La capa de negocio en Spring: el contenedor de beans

Índice

Spring es un *framework* que se ha convertido en el máximo exponente de toda una filosofía "alternativa" a la visión oficial de Sun sobre el desarrollo de aplicaciones *enterprise*. En lugar de la "artillería pesada" empleada tradicionalmente en aplicaciones JavaEE (servidores de aplicaciones + EJBs), los defensores de Spring propugnan el uso de "contenedores ligeros" y objetos Java convencionales (POJOs o Plain Old Java Objects).

Veremos en esta sesión una introducción al *framework* y a una parte fundamental de su núcleo básico: el contenedor que gestiona el ciclo de vida de los objetos que forman nuestra capa de negocio. Este contenedor ejerce unas funciones muy similares a las de un servidor de aplicaciones, pero como veremos, la diferencia más que en "qué" hace reside en "cómo" lo hace.

1. El framework Spring

Spring es un framework que se propone como alternativa al uso de EJBs y servidores de aplicaciones para el desarrollo de aplicaciones J2EE. Intenta basarse en una serie de "buenos principios" de desarrollo, como la inversión de control (Inversion of Control), programación orientada a aspectos (AOP) y otros, que permitan superar la excesiva complejidad a la que se han enfrentado tradicionalmente los desarrolladores J2EE.

El framework está organizado en 7 módulos diferentes, y diseñado de forma que se pueden usar todos los módulos o solo los que se necesiten:

- **El núcleo (core):** contiene las clases básicas, y el contenedor de beans, que veremos en este tema
- **El módulo AOP** proporciona soporte a la mayor parte de servicios que ofrece Spring a los objetos de negocio, similares a los que da un contenedor EJB: transaccionalidad, seguridad,... Será el objeto del tema 2.
- **El módulo ORM** se usa para proporcionar un interfaz uniforme a diversas herramientas de persistencia ORM. En los temas 3 y 4 veremos una de estas herramientas: Hibernate.
- **El módulo DAO** se usa en aquellas aplicaciones en la que nos baste con JDBC para implementar la persistencia. Proporciona una capa de abstracción que simplifica el trabajo con JDBC.
- **El módulo Web MVC** es un framework MVC al estilo de Struts, aunque integrado con el contenedor de beans propio de Spring.
- **El módulo Web** representa el interfaz entre la capa de negocio y la de presentación en aplicaciones web. Puede usarse solo, con un framework MVC de terceros (p.ej. Struts) o bien con el MVC propio de Spring.
- **El módulo Context** proporciona una forma de acceder a los beans de modo similar a como lo hace JNDI, además de dar soporte a tareas de comunicación diversas como mail, acceso a EJBs, etc.

La versión actual de Spring en el momento de escribir estas páginas es la 2, que incorpora bastantes mejoras en cuanto a potencia y ofrece mecanismos de configuración más sencillos que la versión inmediatamente anterior, la 1.2. El framework dispone de una documentación excelente, que hemos tomado como referencia básica para estos apuntes.

2. Gestionando los objetos de negocio: Inversion of Control y Dependency injection

Un aspecto vital en cualquier *framework* es cómo se gestiona el ciclo de vida de los objetos de negocio. En la mayor parte de casos (EJBs incluidos) en la actualidad se utiliza un paradigma llamado **Inversion of Control (IoC)**, que consiste en que los objetos no controlan su propio ciclo de vida, sino que lo hace el contenedor. El objeto puede implementar métodos de *callback* a los que el contenedor llamará cuando se produzca un determinado evento.

Una cuestión adicional es cómo localiza un objeto a los otros que necesita para hacer su trabajo. Por ejemplo un EJB de sesión llamado `gestorPedidos` podría necesitar el acceso a otro EJB llamado `gestorAlmacen`. En EJBs se asume JNDI como la forma estándar de localizar objetos de negocio y otros recursos. Nótese que esto implica la dependencia del API JNDI y del propio servicio de JNDI para poder ejecutar y probar la aplicación. Genéricamente a esta forma de trabajar en la que un objeto es responsable de localizar a los que necesite a través de un API determinado se le llama **dependency lookup**.Tenemos un ejemplo en el siguiente fragmento de código (sin los `import`)

```
public class GestorPedidos implements SessionBean {
   private GestorAlmacen almacen;

   public void ejbCreate() {
           Context ctx = new InitialContext();
           Object o = ctx.lookup("java:comp/env/GestorAlmacen");
           GestorAlmacenHome home = (GestorAlmacenHome)
PortableRemoteObject.narrow(o, GestorAlmacenHome.class);
           almacen = home.create();
    }
}
```

