Acceso remoto a componentes y transaccionalidad

Índice

Aun siendo los EJBs complejos de programar y desplegar en un servidor de aplicaciones, ofrecen una serie de servicios extremadamente valiosos para el desarrollador, que tendría un trabajo aún más complicado si tuviera que implementar manualmente aspectos como la seguridad, la transaccionalidad y el acceso remoto. Veremos aquí cómo se puede conseguir la transaccionalidad y el acceso remoto en Spring con un nivel de complejidad mucho menor. La seguridad la dejaremos aparte por cuestiones de espacio, ya que es un tema amplio y objeto de un subproyecto completo de Spring.

1. Acceso remoto a componentes sin EJBs

Uno de los puntos que hacen atractivos a los EJB es que permiten simplificar la programación distribuida, proporcionando un acceso a objetos remotos "casi transparente" para el programador. Sin más que hacer una llamada a JNDI se puede localizar un objeto remoto y llamar a sus métodos "casi" como si estuviera en la máquina local. En este tema veremos alternativas para conseguir el mismo objetivo, que si bien no son tan sofisticadas, son mucho más ligeras que la implementación de muchos contenedores de EJBs.

1.1. Evaluación de las alternativas

Spring no proporciona una única alternativa a EJBs para acceso remoto. Según los requerimientos de la aplicación y las características de la implementación será más apropiada una u otra alternativa. Veamos cuáles son:

- **RMI:** A pesar de que usar RMI directamente pueda parecer algo "primitivo", Spring implementa una serie de clases que proporcionan una capa de abstracción sobre el RMI "puro", de modo que por ejemplo no hay que gestionar directamente el servidor de nombres, ni ejecutar manualmente `rmic` y el cliente puede abstraerse totalmente de que el servicio es remoto. RMI será la alternativa adecuada cuando nos interese **buen rendimiento, clientes Java y sepamos que el servidor de nombres no es un problema (p.ej. con firewalls)**
- **HTTP invoker:** Es una opción muy similar a la de RMI, usando serialización de Java, pero a través del puerto HTTP estándar, con lo que eliminamos los posibles problemas de firewalls. Nótese que el cliente también debe ser Spring, ya que el protocolo está implementado en librerías propias del framework (un cliente RMI podría ser no-Spring). Será apropiado cuando nos interese **buen rendimiento, clientes Spring y tengamos posibles problemas con los puertos permitidos.** .
- **Protocolos Hessian y Burlap:** son protocolos que funcionan a través del puerto HTTP estándar. Hessian es binario y Burlap XML, por lo que el primero es más eficiente. Teóricamente pueden funcionar con clientes no-Java, aunque con ciertas limitaciones. No son protocolos originalmente diseñados en el seno de Spring, sino de una empresa

llamada Caucho (aunque son también *open source*). Será interesante cuando queramos **buen rendimiento, clientes no-Java y tengamos posibles problemas con los puertos permitidos.**

- **Servicios web:** son la "tecnología dominante" en la actualidad para el acceso a componentes remotos en plataformas heterogéneas (con clientes escritos en casi cualquier lenguaje). Su punto débil es básicamente el rendimiento: la necesidad de transformar los mensajes que intercambian cliente servidor a formato neutro en XML hace que sean poco eficientes, pero es lo que al mismo tiempo los hace portables. Serán apropiados cuando **El rendimiento no sea crítico, y queramos la máxima portabilidad en cuanto a clientes.**

Nótese que en el acceso remoto a componentes los EJBs siguen teniendo ciertas ventajas sobre Spring, en particular la propagación remota de transacciones. Será una cuestión de prioridades la decisión de si esta característica es vital para nuestra aplicación o bien puede sacrificarse en aras de un mejor rendimiento y una mayor simplicidad.

Discutiremos a continuación con más detalle las características de las tres primeras alternativas (ya que la cuarta se verá en profundidad en el módulo de servicios web) y cómo usarlas y configurarlas dentro de Spring.

En todos los casos vamos a usar el siguiente ejemplo, muy sencillo, de componente al que deseamos acceder de forma remota, con su interfaz:

```
package servicios;

public interface ServicioSaludo {
   public String getSaludo();
}

package servicios;

public class ServicioSaludoImpl implements ServicioSaludo {
   String[] saludos = {"hola, ¿qué tal?", "me alegra verte", "yeeeeeey"};

   public String getSaludo() {
         int pos = (int)(Math.random() * saludos.length);
         return saludos[pos];
   }
}
```

1.2. RMI en Spring

Aunque el uso directo de RMI puede resultar tedioso, Spring ofrece una capa de abstracción sobre el RMI "puro" que permite acceder de forma sencilla y casi transparente a objetos remotos.

Usando la clase `RmiServiceExporter` podemos exponer la interfaz de nuestro servicio como un objeto RMI. Se puede acceder desde el cliente usando RMI "puro" o bien, de modo más sencillo con un `RmiProxyFactoryBean`.

