



Eusko Jaurlaritzaren Informatika Elkartea
Sociedad Informática del Gobierno Vasco

UDA - Utilidades de desarrollo de aplicaciones

Desarrollo de webservices en UDA

Fecha: 02/01/2012

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



UDA – Utilidades de desarrollo de aplicaciones by [EJIE](http://www.ejie.es) is licensed under a [Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 Unported License](https://creativecommons.org/licenses/by-nc-sa/3.0/).

Control de documentación

Título de documento: Web Services

Histórico de versiones

Código:	Versión:	Fecha:	Resumen de cambios:
	1.0.0	02/01/2012	Primera versión
	1.0.1	04/04/2012	Comentado añadir ruta de certificación en cliente de Weblogic 11
	1.0.2	29/09/2015	Creación web service REST y publicación en ERPI

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

	Capítulo/sección	Página
1	Introducción	1
2	Creación y publicación de un Web Service en Spring	2
2.1	Creación y publicación de un Web Service SOAP en Spring	2
2.2	Creación y publicación de un Web Service REST en Spring	7
3	Creación de un cliente de Web Service SOAP	9
3.1	Generación de los recursos necesarios	9
3.2	Acceso a las propiedades dependientes de entorno	11
3.3	Implementación del cliente mediante Spring	12
4	Seguridad en los Web Services SOAP	15
4.1	Niveles de seguridad	15
4.2	Transformaciones de seguridad	15
5	Seguridad con XLNets servicio SOAP	17
5.1	Handler de servidor	18
5.2	Handler de cliente	21
6	Clientes de servicios expuestos con Ws-Security	24
6.1	Políticas de seguridad aplicadas en el OSB	24
6.2	Cliente Weblogic 11	25
6.3	Cliente wss4j	31
7	Referencias externas	35

1 Introducción

El presente documento tiene los siguientes objetivos:

- Exponer un Web Service integrado con Spring.
- Creación de un cliente de Web Service mediante Spring.
- Mostrar cómo realizar la securización de los Web Services.

El contenido del documento se basa en la Normativa de Desarrollo de Servicios para ERPI. Los ejemplos aquí expuestos se ajustan a los requerimientos indicados por dicha normativa.

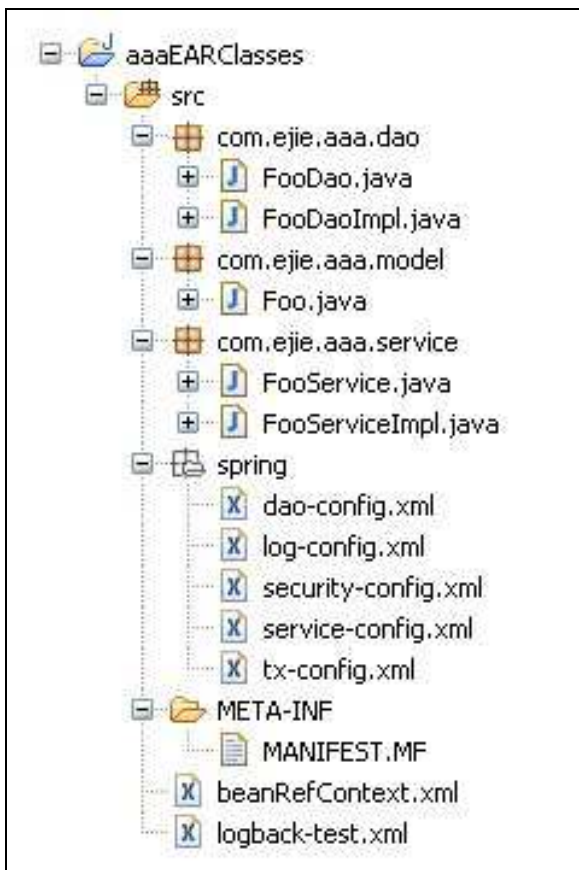
2 Creación y publicación de un Web Service en Spring

2.1 Creación y publicación de un Web Service SOAP en Spring

Para la mejor comprensión del proceso de creación y publicación de un Web Service, se va a mostrar un ejemplo explicando cada uno de los pasos que se deben de llevar a cabo.

Como punto de partida crearemos una aplicación mediante el plugin de UDA y su correspondiente código de negocio a partir de una tabla de base de datos con nombre FOO. El servicio generado permite realizar las operaciones CRUD sobre su correspondiente tabla de base de datos.

Esta sería la estructura de ficheros generada:



El objetivo es la creación de un Web Service que exponga los métodos CRUD generados en el servicio FooService. Estos métodos nos permitirán realizar las siguientes operaciones.

- Inserción
- Búsqueda
- Modificación
- Eliminación

El código de la interface del servicio generado mediante UDA es el siguiente:

```
package com.ejie.aaa.service;

import com.ejie.aaa.model.Foo;

public interface FooService {

    Foo add(Foo foo);

    Foo update(Foo foo);

    Foo find(Foo foo);

    void remove(Foo foo);

}
```

Y a continuación, tenemos su implementación:

```
package com.ejie.aaa.service;

import com.ejie.aaa.dao.FooDao;
import com.ejie.aaa.model.Foo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service(value = "fooService")
public class FooServiceImpl implements FooService {

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

    @Autowired
    private FooDao fooDao;

    @Transactional(rollbackFor = Throwable.class)
    public Foo add(Foo foo) {
        return this.fooDao.add(foo);
    }

    @Transactional(rollbackFor = Throwable.class)
    public Foo update(Foo foo) {
        return this.fooDao.update(foo);
    }

    public Foo find(Foo foo) {
        return (Foo) this.fooDao.find(foo);
    }

    @Transactional(rollbackFor = Throwable.class)
    public void remove(Foo foo) {
```

```
        this.fooDao.remove(foo);  
    }  
}
```

El siguiente paso es crear el Web Service que haga uso de este servicio de Spring. Para ello se deberá crear una clase java y utilizar las anotaciones necesarias. Un ejemplo de dicha clase es el siguiente:

```
package com.ejie.aaa.webservice.foo webservice;  
  
import javax.jws.WebMethod;  
import javax.jws.WebService;  
import javax.jws.soap.SOAPBinding;  
import javax.jws.soap.SOAPBinding.ParameterStyle;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.context.support.SpringBeanAutowiringSupport;  
  
import com.ejie.aaa.model.Foo;  
import com.ejie.aaa.service.FooService;  
  
@WebService(serviceName = "fooWebService", portName = "fooWebServicePort", targetNamespace  
= "http://com.ejie.aaa.webservice")  
@SOAPBinding(parameterStyle = ParameterStyle.WRAPPED)  
@HandlerChain(file = "server-handlers.xml")  
public class FooWebServiceImpl extends SpringBeanAutowiringSupport {  
  
    @Autowired  
    private FooService fooService;  
  
    @WebMethod  
    public Foo add(Foo foo) {  
        return this.fooService.add(foo);  
    }  
  
    @WebMethod  
    public Foo update(Foo foo) {  
        return this.fooService.update(foo);  
    }  
  
    @WebMethod  
    public Foo find(Foo foo) {  
        return this.fooService.find(foo);  
    }  
  
    @WebMethod  
    public void remove(Foo foo) {  
        this.fooService.remove(foo);  
    }  
}
```

Las anotaciones que se indican son las siguientes:

@WebService. Tiene como atributos el nombre del Web Service y el nombre del namespace del Web Service.

