Spring en SIGEM

Índice

1. Uso de Spring en SIGEM

Podríamos decir que el uso de Spring en SIGEM es bastante limitado, ya que se reduce al uso del contenedor de beans. No obstante, esta funcionalidad está en el núcleo de SIGEM y es básica para el sistema. Los beans de Spring son los que hacen posible el acceso a la capa SOA de manera transparente, independientemente de que la implementación sea local o a través de servicios web.

Nota:

Como el único componente de Spring que se usa en SIGEM es el núcleo (*core*), sería mucho más eficiente incluir en los proyectos solo este módulo en lugar de la distribución completa de Spring (que es lo que incluye ahora mismo SIGEM). El `spring-core.jar` se puede obtener de la distribución de Spring,en la carpeta `modules`, accesible a través de la web de Spring

La versión de Spring usada en SIGEM es la 2.0. En concreto, los JARs incluidos son de la 2.0.6. Hay que tener en cuenta que el uso de anotaciones para definir beans empezó a usarse extensivamente a partir de la 2.5. Por lo demás lo visto en clase es totalmente aplicable a la 2.0. Si se van a usar más características de Spring y no se desea migrar a la 2.5 se recomienda consultar las secciones de los apuntes que tratan de la configuración en formato XML.

1.1. Configuración de los beans

El fichero principal de configuración de Spring está en el módulo `SIGEM_Core`, que como veremos es también el módulo encargado de instanciar los beans. Además, varios módulos incluyen ficheros adicionales, pero se reducen a referenciar ficheros `.properties` adicionales.

En la configuración se hacen uso de algunas construcciones de Spring que no hemos visto hasta el momento en el curso por motivos de tiempo. Vamos a verlas brevemente. Por supuesto, en la documentación de Spring aparecen descritas con más detalle.

1.1.1. Inicialización "diferida" (lazy)

En todos los ejemplos que hemos hecho en las distintas sesiones de Spring, los beans con ámbito *singleton* se instanciaban durante el arranque del contenedor. Esta es la configuración por defecto de Spring, basándose en el razonamiento de que permite detectar los fallos en las dependencias durante el arranque, facilitando el mantenimiento. No obstante, en SIGEM la mayoría de beans se inicializan de manera *lazy*, es decir, solo cuando se solicitan (desde otro bean o desde el código java). Para ello basta con poner un atributo `lazy-init="true"` en la configuración del bean.

1.1.2. Parámetros de configuración en archivos .properties

La clase `PropertyPlaceholderConfigurer` se usa para externalizar propiedades a un fichero `.properties`. De este modo se pueden modificar propiedades que dependerán del entorno de instalación (como URLs de bases de datos, usuarios, passwords, etc) sin tocar el XML con los beans, que puede ser complejo o "intimidatorio" para el administrador sin experiencia previa en Spring.

La propiedad `locations` especifica la localización física del archivo .properties. Para usar una propiedad se pone su clave en dicho archivo en el formato `${clave}`.

Por ejemplo, veamos las líneas 166 a 176 del `SIGEM_Spring.xml`

```
166 <bean
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
167      <property name="locations"
value="classpath:database.properties"/>
168 </bean>
169
170 <!-- Para utilizar desde fuera del servidor de aplicaciones -->
171 <bean id="DATASOURCE_STANDALONE"
         class="ieci.tecdoc.sgm.core.db.impl.DataSourceSpringImpl"
lazy-init="true">
172    <property name="driverClassName"
value="${sigem.springDatasource.driver}"/>
173    <property name="url"
value="${sigem.springDatasource.database}"/>
174    <property name="username"
value="${sigem.springDatasource.user}"/>
175    <property name="password"
value="${sigem.springDatasource.password}"/>
176 </bean>
```

Como se especifica en `locations` el fichero .properties debe estar en el classpath y su nombre físico es `database.properties`.

1.1.3. Acceso a recursos JNDI

En el curso hemos visto cómo acceder a recursos JNDI usando la etiqueta `<jee:jndi-lookup/>`. Antes de la introducción de esta etiqueta, el acceso a recursos JNDI en Spring se hacía a través de la clase `JndiObjectFactoryBean`, que se sigue manteniendo por motivos de compatibilidad.