La configuración en el lado del servidor quedará como sigue:

```
<bean id="miSaludador" class="servicios.ServicioSaludoImpl">
</bean>

<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
    <!-- En este caso el nombre del servicio y del bean son iguales, pero
no tiene por que -->
    <property name="serviceName" value="miSaludador"/>
    <property name="service" ref="miSaludador"/>
    <property name="serviceInterface" value="servicios.ServicioSaludo"/>
    <!-- El puerto por defecto es el 1099 -->
    <property name="registryPort" value="1199"/>
</bean>
```

En la configuración anterior se ha cambiado el puerto del servidor de nombres RMI para evitar posibles conflictos con el del servidor de aplicaciones. A partir de este momento, el objeto remoto es accesible a través de la URL `rmi://localhost:1199/miSaludador`.

La configuración en el cliente quedaría como sigue:

```
<bean id="saludadorRMI"
class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
        <property name="serviceUrl"
value="rmi://localhost:1199/miSaludador"/>
        <property name="serviceInterface"
value="servicios.ServicioSaludo"/>
</bean>
```

Y para acceder al objeto de forma remota todo lo que necesitaríamos es (teniendo accesible en el cliente el código de `ServicioSaludo`).

```
   ClassPathXmlApplicationContext contexto = new
ClassPathXmlApplicationContext("clienteRMI.xml");
   ServicioSaludo ss =  (ServicioSaludo) contexto.getBean("saludadorRMI");
   System.out.println(ss.getSaludo());
```

Suponiendo que el fichero de configuración en el cliente lo hemos llamado `clienteRMI.xml`

1.3. Hessian y Burlap

Hessian y Burlap son dos protocolos diseñados originalmente por la empresa Caucho, desarrolladora de un servidor de aplicaciones J2EE de código abierto llamado `Resin`. Ambos son protocolos para acceso a servicios remotos usando conexiones HTTP estándar. La diferencia básica entre ambos es que Hessian es binario (y por tanto más eficiente que Burlap) y este es XML (y por tanto las comunicaciones son más sencillas de depurar). Para ambos también se han desarrollado implementaciones en distintos lenguajes de manera que el cliente de nuestra aplicación podría estar escrito en C++, Python, C#, PHP u otros.

1.3.1. Uso de los protocolos

Usaremos Hessian en el siguiente ejemplo, aunque la configuración de Burlap es prácticamente idéntica. Hessian se comunica mediante HTTP con un servlet . Por tanto el primer paso será crear dicho servlet en nuestro `web.xml`. Nos apoyaremos en la clase `DispatcherServlet` propia de Spring, ya que se integra de manera automática con el resto de elementos de nuestra configuración. A continuación se muestra el fragmento significativo del `web.xml`.

```
<servlet>
        <servlet-name>remoting</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
        <servlet-name>remoting</servlet-name>
        <url-pattern>/remoting/*</url-pattern>
</servlet-mapping>
```

Esto hace accesible el servlet a través de la URL `http://localhost:8080/remoting` (si por ejemplo usamos Tomcat cuyo puerto por defecto es el 8080). En Spring, la configuración de cada `DispatcherServlet` se debe guardar en un xml con nombre *nombreDelServlet*-servlet.xml (en nuestro caso `remoting-servlet.xml`).

```
<bean id="miSaludador" class="servicios.ServicioSaludoImpl">
</bean>

<bean name="/saludador"
class="org.springframework.remoting.caucho.HessianServiceExporter">
  <property name="service" ref="miSaludador"/>
```

```
  <property name="serviceInterface" value="servicios.ServicioSaludo"/>
</bean>
```

Aquí hacemos uso de la clase `HessianServiceExporter`, que nos permite exportar de forma sencilla un servicio Hessian. En nuestro caso estará accesible en la URL `http://localhost:8080/contexto-web/remoting/saludador`. Nos falta la configuración del cliente y el código que llama al servicio:

```
<bean id="miSaludador"
class="org.springframework.remoting.caucho.HessianProxyFactoryBean">
        <property name="serviceUrl"
value="http://localhost:8080/contexto-web/remoting/saludador"/>
        <property name="serviceInterface"
value="servicios.ServicioSaludo"/>
</bean>
```

Suponiendo que el código XML anterior se guarda en un fichero llamado `clienteHessian.xml`

```
ClassPathXmlApplicationContext contexto = new
ClassPathXmlApplicationContext("clienteHessian.xml");
ServicioSaludo ss =  (ServicioSaludo) contexto.getBean("miSaludador");
System.out.println(ss.getSaludo());
```

En el ejemplo anterior bastaría con escribir `Burlap` allí donde aparece `Hessian` y todo debería funcionar igual, pero ahora usando este procolo basado en mensajes XML.