Una alternativa al *dependency lookup* es conseguir que el propio contenedor resuelva las referencias a los objetos necesarios y se las pase al que las necesita a través de algún método tipo *callback*, que idealmente no debería depender de ningún API propio. Esto se denomina **dependency injection**, y aquí tenemos cómo quedaría el ejemplo anterior con esta filosofía (los objetos ya no son EJBs, así que se han eliminado las referencias a dicho API):

```
public class GestorPedidos {
   private GestorAlmacen almacen;
```

```
   public GestorAlmacen getAlmacen() {
          return almacen;
   }

   public void setAlmacen(GestorAlmacen almacen) {
          this.almacen = almacen;
   }
}
```

Evidentemente, este código por sí solo no hace mucho. Es necesario, para empezar, tener guardada en algún sitio (típicamente un fichero de configuración) información sobre el objeto de la clase `GestorAlmacen` al que hace referencia `GestorPedidos` (ya que por ejemplo podría ser necesario inicializarlo con determinados valores). El contenedor, cuando se cree el objeto `GestorPedidos`,es el encargado de instanciar un objeto de la clase `GestorAlmacen` y llamar automáticamente al método `setAlmacen` con la referencia a dicho objeto. El objeto `GestorPedidos` ya no busca el otro objeto del que depende (*dependency lookup*), sino que esta dependencia se la "inyecta" el contenedor (*dependency injection*). Como puede verse, en realidad la idea es bastante sencilla y consigue eliminar del código la dependencia de APIs propios del contenedor. Lo único que hacemos es seguir la convención habitual en *JavaBeans* de usar métodos `get/set`, pero el código podría probarse perfectamente fuera del contenedor siempre que creáramos manualmente "objetos de prueba" para resolver las dependencias.

Spring proporciona un formato XML para almacenar las relaciones entre objetos de negocio y los valores iniciales para sus propiedades. El contenedor se ocupa, en tiempo de ejecución de instanciar los objetos necesarios y resolver las referencias. En el siguiente apartado veremos cómo definir los beans y las relaciones entre ellos.

3. Definir e instanciar beans

Lo habitual es almacenar la información sobre los beans en un fichero de configuración XML, aunque también se puede especificar en forma de código Java. El formato general del fichero XML con los beans es el siguiente:

```
<?xml version="1.0"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
  <bean id="miBean" class="mipaquete.MiClase">
     <!-- propiedades del bean y referencias a otros beans -->
     ...
  </bean>
```

```
  <bean id="..." class="...">
    ...
  </bean>
  ...
</beans>
```

Nota:

En Spring 2.0 se usan *schemas* para la definir la gramática del XML, en lugar de los DTDs que se usaban en la 1.2 y anteriores. Aunque el DTD antiguo sigue siendo válido, se anima a los desarrolladores a usar el schema, ya que es mucho mejor desde el punto de vista de la capacidad de validación y además incorpora las mejoras de sintaxis de la versión 2.

Instanciar el contenedor de beans es bastante sencillo. Hay varias formas de hacerlo, todas ellas implementan el interfaz `BeanFactory`

En una **aplicación web**, en el `web.xml` especificaremos la lista de ficheros XML con definiciones de beans y el contenedor web será el encargado de poner en marcha el contenedor de beans de Spring al arrancar la aplicación. Para ello se usa el mecanismo estándar de ejecutar código al arrancar una aplicación web: los *listener*. El `web.xml` quedaría así:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app ...>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/misBeans.xml</param-value>
  </context-param>

  <listener>
     <listener-class>
        org.springframework.web.context.ContextLoaderListener
     </listener-class>
  </listener>

  <!--  resto de etiquetas del web.xml -->
  ...
</web-app>
```

La clase `ContextLoaderListener` carga el fichero o ficheros XML especificados en el `<context-param>` llamado `contextConfigLocation` (suponemos que el fichero `misBeans.xml` está en el directorio `WEB-INF`). Como `<param-value>` se puede poner el nombre de varios ficheros XML, separados por espacios o comas.

Para acceder al bean desde cualquier página JSP o clase Java de la aplicación web se puede hacer uso de la clase `WebApplicationContext`:

```
<%@ page import ="org.springframework.web.context.*,
org.springframework.web.context.support.*" %>
<%@ page import ="springbeans.*, mipaquete.MiClase" %>
<html>
<head>
   <title>Acceso a beans de spring desde un JSP</title>
</head>
<body>
<%
  ServletContext sc = getServletContext();
  WebApplicationContext wac =
WebApplicationContextUtils.getWebApplicationContext(sc);
  MiClase ejemplo = (MiClase) wac.getBean("miBean");
%>
</body>
</html>
```

En una **aplicación Java de escritorio**, la forma más usada de instanciar el contenedor es a través de `XmlBeanFactory`.