@SOAPBinding. Permite indicar la manera de aportar los parámetros de entrada y cómo se recibirá el parámetro de salida.

@Autowired. Anotación que permite asociar un objeto java con el servicio de Spring creado.

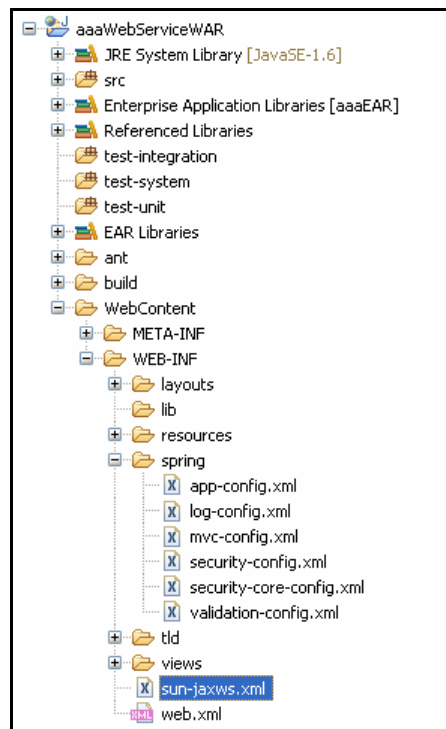
@WebMethod. Los métodos marcados con esta anotación se traducen en el descriptor WSDL como operaciones dentro del binding.

A continuación, se debe configurar el fichero web.xml para poder publicar el Web Service correctamente. Para ello deberemos añadir un nuevo servlet al fichero web.xml.

```
<servlet>
  <servlet-name>servicioWeb</servlet-name>
  <servlet-class>
    com.sun.xml.ws.transport.http.servlet.WSServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>servicioWeb</servlet-name>
  <url-pattern>/servicioWeb</url-pattern>
</servlet-mapping>
```

Por último, para referenciar el Web Service que hemos creado dentro de nuestro proyecto, deberemos crear un nuevo fichero de configuración que indicará dónde se encuentra la implementación del Web Service, para que este Servlet pueda encontrar el Web Service.

El fichero se deberá llamar *sun-jaxws.xml*, y deberá localizarse en el directorio *WebContent/WEB-INF* del proyecto War:



En este xml se indicarán todos los “endpoints” o entradas a los servicios web definidos en el fichero web.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/jax-ws/ri/runtime
http://java.sun.com/webservices/docs/2.0/jaxws/sun-jaxws.xsd"
  version='2.0'>

  <endpoint name="fooWebService"
    implementation="com.ejje.aaa.webservice.fooWebService.FooWebServiceImpl"
    url-pattern="/fooWebService" />