Por ejemplo, en las líneas 178 a 184 del `SIGEM_Spring.xml` Se define un bean llamado `DATASOURCE_ADMINISTRACION` que sirve para acceder a un DataSource, y se le inyecta el Datasource JNDI correspondiente.

```
178 <!-- Configuración para el origen de datos de Administración de
SIGEM -->
179 <bean id="DATASOURCE_ADMINISTRACION"
     class="ieci.tecdoc.sgm.core.db.impl.DataSourceJNDISpringImpl"
     lazy-init="true">
180    <property name="jndiDataSource" ref="jndiDataSource"/>
181 </bean>
182 <bean id="jndiDataSource"
     class="org.springframework.jndi.JndiObjectFactoryBean"
     lazy-init="true">
183   <property name="jndiName"
value="${sigem.administracion.jndiDatasource.jndiName}"/>
184 </bean>
```

Aunque `<jee:jndi-lookup/>` ya existe en la versión 2.0 de Spring, no se puede usar directamente en el archivo ya que esta etiqueta necesita del uso del schema XML mientras que el archivo de configuración SIGEM usa el formato antiguo, basado en DTD. Esta era la forma de archivo de configuración más común en Spring 1.x y probablemente se haya "heredado" de esa época en la versión actual de SIGEM.

1.1.4. Alias para beans

Esta es una característica de Spring que nos permite poner alias a un bean de modo que distintos módulos de la aplicación lo puedan llamar de distinta manera. Continuando con el `SIGEM_Spring.xml`:

```
160 <alias name="DATASOURCE_ADMINISTRACION"
alias="SIGEM_ADMIN_DATASOURCE"/>
...
178 <!-- Configuración para el origen de datos de Administración de
SIGEM -->
179 <bean id="DATASOURCE_ADMINISTRACION"
     class="ieci.tecdoc.sgm.core.db.impl.DataSourceJNDISpringImpl"
lazy-init="true">
```

Así, desde nuestro código, o desde otros beans, podemos referenciar a éste de modo indiferente como DATASOURCE_ADMINISTRACION (su nombre "real") o como SIGEM_ADMIN_DATASOURCE (su alias). En realidad la primera denominación no se usa en ningún sitio de SIGEM, pero esa ya es otra cuestión...

1.1.5. Factory beans

Cuando se usan "factorías de objetos" en un API, no se crean objetos llamando a new() sino a través de algún método. Así el código se independiza de la clase concreta a crear. Estas factorías serían el equivalente en código Java a la abstracción proporcionada por el contenedor de Spring. Si un bean de Spring depende de un objeto que se crea con una

factoría, tenemos que configurarlo con el atributo `factory-method`, en el que decimos qué método hay que llamar para obtener una nueva instancia del bean. El objeto que implementa este método será el bean referenciado con `factory-bean`.

1.2. Uso de los beans

En SIGEM, los beans de Spring se usan básicamente para dos propósitos:

- Acceder a los DataSources
- Acceder a los servicios de la capa SOA. Como ya visteis en la sesión de integración de servicios web, cada servicio tiene dos implementaciones distintas: la remota y la local. Ambas son beans de Spring e instanciamos una o la otra distinguiéndolas por el nombre del bean.

Hay un tercer uso, importante pero casi "anecdótico". Se emplea un fichero de beans para definir la lista de idiomas que soporta SIGEM. Dicha lista está en un fichero properties, por lo que se usa la clase `PropertiesFactoryBean`, que en versiones 1.x de Spring era la única alternativa para referenciar un .properties. En la versión actual de Spring se usaría la etiqueta "util:properties":

```
<util:properties id="idiomas.propiedades"
location="classpath:com.ieci.tecdoc....idiomas.properties"/>
```

Para ver los otros dos usos, emplearemos como ejemplo el módulo de administración del registro presencial, que ya vimos en el proyecto de integración de Struts (aunque solo la capa web).

1.2.1. Acceso a la capa SOA

Los servlets o las acciones de Struts necesitan de los servicios de la capa de negocio. Estos servicios se obtienen a través de la clase `LocalizadorServicios` del módulo `SIGEM_Core`.