```
ClassPathResource res = new ClassPathResource("misBeans.xml");
XmlBeanFactory factory = new XmlBeanFactory(res);
```

Donde el `XmlBeanFactory` es un tipo especial de `BeanFactory` que lee la configuración de un fichero XML (de hecho, el de uso más común). El fichero `misBeans.xml` se buscaría en cualquier directorio del CLASSPATH, al estar usando la clase `ClassPathResource`. Una vez creado el `BeanFactory`, para obtener un bean basta con hacer: (siguiendo con el código anterior)

```
MiClase ejemplo = (MiClase) factory.getBean();
```

Nota:

Nótese que en los ejemplos mostrados no estamos usando *dependency injection* para acceder al `BeanFactory`, sino *dependency lookup*, ya que necesitamos pedirle explícitamente a Spring que nos lo localice. Si se usan las características de MVC que incorpora Spring la necesidad de hacer *dependency lookup* desaparece totalmente, como veremos en sesiones posteriores del módulo.

4. Especificar las propiedades de un bean

Si el trabajo del contenedor se limitara a instanciar beans del mismo modo que lo podemos hacer con un simple `new` sin parámetros, no sería demasiado útil. El interés estriba en la posibilidad de **definir valores iniciales para los beans y relaciones entre ellos**. Los objetos de la capa de negocio suelen trabajar "en equipo": un bean necesitará una referencia a una instancia de otros para hacer su trabajo.

Las propiedades del bean se definen con la etiqueta `<property>`. Pueden ser Strings,

valores booleanos o numéricos y Spring los convertirá al tipo adecuado, siempre que la clase tenga un método `setXXX` para la propiedad. Podemos convertir otros tipos de datos (fechas, expresiones regulares, URLs, ...) usando lo que en Spring se denomina un `PropertyEditor`. Spring incorpora varios predefinidos y también podemos definir los nuestros. Veremos su uso en el siguiente apartado

Por ejemplo, supongamos que en una aplicación web necesitamos un buscador (que implementaremos con el bean `Buscador`) y deseamos otro bean para almacenar las preferencias de búsqueda. La clase Java para almacenar las preferencias sería un *JavaBean* común:

```
package springbeans;
public class PrefsBusqueda {
    private int maxResults;
    private boolean ascendente;
    private String idioma;

    public boolean isAscendente() {
        return ascendente;
    }
    public void setAscendente(boolean ascendente) {
        this.ascendente = ascendente;
    }
    public String getIdioma() {
        return idioma;
    }
    //resto de get/set
    ...
}
```

Los valores iniciales para las propiedades pueden configurarse en XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
   <bean id="misPrefs" class="springbeans.PrefsBusqueda">
     <property name="maxResults">
       <value>100</value>
     </property>
     <property name="idioma">
       <value>es</value>
     </property>
     <property name="ascendente">
       <value>true</value>
     </property>
  </bean>
</beans>
```

En la versión 2 de Spring se ha añadido una forma alternativa de especificar propiedades que usa una sintaxis mucho más corta. Se emplea el espacio de nombres `http://www.springframework.org/schema/p`, que permite especificar las propiedades del bean como atributos de la etiqueta `bean`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
   <bean id="misPrefs"
class="es.ua.jtech.spring.prueba.beans.PrefsBusqueda"
       p:maxResults="100" p:idioma="es" p:ascendente="true">
   </bean>
</beans>
```

Las propiedades también pueden ser colecciones: `Lists`, `Maps`, `Sets` o `Properties`. Supongamos que en el ejemplo anterior queremos una lista de idiomas preferidos:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
   <bean id="misPrefs" class="springbeans.PrefsBusqueda">
    <property name="listaIdiomas">
      <list>
       <value>es</value>
       <value>en</value>
      </list>
    </property>
    <!-- resto de propiedades -->
  </bean>
</beans>
```

Para ver cómo se especifican los otros tipos de colecciones, acudir a la documentación de referencia de Spring.

