapping(method = RequestMethod.GET)
     public String getXXX(Model model) {
     	foo.do();
     }
     ```

     Habría que declarar la inyección de la variable en el fichero XML

     ```
     <bean id="fooController"
     	class="com.ejie.xxx.control.FooController">
     	<property name="foo" ref="foo" />
     </bean>
     ```

   - Anotación

     ```
     @Autowired
     private Foo foo;

     @RequestMapping(method = RequestMethod.GET)
     public String getXXX(Model model) {
     	foo.do();
     }
     ```

2. Creando una instancia de la clase de utilidades e invocando el método

   ```
   @RequestMapping(method = RequestMethod.GET)
   public String getXXX(Model model) {
   	Foo foo = new Foo();
   	foo.do();
   }
   ```

En este segundo caso (al invocar el constructor directamente) el objeto foo inicializará su variable interna *appConfig* a **null** por lo que en la invocación del método se lanzará una excepción (NullPointerException). Para evitar esto bastará con modificar la clase de utilidades para que extienda de "SpringBeanAutowiringSupport" del siguiente modo:

```
public class Foo extends SpringBeanAutowiringSupport {
      ...
}
```

6.4 Corolario

Para evitar inconsistencias y problemas de esta índole, se recomienda el uso de **anotaciones** para la inyección de dependencia y **evitar el uso de bloques estáticos (static)** en la medida de lo posible. Esto se justifica porque Spring instancia sus *beans* siguiendo el patrón *singleton* por lo que internamente se crean una única vez.

SpringSource.org también es partidario de seguir estas pautas en el desarrollo de aplicaciones porque este tipo de bloques pueden generar problemas al integrarse con Spring.

7 Conclusiones

La explotación de los ficheros de properties en UDA se realiza a través de objetos es idéntica a la explotación de cualquier objeto en Spring.

Una vez que se configuran los objetos encargados de su gestión (appConfiguration, appMessageSource y messageSource), su explotación como se ha visto en este documento es fácil y sencilla. Independientemente del modo en el que se trabaje (anotaciones o a través de xml) el acceso a las propiedades es idéntica y no presenta grandes problemas.

El único problema o complejidad puede presentarse cuando se utilizan clases de utilidades (clases que no entran en las capas de presentación, servicios y acceso a datos) que habrá que configurar manualmente (al igual que habría que hacer con el resto de clases si no utilizamos el plugin de UDA).