Por ejemplo, en la acción `ListadoLibrosAction`, que se ejecuta al entrar en el módulo de administración de registro presencial, podemos ver la llamada al localizador:

```
19 public class ListadoLibrosAction extends RPAdminWebAction {

21 private static final Logger logger =
Logger.getLogger(ListadoLibrosAction.class);
22 public ActionForward executeAction(ActionMapping mapping,
ActionForm form,
23                 HttpServletRequest request, HttpServletResponse
response) throws Exception {
24
25         ServicioRPAdmin oServicio =
LocalizadorServicios.getServicioRPAdmin();
```

El método de localización del servicio usa como nombre del mismo "RPADMIN_SERVICE_DEFAULT_IMPL", es decir la implementación local. Al final se acaba llamando a otro método que carga la definición del bean de un fichero de configuración de Spring:

```
774 public static ServicioRPAdmin getServicioRPAdmin(Config
poConfig, String pcImpl) ...
775         if(poConfig != null){
776                 ServicioRPAdmin oService = null;
777                 try {
778                   oService =
(ServicioRPAdmin)poConfig.getBean(pcImpl);
779                 } catch (Exception e) {
780                   throw new
SigemException(SigemException.EXC_GENERIC_EXCEPCION, e);
781                 }
782                 return oService;
783         }else{
784                 throw new
SigemException(SigemException.EXC_GENERIC_EXCEPCION);
785         }
786 }
```

Básicamente la clase `Config` no es más que una capa sobre un `ClassPathXMLContext` de Spring, como se puede comprobar si vamos a su código fuente. Dicho context es el que finalmente carga el bean "RPADMIN_SERVICE_DEFAULT_IMPL" del archivo ya nombrado "SIGEM_Spring.xml". Finalmente, en dicho archivo podemos ver que la clase que se usará es `ServicioRPAdminAdapter`.

```
<alias name="&RPADMIN;.&SIGEM;.&API;"
alias="RPADMIN_SERVICE_DEFAULT_IMPL"/>

<bean abstract="true" id="&RPADMIN;"
   class="ieci.tecdoc.sgm.core.services.rpadmin.ServicioRPAdmin">
</bean>

<bean id="&RPADMIN;.&SIGEM;.&API;"
class="ieci.tecdoc.sgm.rpadmin.ServicioRPAdminAdapter"
  parent="&RPADMIN;" lazy-init="true">
</bean>
```

Como puede observarse en la definición anterior, el bean que implementa el servicio (`ServicioRPAdminAdapter`) es "hijo" del interfaz `ServicioRPAdmin`. Esta idea de bean "hijo" se usa en Spring para heredar la configuración del bean padre. En este caso el padre no tiene ninguna configuración especial, pero es de suponer que era la idea inicial del diseño especificar las propiedades en los beans "padre".

1.2.2. Acceso a los DataSources

Los beans también se usan para abstraer los datasources. En este caso, las encargadas de cargar el bean son las clases `DataSourceManager` y `DataSourceManagerMultiEntidad`, del módulo `SIGEM_Core`. El mecanismo de carga es muy similar al del acceso a la capa SOA,por lo que no nos extenderemos más en él (Uso de la clase propia `Config` que enmascara un `ClassPathXMLApplicationContext`).

2. Enlace de Spring con la capa web

Como vimos en una sesión anterior, la capa web del módulo de administración del registro presencial está implementada en Struts. Por muchas que sean las ventajas de Spring MVC frente a Struts, este es un *framework* muy probado y no tiene sentido hacer el cambio salvo que estuviéramos implementando un módulo desde cero. Otra cuestión que sí es mejorable es el enlace entre la capa web (Struts) y la de negocio (Spring).

El código del localizador de servicios es demasiado complejo para una tarea tan sencilla como la de cargar un bean. Si se observa el fuente se verá que es muy repetitivo. El problema es que al no ser las acciones de struts beans ellas mismas, no pueden recibir la inyección de componentes de negocio. Reconociendo el problema, los desarrolladores de Spring realizaron hace tiempo un plugin para Struts que permite enlazar acciones con beans de modo transparente, **permitiendo la inyección de un bean en una acción de Struts**. Esto elimina la necesidad del código del localizador de servicios. Vamos a introducir esta modificación a modo de ejemplo en una acción concreta (la de listar oficinas, que se ejecuta nada más entrar en el módulo).

2.1. Configuración del plugin

El plugin se distribuye junto con los demás JARs de Spring, en un fichero spring-struts.jar. Nosotros tomaremos aquí el correspondiente a la versión 2.0.6, que es la usada en SIGEM.