4.1. Conversión de tipos automática

Como ya se ha comentado, Spring convierte automáticamente los valores puestos en el XML a numéricos, Strings y booleanos, pero podemos convertir los datos a cualquier clase Java sin más que definir y/o usar un `PropertyEditor`, que no es más que una clase que se encarga de realizar esta conversión. Spring incluye bastantes de estos conversores predefinidos, y si

no nos bastan podemos definir los nuestros propios. La definición de conversores propios queda fuera del ámbito de estos apuntes. No obstante, aunque algunos de ellos ya están definidos, como el conversor a `Date`, no están registrados por defecto en el contenedor y no pueden usarse directamente. Vamos a ver aquí un ejemplo de cómo registrar el `CustomDateEditor`, que nos permitirá convertir a `Date` a partir de patrones cadena del estilo de los que usa la clase `DateFormat`. El registro de un nuevo `PropertyEditor` se hace a través de la clase `CustomEditorConfigurer`

Por ejemplo, supongamos que al bean `PrefsBusqueda` se le añade una propiedad de tipo `java.util.Date` llamada `desde` para poder especificar la fecha más antigua que puede tener un documento que devuelva el buscador.

Si miramos el API de Spring veremos que la clase `CustomEditorConfigurer` tiene una propiedad de tipo `Map` en la que hay que colocar la clase destino de la conversión (en nuestro caso `java.util.Date`) y el `PropertyEditor` registrado (en nuestro caso `CustomDateEditor`). El API de este último nos dice qué parámetros requiere (consultar la documentación de Spring al efecto). El XML cambiaría como sigue:

```
...
<bean id="misPrefs" class="es.ua.jtech.spring.prueba.beans.PrefsBusqueda"
    p:maxResults="100" p:idioma="es" p:ascendente="true"
p:desde="10/02/2007">
</bean>
<bean id="miConfig"
class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
        <map>
            <entry key="java.util.Date">
                <bean
class="org.springframework.beans.propertyeditors.CustomDateEditor">
                    <constructor-arg index="0">
                        <bean class="java.text.SimpleDateFormat">
                            <constructor-arg value="dd/MM/yyyy"/>
                        </bean>
                    </constructor-arg>
                    <constructor-arg index="1" value="false"/>
                </bean>
            </entry>
        </map>
    </property>
</bean>
...
```

5. Dependencias entre beans

Cuando un bean hace uso de otros, en Spring se dice que tiene "colaboradores" (*collaborators*). Para definir un colaborador de un bean se usa el atributo `ref` con el nombre del colaborador. Por ejemplo, supongamos que queremos decir que un objeto `Buscador` necesita un `PrefsBusqueda`.

```
...
<bean id="misPrefs" class="springbeans.PrefsBusqueda">
   <!--propiedades de este bean -->
   ...
</bean>
<bean id="miBuscador" class="springbeans.Buscador">
   <property name="prefs">
      <ref bean="misPrefs"/>
   </property>
<bean>
...
```

Para que esto funcione, lo único que necesitamos es que la clase `Buscador` tenga una propiedad llamada `prefs` que sea de la clase `PrefsBusqueda` y con métodos get/set.

```
package springbeans;

public class Buscador {
   private PrefsBusqueda prefs;

   public PrefsBusqueda getPrefs() {
      return prefs;
   }

   public void setPrefs(PrefsBusqueda misPrefs) {
      this.prefs = misPrefs;
   }
}
```

Cuando arranque la aplicación, Spring resolverá las dependencias, instanciará un bean de la clase `PrefsBusqueda` y le pasará la referencia al bean `miBuscador` llamando a `setPrefs`.

En lugar de hacer una referencia al "bean dependiente" con `ref` podemos usar un "bean interno" (un *inner bean*) poniendo directamente un bean dentro de otro. Obsérvese que el bean interno no necesita nombre:

```
...
<bean id="miBuscador" class="springbeans.Buscador">
   <property name="prefs">
      <bean class="springbeans.PrefsBusqueda">
         <!--propiedades de este bean -->
         ...
      </bean>
   </property>
```

```
</bean>
...
```