</endpoints>
```

En este caso solo existe un único endpoint. A este endpoint se le indica su nombre, la url a la que corresponde y la implementación propiamente dicha del servicio web.

Una vez realizados estos pasos se deberá volver a desplegar la aplicación en el servidor. Accediendo a la consola de Weblogic podremos observar si la aplicación ha desplegado correctamente el Servicio Web.

Despliegues

<div> Instalar Actualizar Suprimir Iniciar Parar </div> <div>Mostrando 1 a 8 de 8 Anterior Siguiente</div>					
<input type="checkbox"/>	Nombre	Estado	Estado	Tipo	Orden de Despliegue
<input type="checkbox"/>	aaaEAR	Activo	OK	Aplicación de Empresa	100
	Módulos				
	aaaWebServiceWar			Aplicación Web	
	EJB				
	Ninguno Que Mostrar				
	Servicios Web				
	fooWebService			Servicio Web	

Si accedemos al Servicio Web de la aplicación en la consola de Weblogic podremos

- acceder al WSDL del propio webservice,
- comprobar su funcionamiento, ya que la propia consola es capaz de generar un cliente.

Valores para fooWebService

Visión General
Configuración
Seguridad
Prueba
Supervisión

Utilice esta página para probar que el servicio web está desplegado y funciona según lo esperado. En la tabla, amplíe el nombre del servicio web para ver una lista de los puntos de prueba. Haga clic en **?WSDL** para ver el WSDL dinámico en una ventana de explorador independiente. Haga clic en **Probar Cliente** para abrir una nueva ventana de explorador en la que puede probar cada operación de forma individual introduciendo los valores de los parámetros, ejecutando las operaciones y viendo los resultados.

Pruebas de Despliegue

Nombre	Punto de Prueba	Comentarios
fooWebService		Test points for this Webservice module.
/aaaWebServiceWar/fooWebService	?WSDL	Página WSDL en el servidor AdminServer
/aaaWebServiceWar/fooWebService	Test client	Probar Cliente en Servidor AdminServer

2.2 Creación y publicación de un Web Service REST en Spring

UDA genera a partir de un modelo de datos de manera automática las operaciones CRUD (clases DAO y los controlers) de una entidad de datos. Lo único que habría que hacer es exponer dicho servicio REST en ERPI mediante la plantilla específica de *Plantilla Solicitud Exposicion Servicios de tipo REST en ERPI*.

Se deberá rellenar la URL raíz del controller creado (en este ejemplo usuarios)

URL	URL's de acceso al servicio (al menos una obligatoria).	Obligatorio
	<p><code>http://ejds56.ejiedes.net/AAAWar/usuarios</code></p> <p><code>http://ejds46.ejiedes.net/AAAWar/usuarios</code></p>	

Es importante exponer todos los métodos HTTP para que las operaciones CRUD se puedan invocar.

Tipo de Servicio REST	Tipo de servicio a exponer (Web Service o REST).	Obligatorio
<input checked="" type="checkbox"/> REST-POST (por ejemplo, servlet con método POST) <input checked="" type="checkbox"/> REST-GET (por ejemplo, servlet con método GET) <input checked="" type="checkbox"/> REST-DELETE (por ejemplo, servlet con método DELETE) <input checked="" type="checkbox"/> REST-PUT (por ejemplo, servlet con método PUT)		

3 Creación de un cliente de Web Service SOAP

En este apartado se va a exponer la creación de un cliente de Web Service. Siguiendo con el ejemplo, se implementará un cliente que realice invocaciones al Web Service creado en el apartado anterior.

3.1 Generación de los recursos necesarios

La implementación del cliente se realiza a partir del descriptor WSDL del servicio. Para facilitar el desarrollo, se utilizará una tarea ant propia de Weblogic (clientgen) que generará las clases necesarias para la creación del cliente y que estará incluida en el build.xml que UDA genera por defecto dentro del directorio xxxEAR.

Este sería el fichero build.xml con la tarea de generación del cliente incluida (obviando el resto de tareas existentes):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE project>
<project name="xxxEAR" default="mavenRunDependencies"
xmlns:artifact="antlib:org.apache.maven.artifact.ant">

    <!-- Permite el uso de variables de entorno -->
    <property environment="env" />

    <!-- OBTENER DEPENDENCIAS -->

    <!-- PORTALIZAR ESTILOS -->

    <!-- GENERACIÓN DE CLIENTE WS -->
    <property name="bea_home" value="C:\bea\weblogic1035" />

    <!-- Informacion de la aplicacion -->
    <property name="source.dir" value="..\${codApp}EARClasses/src" />

    <!-- Informacion del cliente generado -->
    <property name="client.package.name" value="com.ejie.bbb.webservice.foo.webservice"/>
    <property name="client.wsdl"
value="http://desarrollo.jakina.ejiedes.net:7001/aaaWebServiceWar/fooWebService?WSDL"/>

    <!-- variables de ejecucion -->
    <property name="temp.dir" value="${source.dir}/../output" />

    <!-- path de compilacion/generacion de clases -->
    <path id="custom.class.path" refid="wlll.class.path">
    <!-- incluir librerias adicionales necesarias para compilar -->
    </path>

    <!-- classpath entorno Oracle WebLogic Server 11gR1 -->
    <path id="wlll.class.path">
    <pathelement path="${java.class.path}" />
    <pathelement path="${bea_home}/jrockit_160_24_D1.1.2-4/lib/tools.jar" />
    <pathelement path="${bea_home}/utils/config/10.3/config-launch.jar" />
    <pathelement path="${bea_home}/wlserver_10.3/server/lib/weblogic_sp.jar" />
    <pathelement path="${bea_home}/wlserver_10.3/server/lib/weblogic.jar" />
    <pathelement path="${bea_home}/modules/features/weblogic.server.modules_10.3.5.0.jar" />
    <pathelement path="${bea_home}/modules/features/weblogic.server.modules.extra_10.3.5.0.jar"/>
    <pathelement path="${bea_home}/wlserver_10.3/server/lib/webservices.jar" />
    <pathelement path="${bea_home}/modules/org.apache.ant_1.7.1/lib/ant-all.jar" />
    <pathelement path="${bea_home}/modules/net.sf.antcontrib_1.0.0.0_1-0b2/lib/ant-contrib.jar"/>
    </path>
```

```
<!-- definicion de tareas ant de weblogic -->
<taskdef name="wsdlc" classname="weblogic.wsee.tools.anttasks.WsdlcTask"
classpathref="wlll.class.path" />
<taskdef name="jwsc" classname="weblogic.wsee.tools.anttasks.JwscTask"
classpathref="wlll.class.path" />
<taskdef name="clientgen" classname="weblogic.wsee.tools.anttasks.ClientGenTask"
classpathref="wlll.class.path" />

<!-- Limpieza -->
<target name="clean-build-client">
  <delete dir="${temp.dir}" failonerror="false" />
</target>

<!-- Generación de recursos -->
<target name="build-client" depends="clean-build-client">
  <clientgen type="JAXWS" wsdl="${client.wsdl}" destDir="${temp.dir}"
classpathref="custom.class.path" copywsdl="true" packageName="${client.package.name}" />
</target>

<!-- Copia de recursos -->
<target name="copy-generated-sources" depends="build-client">
  <!-- copiar fuentes java generados-->
  <copy todir="${source.dir}" flatten="false" overwrite="true">
    <fileset dir="${temp.dir}">
      <include name="**/*.java" />
    </fileset>
  </copy>

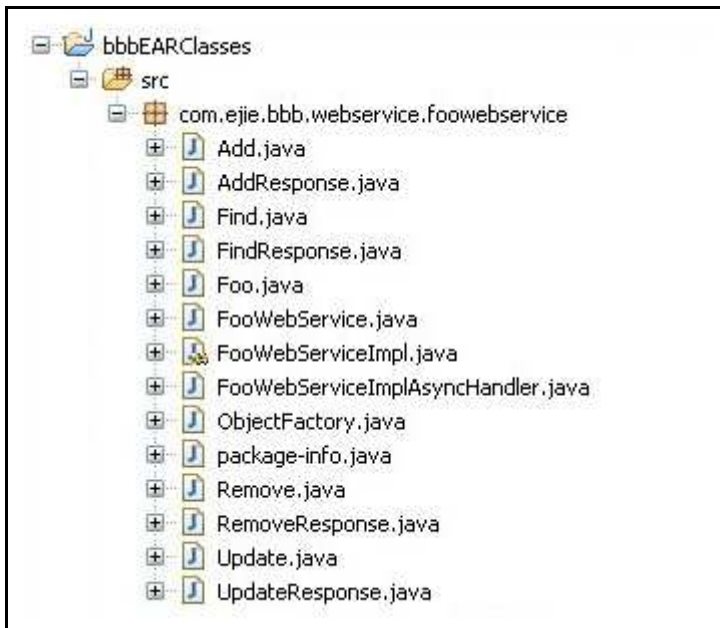
  <!-- copiar WSDL, XSD -->
  <copy todir="${source.dir}" flatten="false" overwrite="true">
    <fileset dir="${temp.dir}">
      <include name="**/*.xml" />
      <include name="**/*.xsd" />
      <include name="**/*.wsdl" />
    </fileset>
  </copy>

  <!-- Eliminar ficheros temporales -->
  <delete dir="${temp.dir}" />
</target>
<target name="generateClientSources" depends="copy-generated-sources">
  <echo message="build-service - generacion de clases del cliente" />
</target>
</project>
```

Dicho script será configurable mediante las siguientes propiedades:

- **client.package.name:** Especifica el paquete en el que se van a generar las clases y recursos necesarios.
- **client.wsdl:** Ruta del wsdl del servicio web que se desea invocar y sobre el que se generarán las clases y recursos.

Una vez ejecutada, la tarea ant nos generará, en el paquete indicado dentro del xxxEARClasses de la aplicación, los siguientes recursos:



3.2 Acceso a las propiedades dependientes de entorno

Durante la creación de los clientes de Web Service, existen determinadas propiedades (p.e. la url del WSDL) que se deben especificar como parámetro de configuración de la aplicación. Puesto que se trata de valores dependientes del entorno, el sitio adecuado para su declaración es el fichero de configuración de la aplicación; `bbb.properties` en el ejemplo.

A continuación se mostrará un ejemplo sobre cómo acceder a los valores de dichas propiedades desde los ficheros de configuración de Spring (en el resto del documento se hará de la misma forma).