Se trata de un plugin de Struts, por lo que hay que configurarlo en el `struts-config.xml`. Debemos hacer dos cosas:

- Instanciar el plugin, indicando dónde están los archivos de definición de beans. Además del original de "SIGEM_Spring.xml" nosotros usaremos uno adicional para hacer el enlace entre Spring y Struts, llamado "spring-struts.xml". Nótese que el primero se incluye desde el módulo Core dentro de un JAR, y por eso indicamos que está "en el classpath" (prefijo `classpath:`). El segundo es propio del módulo web y por eso lo hemos puesto dentro de `WEB-INF`.

```
(al final del struts-config, justo después del otro plugin)

<plug-in
className="org.springframework.web.struts.ContextLoaderPlugIn">
                <set-property property="contextConfigLocation"
value="classpath:SIGEM_spring.xml,WEB-INF/spring-struts.xml" />
```

```
</plug-in>
```

- Definir un controlador para Struts que en realidad está implementado en la librería de Spring. Su trabajo será inyectar los beans de negocio en las acciones antes de ejecutarlas.

```
(sustituyendo a la etiqueta <controller/> vacía del struts-config)
(es decir, después de los action-mappings)

<controller>
  <set-property property="processorClass"
value="org.springframework.web.struts.DelegatingRequestProcessor"
/>
</controller>
```

2.2. Uso del plugin

En nuestro fichero de definición de beans (WEB-INF/spring-struts.xml) debemos definir un bean por cada acción. Su nombre será el "path" de la acción en el struts-config y su clase la de la acción. A este bean le inyectaremos los de la capa de negocio. Por ejemplo:

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
   "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
        <bean name="/listadoOficinas"
class="ieci.tecdoc.sgm.rpadmin.struts.acciones.oficinas.ListadoOficinasAction">
             <property name="servicioOficinas"
ref="RPADMIN_SERVICE_DEFAULT_IMPL"></property>
        </bean>
</beans>
```

Nótese que referenciamos el bean "RPADMIN_SERVICE_DEFAULT_IMPL" del fichero Spring principal de SIGEM

Y ahora cambiamos el código Java de `ListadoOficinasAction` para añadirle un "setter" para la propiedad "servicioOficinas", y en él instanciamos el objeto de negocio:

```
public class ListadoOficinasAction extends RPAdminWebAction {
        ServicioRPAdmin serv;

        private static final Logger logger =
Logger.getLogger(ListadoOficinasAction.class);

        public void setServicioOficinas(ServicioRPAdmin serv) {
                this.serv = serv;
                logger.log(Priority.INFO, "servicio RPAdmin
inyectado");
        }
```

```
        public ActionForward executeAction(ActionMapping mapping,
ActionForm form,
                           HttpServletRequest request,
HttpServletResponse response)
                              throws Exception {

        Oficinas oficinas = serv.obtenerOficinas(
SesionHelper.obtenerEntidad(request));
```

Nótese que la mejora no es en la acción en sí, sino en la simplificación del acceso al objeto de negocio, eliminando totalmente la dependencia del LocalizadorServicios. Tenemos una capa menos y aún así seguimos conservando la abstracción, ya que realmente nos la daba el XML de Spring, no el localizador.

Una mejora evidente a la configuración sería usar el "autowiring" en el XML, simplificándolo:

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
   "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
        <bean name="/listadoOficinas"
class="ieci.tecdoc.sgm.rpadmin.struts.acciones.oficinas.ListadoOficinasAction"
            autowire="byType">
        </bean>
</beans>
```

3. Acceso remoto con Spring

Podemos usar Spring para hacer accesibles ciertos servicios de modo remoto, para clientes Swing o Java en la intranet. Para este tipo de cliente, la maquinaria que requiere un servicio web es excesiva y el rendimiento en la comunicación no será muy grande. Una solución Java-a-Java es más eficiente. Vamos a hacer accesible el servicio por HTTP, pero sería prácticamente igual hacerlo con un protocolo nativo Java como RMI.

Vamos a hacer accesible el servicio de administración del registro presencial.

3.1. Proyecto para exportar el servicio

Crearemos un proyecto de tipo web llamado "ServicioRPAdminExporter". Haremos la configuración:

1. Configuración de las librerías
   - Dependencia de otros proyectos: en las propiedades del proyecto, opción "Java EE module dependencies" marcaremos las dependencias de este servicio: SIGEM_Core, SIGEM_ClasesBase y SIGEM_RegistroPresencialAdmin
   - JARs: necesitamos el commons-logging y el spring.jar que podemos copiar de

algún otro proyecto de SIGEM.