1.3.2. Autentificación HTTP con Hessian y Burlap

Al ser una conexión HTTP estándar podemos usar autentificación BASIC. De este modo podemos usar la seguridad declarativa del contenedor web también para controlar el acceso a componentes remotos. En la configuración del servidor podríamos añadir el siguiente código:

```
<bean
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
   <property name="interceptors">
      <list>
         <ref bean="authorizationInterceptor"/>
      </list>
   </property>
</bean>

<bean id="authorizationInterceptor"
class="org.springframework.web.servlet.handler.UserRoleAuthorizationInterceptor">
   <property name="authorizedRoles">
    <list>
      <value>admin</value>
      <value>subadmin</value>
```

```
    </list>
   </property>
</bean>
```

Usando AOP añadimos un interceptor que resulta ser de la clase `UserRoleAuthorizationInterceptor`. Dicho interceptor solo permite el acceso al bean si el usuario resulta estar en uno de los roles especificados en la propiedad `authorizedRoles`. El `BeanNameUrlHandlerMapping` es el objeto que "tras las bambalinas" se encarga de asociar los beans que comienzan con "/" con los servicios en la URL del mismo nombre (en nuestro caso el bean "/saludador").

1.4. HTTP invoker

Esta es una implementación propia de Spring, que utiliza la serialización estándar de Java para transmitir objetos a través de una conexión HTTP estándar. Será la opción a elegir cuando los objetos sean demasiado complejos para que funcionen los mecanismos de serialización de Hessian y Burlap.

La configuración es muy similar al apartado anterior, podemos usar el mismo `DispatcherServlet` pero ahora para el acceso al servicio se debe emplear la clase `HttpInvokerServiceExporter` en lugar de `HessianServiceExporter`

```
<bean id="miSaludador" class="servicios.ServicioSaludoImpl">
</bean>

<bean name="/saludadorHTTP"
class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
  <property name="service" ref="miSaludador"/>
  <property name="serviceInterface" value="servicios.ServicioSaludo"/>
</bean>



<bean id="httpProxy"
class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
  <property name="serviceUrl"
value="http://localhost:8080/remoting/saludadorHTTP"/>
  <property name="serviceInterface" value="servicios.ServicioSaludo"/>
</bean>
```

Por defecto en el cliente se usan las clases estándar de J2SE para abrir la conexión HTTP. Además Spring proporciona soporte para el `HttpClient` de Jakarta Commons. Bastaría con poner una propiedad adicional en el `HttpInvokerProxyFactoryBean`:

```
<bean id="httpProxy"
class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
  ...
  <property name="httpInvokerRequestExecutor">
    <bean
class="org.springframework.remoting.httpinvoker.CommonsHttpInvokerRequestExecutor"/>
  </property>
  ...
</bean>
```

2. Transacciones declarativas en Spring

Un elemento muy útil es la posibilidad de hacer nuestros objetos de negocio transaccionales. Spring permite hacer esto de forma declarativa de manera sencilla. Veremos aquí cómo hacerlo con anotaciones de Java 1.5. Si no podemos usar esta versión en nuestro proyecto, tendremos que hacerlo en el fichero XML de configuración de beans. La documentación de Spring explica cómo hacerlo, aquí por cuestiones de espacio nos restringiremos únicamente a las anotaciones.

Supongamos que tenemos una clase de negocio y deseamos hacerla transaccional. Basta con precederla de la anotación `@Transactional`

```
@Transactional
public class MiGestorPedidos implements GestorPedidos {
    Pedido getPedido(String id) {
            //...
    }
    void insertPedido(Pedido p) {
                //...
    }
}
```

Necesitamos también añadir algo de código en el XML de definición de beans, aunque no son demasiadas líneas:

```
<?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:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
       http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">
```

```
  <!-- objeto que queremos hacer transaccional -->
  <bean id="miGestorPedidos" class="x.y.service.MiGestorPedidos"/>

  <!-- habilitar transacciones basadas en anotaciones -->
  <tx:annotation-driven transaction-manager="txManager"/>

  <!-- Necesitamos un gestor de transacciones -->
  <bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!-- este gestor opera sobre una fuente de datos -->
    <property name="dataSource" ref="dataSource"/>
  </bean>

  <!-- resto del archivo -->

</beans>
```

La siguiente tabla resume las posibles propiedades de la anotación `@Transactional`:

Propiedad	Tipo	Significado
propagation	enum: Propagation	nivel de propagación (opcional)
isolation	enum: Isolation	nivel de aislamiento (opcional)
readOnly	boolean	solo de lectura vs. de lectura/escritura
timeOut	int (segundos)	
rollbackFor	array de objetos Throwable	clases de excepción que deben causar rollback
rollbackForClassName	array con nombres de objetos Throwable	nombres de clases de excepción que deben causar rollback
noRollbackFor	array de objetos Throwable	clases de excepción que no deben causar rollback
noRollbackForClassName	array con nombres de objetos Throwable	nombres de clases de excepción que no deben causar rollback