5.1. Determinar las dependencias de manera automática

Se pueden omitir las referencias a los colaboradores en el fichero de configuración, y decirle a Spring que mediante *reflection* determine cuáles son las referencias necesarias. Esto se denomina **autowiring**. La forma más común es `byType`, que consiste en que Spring asigna el bean del tipo adecuado de entre todos los definidos. Siguiendo con el ejemplo anterior:

```
...
<bean id="misPrefs" class="springbeans.PrefsBusqueda">
   <!--propiedades de este bean -->
   ...
</bean>
<bean id="miBuscador" class="springbeans.Buscador" autowire="byType">
</bean>
```

No hace falta especificar la referencia al bean colaborador en las propiedades. Al usar *autowiring*, Spring detecta automáticamente la referencia (ya que en la clase java hay un método `setPrefs`) y la resuelve por la clase del objeto (ya que `getPrefs` tiene como parámetro un `PrefsBusqueda`, y en el fichero de definición de beans hay uno definido de esta clase). Otro tipo de *autowiring* es `byName`, en el que si hay una propiedad de un bean llamada XXX (o sea, hay un getXXX/setXXX) se asocia automáticamente con un bean de Spring que tenga el mismo nombre. En general, se recomienda usar el `byType` porque es menos propenso a errores.

Si se usa *autowiring* es recomendable decirle a Spring que chequee que todas las propiedades de los *beans* se han resuelto correctamente. Para ello se usa el atributo `dependency-check`.

```
...
<bean id="misPrefs" class="springbeans.PrefsBusqueda">
   <!--propiedades de este bean -->
   ...
</bean>
<bean id="miBuscador" class="springbeans.Buscador" autowire="byType"
        dependency-check="objects">
</bean>
```

Con `dependency-check="objects"` especificamos que se chequeen solo las referencias a objetos, pero no así las propiedades "simples" (tipos primitivos y cadenas). Esto último se consigue poniendo como valor del atributo `all` (o `simple` solo para los tipos "simples").

5.2. Constructor injection

La forma de *dependency injection* que venimos usando se denomina técnicamente **setter injection** , ya que el contenedor le proporciona a los objetos las dependencias que necesitan a través de métodos `setXXX`. Otra forma de conseguir esto sería "inyectar" las dependencias en el propio constructor. Aunque lo más habitual en Spring es usar *setter injection*, el *framework* también implementa esta última posibilidad. Lo único que hay que hacer en el fichero de definición de beans es sustituir las etiquetas `property` por una serie de `constructor-arg`.

```
...
<bean id="misPrefs" class="springbeans.PrefsBusqueda">
   <!--propiedades de este bean -->
   ...
</bean>
<bean id="miBuscador" class="springbeans.Buscador">
   <constructor-arg>
      <ref bean="misPrefs"/>
   </constructor-arg>
</bean>
```

Por supuesto, para que esto funcione, debe haber un constructor de la clase `Buscador` que tenga como único argumento un objeto de la clase `PrefsBusqueda`. Por limitaciones del API de *reflection*, si se usan varios argumentos del mismo tipo en un constructor, no es posible resolver la ambigüedad por nombre del parámetro, de modo que en la definición del bean habrá que especificar el orden. Esto se hace con el atributo `index`.

6. Ámbito de los beans

Por defecto, los beans en Spring son *singletons*. Esto significa que el contenedor solo instancia un objeto de la clase, y cada vez que se pide una instancia del bean en realidad se obtiene una referencia al mismo objeto. Recordemos que se solicita una instancia de un bean cuando se llama a `getBean()` o bien cuando se "inyecta" una dependencia del bean en otro.

El ámbito *singleton* es el indicado en muchos casos. Probablemente un solo `GestorPedidos` comentado pueda encargarse de todas las tareas de negocio relacionadas con pedidos, y por dar otro ejemplo si el bean representa un `DataSource`, aunque lo referenciemos en varios sitios en realidad siempre queremos acceder al mismo objeto.

Podemos usar otros ámbitos para el bean, a través del atributo `scope` de la etiqueta `bean`. Por ejemplo, para especificar que queremos una nueva instancia cada vez que se solicite el bean, se usa el valor `prototype`

```
...
```

```
  <bean id="miBean" class="mipaquete.MiClase" scope="prototype">
     <!-- propiedades del bean y referencias a otros beans -->
     ...
  </bean>
...
```

6.1. Ámbitos especiales para aplicaciones web

En **aplicaciones web**, se pueden usar además los ámbitos de `request` y `session` (hay un tercer ámbito llamado `globalSession` para uso exclusivo en portlets). Para que el contenedor pueda gestionar estos ámbitos, es necesario usar un `listener` especial cuya implementación proporciona Spring. Habrá que definirlo por tanto en el `web.xml`

```
...
<web-app>
  ...
  <listener>
<listener-class>org.springframework.web.context.request.RequestContextListener</listene
  </listener>
  ...
</web-app>
```