2. en la carpeta META-INF introduciremos el siguiente "context.xml", necesario para poder acceder a las fuentes de datos.

```
<Context>
  <ResourceLink global="jdbc/sigemAdmin" name="jdbc/sigemAdmin"
                type="javax.sql.DataSource"/>
  <ResourceLink global="jdbc/registroDS_000"
name="jdbc/registroDS_000"
                type="javax.sql.DataSource"/>
</Context>
```

3. En el web.xml configuraremos
   - La localización del fichero con los beans de spring. Será en realidad el que viene con SIGEM, que es donde está implementada la capa de negocio.
   - El servlet dispatcher para permitir acceso remoto a través de URLs del tipo /remoting/*

```
<web-app id="WebApp_ID" version="2.4"
 xmlns="http://java.sun.com/xml/ns/j2ee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
   http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
 <display-name>ServicioRPAdminExporter</display-name>
 <context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:SIGEM_spring.xml</param-value>
 </context-param>
 <listener>
<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class
 </listener>
 <servlet>
  <servlet-name>dispatcher</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
 </servlet>
 <servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>/remoting/*</url-pattern>
 </servlet-mapping>
...
```

4. Crearemos un WEB-INF/dispatcher-servlet con la configuración del servicio exportado:

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
        <bean name="/RPAdmin"
class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
                <property name="service"
ref="RPADMIN_SERVICE_DEFAULT_IMPL"/>
```

```
                <property name="serviceInterface"
value="ieci.tecdoc.sgm.core.services.rpadmin.ServicioRPAdmin">
                </property>
        </bean>
</beans>
```

5. Comprobaremos a través del navegador que la URL "http://localhost:8080/ServicioRPAdminExporter/remoting/RPAdmin" no da un 404. Dará una excepción de tipo EOF ya que accedemos a ella con un cliente que no es el adecuado.

3.2. Proyecto cliente

Crearemos un proyecto de tipo Java llamado "ClienteRPAdmin"

1. Librerías y dependencias:
   - en el "Configure Build Path", solapa "Projects" marcaremos como requerido el proyecto "SIGEM_Core" ya que en él se encuentra definido el interfaz del servicio al que vamos a acceder.
   - Copiaremos los JARs de spring y de commons-logging de cualquier otro proyecto y los añadiremos al build path.
2. Crearemos un spring.xml en la raíz de los fuentes (carpeta src) con el bean cliente:

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
   "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
        <bean id="RPAdmin"
class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
                <property name="serviceUrl"
value="http://localhost:8080/ServicioRPAdminExporter/remoting/RPAdmin"/>
                <property name="serviceInterface"
value="ieci.tecdoc.sgm.core.services.rpadmin.ServicioRPAdmin"></property>
        </bean>

</beans>
```

   Por desgracia, en Eclipse por defecto cuando se crea un XML no se copia a la carpeta de binarios junto con las clases. Necesitamos que esté ahí porque lo vamos a cargar desde el classpath. Debemos ir a las propiedades del proyecto y en la opción Java compiler > Building > Enable specific settings > Filtered resources debemos eliminar el *.xml, para indicar que SI queremos que se copie junto con los .class al compilar.
3. Crearemos una clase que actúe de cliente. Dicha clase tendrá un código como el siguiente, que accedería a la lista de oficinas de una entidad:

```
package sigem;

import java.util.Iterator;

import
```

```
org.springframework.context.support.ClassPathXmlApplicationContext;

import ieci.tecdoc.sgm.core.services.dto.Entidad;
import ieci.tecdoc.sgm.core.services.rpadmin.Oficinas;
import ieci.tecdoc.sgm.core.services.rpadmin.RPAdminException;
import ieci.tecdoc.sgm.core.services.rpadmin.ServicioRPAdmin;

public class Cliente {
        public static void main(String[] args) throws
RPAdminException {
                ClassPathXmlApplicationContext contexto;

                contexto = new
ClassPathXmlApplicationContext("spring.xml");
                ServicioRPAdmin servicio = (ServicioRPAdmin)
contexto.getBean("RPAdmin");
                Entidad e = new Entidad();
                e.setIdentificador("000");
                Oficinas lista = servicio.obtenerOficinas(e);
                for(int i=0;i<lista.count();i++) {
System.out.println(lista.get(i).getNombre());
                }

        }
}
```

4. Necesitaremos que las clases que van a viajar por la red sean serializables. Esto nos pasará con la clase Oficina, Oficinas y Entidad (la del paquete "ieci.tecdoc.sgm.core.services.dto")
5. Una vez hecho todo esto, ya podemos probar el cliente, ejecutándolo como una aplicación Java (¡no en el servidor!).