En el fichero de configuración de propiedades dependientes del entorno se incluirán las declaraciones:

```
webservice.fooWebService.wsdl=http://desarrollo.jakina.ejiedes.net:7001/aaaWebServiceWar/fo
oWebService?WSDL
webservice.fooWebService.namespace=http://com.ejie.aaa.webservice
```

Para acceder a estas propiedades, será necesario indicar cuál es el fichero que las contiene

```
<ctx:property-placeholder location="classpath:/bbb/bbb.properties" />
```

Y luego acceder a las mismas para cargar el valor correspondiente:

```
<property name="wsdlDocumentUrl" value="${webservice.fooWebService.wsdl}" />
```

A continuación se muestra un fichero de configuración de Spring donde se ha incluido el acceso a las propiedades declaradas

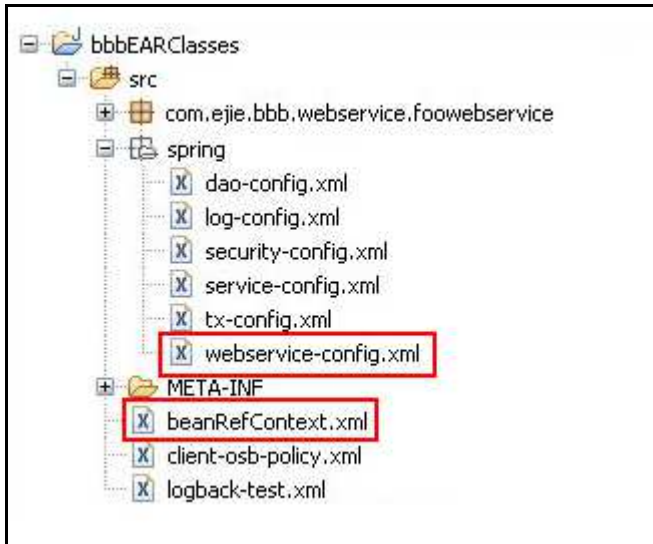
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:ctx="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-3.0.xsd
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <ctx:property-placeholder location="classpath:/bbb/bbb.properties" />

    <!-- WebService con token de seguridad de N38 -->
    <bean id="fooWebService"
        class="org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean">
        <property name="wsdlDocumentUrl" value="{webService.fooWebService.wsdl}" />
        <property name="serviceInterface"
        value="com.ejie.bbb.webservice.fooWebService.FooWebServiceImpl" />
        <property name="serviceName" value="fooWebService" />
        <property name="portName" value="fooWebServicePort" />
        <property name="namespaceUri" value="{webService.fooWebService.namespace}" />
        <property name="lookupServiceOnStartup" value="false" />
        <property name="handlerResolver" ref="jaxWsHandlerResolver" />
    </bean>
</beans>
```

3.3 Implementación del cliente mediante Spring

Una vez generados los recursos necesarios para la creación del cliente, se deberá configurar Spring convenientemente.



La configuración de los clientes de Web Services se realizará en el fichero de configuración de Spring *webservice-config.xml*. Por lo que se deberá modificar el fichero *beanRefContext.xml* para que dicho fichero sea cargado al inicio junto con el resto de la configuración de Spring, del siguiente modo:

```
<?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-3.0.xsd">

    <!-- contents of beanRefContext.xml: notice that the bean id is the value
         specified by the parentContextKey param -->
    <bean id="ear.context"
         class="org.springframework.context.support.ClassPathXmlApplicationContext">
        <constructor-arg>
            <list>
                <value>spring/dao-config.xml</value>
                <value>spring/log-config.xml</value>
                <value>spring/service-config.xml</value>
                <value>spring/security-config.xml</value>
                <value>spring/tx-config.xml</value>
                <value>spring/webservice-config.xml</value>
            </list>
        </constructor-arg>
    </bean>
</beans>
```

En el fichero webservice-config.xml deberemos declarar el bean de Spring que se utilizará para acceder al Web Service. Para el ejemplo que nos ocupa, la declaración sería la siguiente:

```
<?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-3.0.xsd">

    <ctx:property-placeholder location="classpath:/bbb/bbb.properties" />

    <bean id="fooWebService"
class="org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean">
        <property name="wsdlDocumentUrl" value="{webservice.fooWebService.wsdl}"/>
        <property name="serviceInterface"
value="com.ejie.bbb.webservice.fooWebService.FooWebServiceImpl"/>
        <property name="serviceName" value="fooWebService"/>
        <property name="portName" value="fooWebServicePort"/>
        <property name="namespaceUri"
value="{webservice.fooWebService.namespace}"/>
        <property name="lookupServiceOnStartup" value="false"/>
    </bean>
</beans>
```

Las propiedades que deberemos indicar serán las mostradas a continuación:

- **wsdlDocumentUrl**: URL del WSDL donde está expuesto el Web Service.
- **serviceInterface**: Implementación del Web Service que ha sido generada por script ant utilizado para la generación de los recursos necesarios para la creación del cliente.
- **serviceName**: Nombre del Web Service presente en el WSDL.
- **portName**: Nombre del port presente en el WSDL.
- **namespaceUri**: URI del namespace presente en el WSDL.

Como se ha comentado anteriormente, los valores de las propiedades dependientes de entorno se almacenarán en el fichero `bbb.properties`. Para este ejemplo, serían las siguientes:

```
webservice.fooWebService.wsdl=http://desarrollo.jakina.ejiedes.net:7001/aaaWebServiceWar/fo
oWebService?WSDL
webservice.fooWebService.namespace=http://com.ejie.aaa.webservice
```

Por último, como ejemplo de llamada al cliente de Web Service, crearemos un controller de Spring que a partir de un identificador (pasado como parámetro) devuelva los datos del registro correspondiente al mismo tras invocar al Web Service.

La implementación del controller sería la siguiente:

```
package com.ejie.bbb.control;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;

import com.ejie.bbb.remoting.jaxws.wlgen.Foo;
import com.ejie.bbb.remoting.jaxws.wlgen.FooWebServiceImpl;

@Controller
@RequestMapping(value = "/fooWebService")
public class WebServiceController {

    @Autowired
    private FooWebServiceImpl fooWebService;

    @RequestMapping(value =("/{bar1})", method = RequestMethod.GET)
    public @ResponseBody
    Foo getUsuario(@PathVariable String bar1) {

        Foo foo = new Foo();

        foo.setBar1(bar1);

        return this.fooWebService.find(foo);
    }
}
```

Para realizar un test, se deberá de realizar una petición a la siguiente URL:

`http://desarrollo.jakina.ejiedes.net:7001/bbbWebServiceWar/fooWebService/1`

Con lo que el controller, mediante el método `find` del Web Service, realizará una búsqueda del elemento correspondiente al identificador pasado como parámetro, en este caso el "1".

La respuesta de dicho controller sería:

```
{"bar1":"1","bar2":"aaa","bar3":"AAA"}
```

4 Seguridad en los Web Services SOAP

En este apartado se pretende explicar cómo implementar las diferentes configuraciones de seguridad que se pueden aplicar a los Web Services expuestos. Para ello se ha tomado como referencia el documento de la normativa de ERPI para desarrollo de servicios.

Como norma general, la plataforma de seguridad no ejerce funciones de intermediario en la seguridad de los servicios. Sin embargo, es posible transformar la seguridad en los servicios expuestos a nivel de transporte y a nivel de mensaje, como se explica a continuación.

4.1 Niveles de seguridad

Seguridad a nivel de transporte

Cómo norma general, todas las comunicaciones entre clientes y servicios internos (ubicados en los CPD's de EJIE) se realizarán sobre HTTP.

Sin embargo, para los servicios internos expuestos en contextos externos, el OSB es capaz de:

- Exponer el servicio sobre HTTPS.
- Exponer el servicio sobre HTTPS con autenticación de cliente.

En el segundo caso, el cliente deberá disponer de un certificado X.509 válido expedido por **Izenpe** para invocar al servicio.

Seguridad a nivel de mensaje

La plataforma puede añadir políticas de seguridad WS-Security (*X.509 token de autenticación y firma*) a los servicios expuestos, sin necesidad de realizar ninguna modificación en el servicio destino.

En el caso de utilizar *Username Token Profile*, se ha comprobado que debido a estar basado en cabeceras, el bus no presenta problemas a la hora de traspasarlas desde la petición al servicio de destino.

El cliente deberá ser capaz de firmar el mensaje e incluir en la petición un certificado X.509 válido expedido por **Izenpe**.

4.2 Transformaciones de seguridad

XLNets a WS-Security

Para los servicios externos que requieran seguridad a nivel de mensaje con WS-Security existen dos opciones:

- Se deja al cliente interno la gestión de la seguridad, con lo que el OSB hace de mero intermediario.

- Se firma el mensaje con un certificado genérico asociado al OSB (que deberá ser aceptado por el servicio externo como válido) de manera que cualquier cliente interno podrá invocar en claro al servicio externo. Todas las llamadas al exterior se identificarán con el mismo certificado del OSB.

En este segundo caso, se añadirá al proxy service del servicio externo un Service Key Provider que deberá firmar el mensaje con la clave privada genérica almacenado en el servidor.

HTTPs a XLNets

Para los clientes externos que invoquen servicios internos con seguridad a nivel de transporte, el proxy service expuesto se configurará para validar la identidad del cliente a partir del certificado X.509 presentado por el cliente. Esto desencadenará el proceso de autenticación expuesto previamente, almacenando en el "Subject" asociado a la petición el token de sesión correspondiente.

En la pipeline de dicho proxy service se añadirá una acción que recupere el token de sesión del "Subject" y lo añada como una cabecera SOAP al mensaje entrante, de manera que la seguridad pueda ser validada por el servicio destino.

WS-Security a XLNets

Para los clientes externos que invoquen servicios internos con seguridad a nivel de mensaje, el proxy service expuesto se configurará para incluir políticas de seguridad (en principio las predefinidas en el OSB Auth.xml y Sign.xml). Esto requerirá al cliente que incluya las cabeceras WS-Security de firma y su certificado X.509 en el mensaje. En la invocación se desencadenará el proceso de autenticación expuesto previamente almacenando en el "Subject" asociado a la petición el token de sesión correspondiente.

En la pipeline de dicho proxy service se añadirá una acción que recupere el token de sesión del "Subject" y lo añada como una cabecera SOAP al mensaje entrante, de manera que la seguridad pueda ser validada por el servicio destino.

5 Seguridad con XLNets servicio SOAP

En este apartado se pretende explicar cómo implementar una seguridad basada en XLNets en el ejemplo que se ha seguido hasta ahora.

ERPI solicita que se incluya en la cabecera de la petición el token de seguridad de XLNets. Nuestro servicio deberá ser capaz de obtener esta sesión y asegurar que es una sesión válida de XLNets.

Supongamos que queremos realizar los siguientes pasos relativos a la seguridad:

1. El cliente crea una sesión de aplicación.
2. El cliente incluye su sesión de aplicación en la petición al Web Service.
3. El servicio recibe la petición y recupera la sesión del cliente.
4. El servicio verifica que la sesión del cliente es válida (**Autenticación**).
5. El servicio verifica que el cliente realmente tiene derecho a invocar el servicio (**Autorización**).

El primer problema que nos encontramos es cómo incluir la sesión del cliente en la petición. La solución obvia consiste en añadir a los mensajes definidos en el descriptor WSDL, un parámetro en el que se emplace la sesión XLNets. Sin embargo, resulta una mala práctica mezclar la lógica de negocio con la solución de seguridad, y perjudica la mantenibilidad del servicio.

El protocolo SOAP ofrece alternativas para estas situaciones. El siguiente fragmento muestra la estructura de un mensaje SOAP.

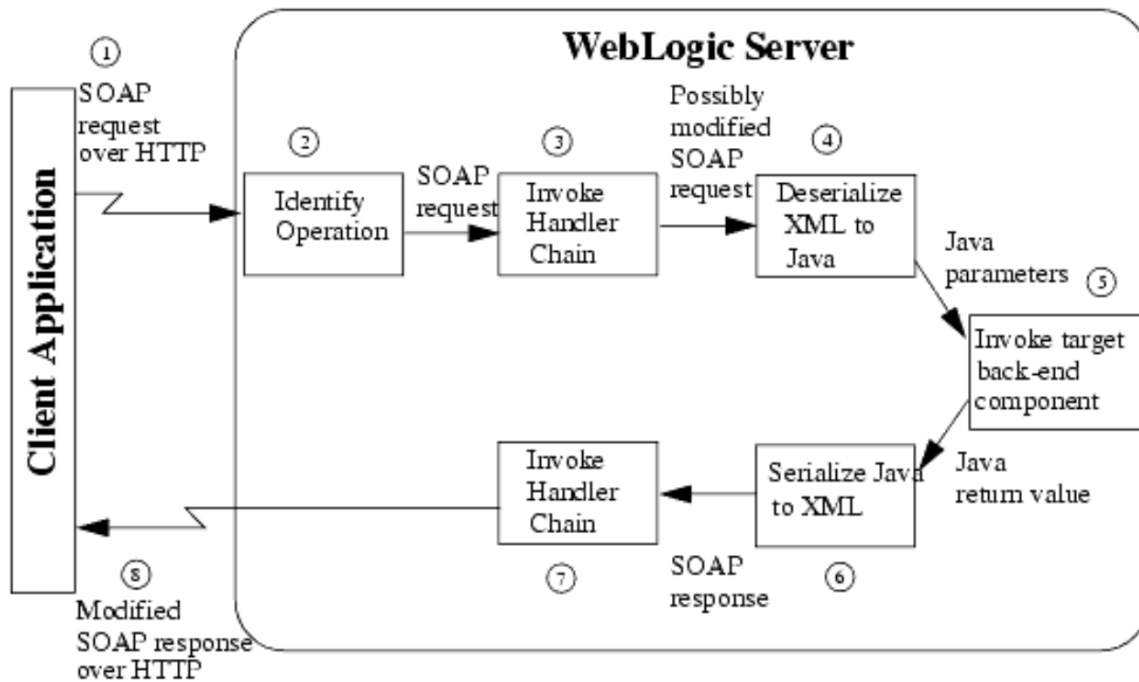
```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <!-- cabeceras (XML bien formado, sin restricciones) -->
    <n38>
      <elementos tipo="n38APISesionCrearToken">
        <elemento subtipo="N38Sesion">
          ...
        </elemento>
      </elementos>
    </n38>
  </env:Header>
  <env:Body>
    <!-- contenido XML definido en el WSDL -->
    ...
  </env:Body>
</env:Envelope>
```

Como se puede apreciar, existe una sección “Header” en la que se pueden incluir fragmentos de XML, con el objeto de tratar aspectos transversales independientes de la lógica de negocio, como por ejemplo, la seguridad. El descriptor WSDL no contiene ninguna especificación acerca de estas cabeceras, ya que sólo determina el contenido de la sección “Body”.

Por tanto, se recomienda que el cliente incluya su sesión de aplicación como una cabecera SOAP, y el servicio la recupere para realizar las validaciones de seguridad.

Técnicamente, para tratar la cabecera SOAP, la tecnología **JAX-WS** hace uso del concepto de **Handler** (traducido por manejador, filtro o interceptor), que se trata de una clase independiente capaz de procesar el mensaje entrante y/o saliente que determina si se debe continuar el procesamiento. Los Handlers pueden tener muchas aplicaciones, como generación de trazas, medidas de rendimiento y tal y como se muestra en este apartado, gestionar la seguridad.

La siguiente figura ilustra el proceso completo de una petición SOAP:



Por tanto, en un Handler tenemos la posibilidad de:

- Acceder al contenido XML en bruto de la petición SOAP antes de que se invoque el componente que implementa la lógica de negocio.
- Modificar el contenido de la petición.
- Detener el procesamiento de la petición SOAP con un error (**SOAP Fault**).
- Acceder al contenido XML en bruto de la respuesta SOAP.

Tanto para el cliente como para el servicio se puede definir una cadena de Handlers.

La solución para el Handler del servicio debe validar la sesión de aplicación recibida en la petición.

5.1 Handler de servidor

A continuación se va a describir como realizar la implementación y uso del handler de servidor para verificar si el cliente dispone de la autorización necesaria para realizar la invocación del servicio web.

Un ejemplo de la implementación de un handler de servidor sería la siguiente:

```

package com.ejie.aaa.remoting.jaxws;

import java.io.StringWriter;
import java.util.HashSet;
import java.util.Set;
import javax.xml.namespace.QName;
import javax.xml.transform.Result;
import javax.xml.transform.Source;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;

```

```
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;
import javax.xml.xpath.XPath;

import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathExpressionException;
import javax.xml.xpath.XPathFactory;
import org.w3c.dom.Node;

public class N38TokenServerHandler implements SOAPHandler<SOAPMessageContext> {
    private String tokenN38XQuery = "//n38";
    private XPathExpression expression;
    private TransformerFactory transformerFactory;

    /**
     * Constructor por defecto.
     *
     * @throws XPathExpressionException
     */
    public N38TokenServerHandler() throws XPathExpressionException {
        // inicializacion de la expresión XPATH
        XPathFactory xpFactory = XPathFactory.newInstance();
        XPath xpath = xpFactory.newXPath();
        this.expression = xpath.compile(tokenN38XQuery);
        this.transformerFactory = TransformerFactory.newInstance();
    }

    public Set<QName> getHeaders() {
        return new HashSet<QName>();
    }

    public void close(MessageContext context) {
        // Nada que hacer
    }

    /**
     * @see javax.xml.ws.handler.Handler#handleFault(javax.xml.ws.handler.MessageContext)
     */
    public boolean handleFault(SOAPMessageContext context) {
        return true;
    }

    /**
     * @see javax.xml.ws.handler.Handler#handleMessage(javax.xml.ws.handler.MessageContext)
     */
    public boolean handleMessage(SOAPMessageContext context) {
        boolean valid = true;
        Boolean outbound = (Boolean) context
            .get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
        // solo para los mensajes entrantes
        if (outbound != null && !outbound.booleanValue()) {
            valid = false;
            try {
                String tokenN38 = doExtractToken(context.getMessage()
                    .getSOAPHeader());
                if (tokenN38 != null && !tokenN38.equals("")) {
                    // TODO verificar que la sesion es valida
                    // TODO verificar los permisos del cliente
                    // En el ejemplo unicamente comprobamos que se ha enviado
                    // una sesion de aplicacion
                    valid = true;
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        return valid;
    }
}
```

```
/**
 * Extrae el xml correspondiente a la sesión de aplicación.
 *
 * @param node
 * @return
 * @throws XPathExpressionException
 * @throws TransformerException
 */
private String doExtractToken(Node node) throws XPathExpressionException,
    TransformerException {
    String tokenN38 = "";
    // búsqueda XPath del nodo n38
    Node tokenN38Node = (Node) expression.evaluate(node,
        XPathConstants.NODE);
    if (tokenN38Node != null) {
        // convertimos a String el nodo del token
        Source tokenN38NodeSource = new DOMSource(tokenN38Node);
        StringWriter writer = new StringWriter();
        Result result = new StreamResult(writer);
        transformerFactory.newTransformer().transform(tokenN38NodeSource,
            result);
        tokenN38 = writer.toString();
    }
    return tokenN38;
}
}
```

El Handler realiza una búsqueda XPath en el nodo de la cabecera SOAP para encontrar la sesión de aplicación. Si la encuentra, permite que se continúe procesando la petición. Una aplicación real debería además verificar que la sesión es válida y que la aplicación cliente tiene los permisos oportunos utilizando el API de XLNets.

Para asociar el handler al servicio nos creamos un fichero XML llamado “server-handlers.xml” con el siguiente contenido:

```
<?xml version="1.0" encoding="UTF-8"?>
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
    <handler-chain>
        <handler>
            <handler-
class>com.ejje.aaa.remoting.jaxws.N38TokenServerHandler</handler-class>
            </handler>
        </handler-chain>
    </handler-chains>
```

Y lo situamos en la misma carpeta que la clase

“com.ejje.aaa.webservice.fooWebService.FooWebServiceImpl” (la clase que tiene la anotación “@WebService”). Editamos esta clase para añadirle la anotación “@HandlerChain”:

```
package com.ejje.aaa.webservice.fooWebService;

import javax.xml.ws.HandlerChain;

@WebService(serviceName = "fooWebService", portName = "fooWebServicePort", targetNamespace
    = "http://com.ejje.aaa.webservice")
@SOAPBinding(parameterStyle = ParameterStyle.WRAPPED)
@HandlerChain(file = "server-handlers.xml")
public class FooWebServiceImpl extends SpringBeanAutowiringSupport {...}
```

El parámetro “file” de la anotación “@HandlerChain” indica la ruta relativa donde se encuentra el fichero descriptor de la cadena de Handlers.

Ahora el servicio está listo para su despliegue. De nuevo, un mismo Handler o cadena de Handlers puede ser reutilizado entre distintos servicios e incluso entre clientes y servicios.

5.2 Handler de cliente

Por su parte, el cliente deberá de incluir su sesión de aplicación como cabecera SOAP en la petición al servicio web expuesto.

Este sería un ejemplo de implementación del handler de cliente:

```
package com.ejie.bbb.webservice.util;

import java.util.HashSet;
import java.util.Set;
import javax.xml.namespace.QName;
import javax.xml.soap.SOAPHeader;
import javax.xml.transform.Result;
import javax.xml.transform.Source;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMResult;
import javax.xml.transform.dom.DOMSource;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;

import n38a.exe.N38APISesion;
import org.w3c.dom.Document;

import com.ejie.x38.util.StaticsContainer;

public class N38TokenClientHandler implements SOAPHandler<SOAPMessageContext> {
    private TransformerFactory transformerFactory;