Ahora ya podemos usar los ámbitos especiales para aplicaciones web. Por ejemplo para definir un bean que tenga como ámbito la sesión HTTP:

```
...
  <bean id="miBean" class="mipaquete.MiClase" scope="session">
     <!-- propiedades del bean y referencias a otros beans -->
     ...
  </bean>
...
```

6.2. Dependencias entre beans con distinto ámbito

Cuando un bean depende de otro que tiene un ciclo de vida más corto se pueden plantear problemas. Por ejemplo, supongamos que un bean de la clase `Buscador` con ámbito `singleton` depende de otro de la clase `PrefsBusqueda` con ámbito de sesión HTTP.

```
...
<bean id="misPrefs" class="springbeans.PrefsBusqueda" scope="session"/>

<bean id="miBuscador" class="springbeans.Buscador">
    <property name="prefs" ref="misPrefs"/>
</bean>
...
```

La configuración anterior tiene un problema que quizá no sea evidente a primera vista: como

el bean `miBuscador` es un *singleton*, el contenedor lo instancia una sola vez, inyectando la dependencia de `misPrefs` *también una sola vez*. Pero nosotros no queremos eso, queremos que `misPrefs` se conserve durante la duración de la sesión HTTP y que si ésta se invalida se instancie *un nuevo* `misPrefs`.

Spring soluciona este problema con el uso de lo que se denomina un *proxy AOP*, cuyo funcionamiento veremos en sesiones posteriores. Baste saber por ahora que es un objeto que se "interpone" entre los dos beans y que se encarga de manera "inteligente" de gestionar el ciclo de vida del bean con ámbito más corto, creando nuevas instancias de él cuando sea necesario. El bean `miBuscador` permanece ajeno al hecho de que trata no con el verdadero `misPrefs` sino con un proxy, que toma sus peticiones y las pasa al verdadero destinatario. En realidad, si nos ceñimos simplemente a la sintaxis de Spring, el uso de este proxy AOP no puede ser más sencillo:

```
...
<bean id="misPrefs" class="springbeans.PrefsBusqueda" scope="session"/>
    <aop:scoped-proxy/>
</bean>
<bean id="miBuscador" class="springbeans.Buscador">
    <property name="prefs" ref="misPrefs"/>
</bean>
...
```

Si el bean "proxyficado" es una clase en lugar de un interfaz java (como es precisamente el ejemplo que nos ocupa) necesitaremos tener la librería CGLIB en el CLASSPATH. Esto ocurre porque las librerías estándar de Java permiten generar estos proxys de manera automática a partir de interfaces, pero no de clases. CGLIB viene a solucionar precisamente este último caso.

6.3. Notificaciones del ciclo de vida

En algunos casos puede ser interesante llamar a un método del bean cuando éste se inicializa o destruye. Para ello se pueden usar dos atributos en la definición: `init-method` y `destroy-method`. Por ejemplo:

```
...
<bean id="miBuscador" class="springbeans.Buscador" init-method="inicializa"
destroy-method="destruye">
</bean>
...
```

Ambos deben ser métodos sin parámetros. El método de inicialización se llama justo después de que Spring resuelva las dependencias e inicialice las propiedades del bean.

7. Acceso a recursos JNDI con beans de Spring

Un bean *singleton* es perfecto para representar una referencia a un recurso JNDI como un DataSource. Aunque en versiones anteriores la forma de asociar un bean a un recurso JNDI era más compleja, la sintaxis se ha simplificado mucho con la introducción del espacio de nombres `jee`. Para usar este espacio de nombres hay que definir el siguiente preámbulo en el XML de configuración (en negrita aparece la definición del espacio de nombres propiamente dicho)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns:jee="http://www.springframework.org/schema/jee"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                http://www.springframework.org/schema/jee
            http://www.springframework.org/schema/jee/spring-jee-2.0.xsd">
...
</beans>
```

Hacer por ejemplo que un DataSource cuyo nombre JNDI es `jdbc/MiDataSource` sea un bean de Spring es muy sencillo con la etiqueta `jee:jndi-lookup`

```
<jee:jndi-lookup id="miBean" jndi-name="jdbc/MiDataSource"/>
```