    /**
     * Constructor por defecto
     */
    public N38TokenClientHandler() {
        this.transformerFactory = TransformerFactory.newInstance();
    }

    public Set<QName> getHeaders() {
        return new HashSet<QName>();
    }

    public void close(MessageContext context) {
        // Nada que hacer
    }

    public boolean handleFault(SOAPMessageContext context) {
        return true;
    }

    public boolean handleMessage(SOAPMessageContext context) {
        boolean valid = true;
        Boolean outbound = (Boolean) context
            .get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
        // solo para los mensajes SALIENTES
        if (outbound != null && outbound.booleanValue()) {
            valid = false;
            try {
```



```
// recuperar codigo de aplicacion de configuracion
String codigoAplicacion = StaticsContainer.webAppName;
// crear token n38 de aplicacion
N38APISesion n38Sesion = new N38APISesion();
Document token = n38Sesion
    .n38APISesionCrearApp(codigoAplicacion);
// recuperar cabecera SOAP
SOAPHeader soapHeader = context.getMessage().getSOAPHeader();
// incluir token en la cabecera creada
Source source = new DOMSource(token);
Result result = new DOMResult(soapHeader);
this.transformerFactory.newTransformer().transform(source,
    result);
// se puede continuar el procesamiento
valid = true;
    } catch (Exception e) {
        e.printStackTrace();
    }
}
return valid;
}
```

Para poder utilizar el handler de cliente mediante Spring, se deberá de utilizar una clase que arrope el handler. Un ejemplo de la clase es la siguiente:

```
package com.ejie.bbb.webservice.util;

import java.util.ArrayList;
import java.util.List;

import javax.xml.ws.handler.Handler;
import javax.xml.ws.handler.HandlerResolver;
import javax.xml.ws.handler.PortInfo;

@SuppressWarnings(value = "rawtypes")
public class JaxWsPortProxyHandlerResolver implements HandlerResolver {

    private List<Handler> handlerChain;

    /**
     * handlerChain setter.
     *
     * @param handlerChain
     *      Parámetro a asignar.
     */
    public void setHandlerChain(List<Handler> handlerChain) {
        this.handlerChain = handlerChain;
    }

    /**
     * Constructor
     */
    public JaxWsPortProxyHandlerResolver() {
        this.handlerChain = new ArrayList<Handler>();
    }

    /**
     * Constructor inicializado con un handlerChain pasado como parámetro.
     *
     * @param handlerChain
     *      Parámetro a asignar.
     */
    public JaxWsPortProxyHandlerResolver(List<Handler> handlerChain) {
        this.handlerChain = handlerChain;
    }
}
```

```
/**
 * Devuelve la lista de handlers asociados.
 *
 * @param portInfo
 *     Permite al handler obtener la información necesaria del port.
 * @return Lista de handlers asociados.
 */
public List<Handler> getHandlerChain(PortInfo portInfo) {
    return this.handlerChain;
}
```

La configuración requerida para Spring se realizará en el fichero *webservice-config.xml*. En caso de no existir, deberá ser creado junto al resto de ficheros de configuración de spring en el EARClasses y se deberá modificar el fichero *beanRefContext.xml* para indicar su carga al inicio. Para este ejemplo esta sería la configuración que deberíamos incluir:

```
<?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-3.0.xsd">

    <bean id="fooWebService"
class="org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean">
        <property name="serviceInterface"
value="com.ejie.bbb.remoting.jaxws.wlgen.FooWebServiceImpl"/>
        <property name="wsdlDocumentUrl"
value="http://desarrollo.jakina.ejiedes.net:7001/aaaWebServiceWar/fooWebService?WSDL"/>
        <property name="serviceName" value="fooWebService"/>
        <property name="portName" value="fooWebServicePort"/>
        <property name="namespaceUri" value="http://com.ejie.aaa.webservice"/>
        <property name="lookupServiceOnStartup" value="false"/>
        <property name="handlerResolver" ref="jaxWsHandlerResolver"/>
    </bean>

    <bean id="jaxWsHandlerResolver" class="com.ejie.bbb.webservice.util.JaxWsHandlerResolver">
        <property name="handlerChain">
            <list>
                <bean class="com.ejie.bbb.webservice.util.N38TokenClientHandler" />
            </list>
        </property>
    </bean>
</beans>
```

De este modo, la clase *JaxWsPortProxyFactoryBean* de Spring, utilizada para la creación del cliente, será capaz de utilizar el handler de cliente.

6 Clientes de servicios expuestos con Ws-Security

El objeto de este apartado es explicar cómo implementar clientes de servicios expuestos con las políticas de WS-Security (WSS) aplicadas en la plataforma de integración.

En general, estos clientes serán externos (es decir, no estarán ubicados en los CPD's de EJIE), ya que los clientes internos deberían poder acceder al servicio en claro utilizando una seguridad basada en XLNets.

Es importante destacar que para la correcta comunicación con el Web Service que se desea invocar, los clientes que accedan a servicios expuestos con WSS, deberán ser propietarios de un certificado válido expedido por **Izenpe**.

6.1 Políticas de seguridad aplicadas en el OSB

Los servicios expuestos con seguridad WSS esperan recibir cabeceras conformes al siguiente nivel de estándar:

- *Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)*

Por norma, la plataforma añade políticas de autenticación y de integridad de mensaje (incluir firma de cabeceras y cuerpo), basadas en certificados **X.509**.

Por tanto, en las peticiones debe encontrarse:

- **BinarySecurityToken**: certificado X.509 con el que firma el cliente.
- **Timestamp**: cabecera de seguridad que indica cuando fue enviado el mensaje.
- **Signature**: firma del cuerpo y de la cabecera de timestamp que asegura la integridad del mensaje.

La plataforma es capaz de identificar al cliente con el certificado X.509 gracias a la firma incluida en el mensaje. Si dicha firma no estuviera presente, no sería posible asegurar que el cliente es propietario del certificado que manda.

El OSB refleja las políticas aplicadas al servicio expuesto transformando el descriptor WSDL del mismo. Desafortunadamente, estas políticas están definidas en un lenguaje **propietario** de Oracle Weblogic 9.x (espacio de nombres **wssp**: "<http://www.bea.com/wls90/security/policy>"), y únicamente los servicios contruidos en Weblogic 9 o superior con la tecnología **JAX-RPC** serán capaces de interpretarlas correctamente.

Esto no quiere decir que los clientes que vayan a invocar a servicios con WS-Security tengan que ser Weblogic 9 o superior. Ya que como veremos en el siguiente apartado se pueden ignorar estas políticas y definir una política estándar que implique las mismas aserciones y asignarla como la política usada por el cliente.

En el documento de Normativa de Desarrollo de Servicios para ERPI se dispone, a modo de referencia, de ejemplos del esquema de las cabeceras enviadas así como del descriptor WSDL transformado.

6.2 Cliente Weblogic 11

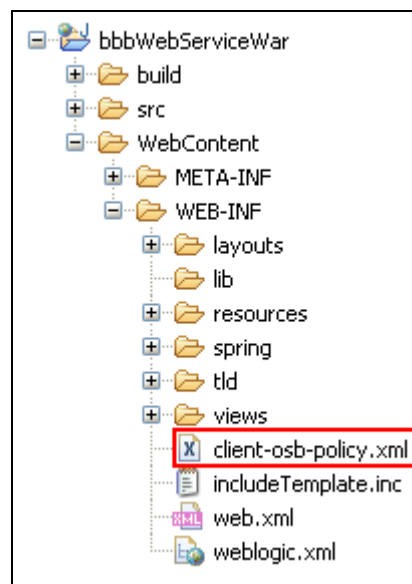
Uno de los objetivos de este documento es incidir en el uso de JAX-WS, el estándar de Web Services definido en JEE 5.

Como hemos dicho anteriormente, los clientes construidos en Weblogic 11 con tecnología JAX-WS no serán capaces de interpretar las políticas de seguridad con las que el OSB modifica los descriptores WSDL.

En Weblogic 11, los clientes JAX-WS son capaces de interpretar políticas de seguridad definidas mediante el estándar de seguridad WS-SecurityPolicy 1.2 (parcialmente soportado por el producto).

Por lo que, la solución para clientes Weblogic 11 pasa por ignorar las políticas propietarias del WSDL, definir una política estándar que implique las mismas aseveraciones y asignarla como política efectiva utilizada por el cliente.

En primer lugar, debemos crear el fichero "client-osb-policy.xml". Este fichero deberá situarse en el directorio WEB-INF del módulo War.



El contenido que deberemos indicar en el fichero es el siguiente:

```
<wsp:Policy wsu:Id="client-osb-policy"
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://schemas.xmlsoap.org/ws/2004/09/policy
http://schemas.xmlsoap.org/ws/2004/09/policy/ws-policy.xsd
http://schemas.xmlsoap.org/ws/2005/07/securitypolicy
http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/ws-securitypolicy.xsd">
  <sp:AsymmetricBinding>
    <wsp:Policy>
      <sp:InitiatorToken>
        <wsp:Policy>
```

```

        <sp:X509Token
sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/Always">
            <wsp:Policy>
                <sp:WssX509V3Token10 />
            </wsp:Policy>
        </sp:X509Token>
    </wsp:Policy>
</sp:InitiatorToken>
<sp:RecipientToken>
    <wsp:Policy>
        <sp:X509Token
sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/Always">
            <wsp:Policy>
                <sp:WssX509V3Token10 />
            </wsp:Policy>
        </sp:X509Token>
    </wsp:Policy>
</sp:RecipientToken>
<sp:IncludeTimestamp />
<sp:ProtectTokens />
<sp:OnlySignEntireHeadersAndBody />
</wsp:Policy>
</sp:AsymmetricBinding>
</wsp:Policy>

```

Este fichero define una política de seguridad estándar que realiza las mismas acciones que las políticas propietarias definidas por el OSB.

Si continuamos con el ejemplo de servicio desarrollado previamente, se deberá realizar lo siguiente para invocar el servicio expuesto con WS-Security:

Se implementará la siguiente clase que nos permitirá configurar la política de seguridad del cliente así como realizar la firma del mensaje e incluir el certificado X.509 en la petición:

```

package com.ejje.bbb.webservice.util;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import javax.xml.namespace.QName;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.Service;

import org.springframework.core.io.Resource;
import org.springframework.remoting.RemoteAccessException;
import org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean;

import weblogic.jws.jaxws.ClientPolicyFeature;
import weblogic.jws.jaxws.policy.InputStreamPolicySource;
import weblogic.wsee.security.bst.ClientBSTCredentialProvider;
import weblogic.xml.crypto.wss.WSSecurityContext;
import weblogic.xml.crypto.wss.provider.CredentialProvider;

/**
 *
 * @author UDA
 */
public class WsSecurityJaxWsPortProxyFactoryBean extends
    JaxWsPortProxyFactoryBean {

```

```
private QName portQName;
private Resource policyConfiguration;
private Object portStub;
private Resource keystorePath;
private String keystorePassword;
private String keyAlias;
private String keyPassword;

@Override()
public Object getPortStub() {
    return this.portStub;
}

/**
 * Setter del atributo policyConfiguration.
 *
 * @param policyConfiguration
 *      Resource que contiene la configuracion de la politica de
 *      seguridad del cliente.
 */
public void setPolicyConfiguration(Resource policyConfiguration) {
    this.policyConfiguration = policyConfiguration;
}

/**
 * Setter del atributo keystorePath.
 *
 * @param keystorePath
 *      Resource que contiene el keystore.
 */
public void setKeystorePath(Resource keystorePath) {
    this.keystorePath = keystorePath;
}

/**
 * Setter del atributo keystorePassword.
 *
 * @param keystorePassword
 *      Password del keystore.
 */
public void setKeystorePassword(String keystorePassword) {
    this.keystorePassword = keystorePassword;
}

/**
 * Setter del atributo keyAlias.
 *
 * @param keyAlias
 *      Alias del keystore.
 */
public void setKeyAlias(String keyAlias) {
    this.keyAlias = keyAlias;
}

/**
 * Setter del atributo keyPassword.
 *
 * @param keyPassword
 *      Password del alias.
 */
public void setKeyPassword(String keyPassword) {
    this.keyPassword = keyPassword;
}

@Override()
public void prepare() {

    ClientPolicyFeature clientPolicyFeature = this.getOSBClientPolicy();

    Service serviceToUse = this.getJaxWsService();
    if (serviceToUse == null) {
```

```

        serviceToUse = this.createJaxWsService();
    }

    this.portQName = this.getQName(this.getPortName() != null ? this
        .getPortName() : this.getServiceInterface().getName());

    Object stub = (this.getPortName() != null ? serviceToUse
        .getPort(this.portQName, this.getServiceInterface(),
            clientPolicyFeature) : serviceToUse.getPort(
            this.getServiceInterface(), clientPolicyFeature));

    this.preparePortStub(stub);
    this.addCredentialProviders((BindingProvider) stub);

    this.portStub = stub;
}

/**
 * Obtiene la politica de seguridad.
 *
 * @return ClientPolicyFeature.
 */
private ClientPolicyFeature getOSBClientPolicy() {
    ClientPolicyFeature clientPolicyFeature = new ClientPolicyFeature();
    try {
        clientPolicyFeature.setEffectivePolicy(new InputStreamPolicySource(
            this.policyConfiguration.getInputStream()));
    } catch (IOException e) {
        throw new RemoteAccessException(e.getMessage(), e);
    }
    return clientPolicyFeature;
}

/**
 * Anyade el security provider.
 *
 * @param bindingProvider
 *         BiningProvider sobre el que se van a anyadir los security
 *         providers.
 */
private void addCredentialProviders(BindingProvider bindingProvider) {
    List<CredentialProvider> credProviders = new ArrayList<CredentialProvider>();

    try {
        credProviders.add(new ClientBSTCCredentialProvider(keystorePath
            .getFile().getAbsolutePath(), keystorePassword, keyAlias,
            keyPassword));
    } catch (Exception e) {
        throw new RemoteAccessException(e.getMessage(), e);
    }

    if (credProviders != null && !credProviders.isEmpty()) {
        bindingProvider.getRequestContext().put(
            WSSecurityContext.CREDENTIAL_PROVIDER_LIST, credProviders);
    }
}
}

```

A continuación se deberá modificar el fichero webservice-config.xml para incluir la configuración correspondiente. En caso de no existir, deberá ser creado junto al resto de ficheros de configuración de spring en el EARClasses y se deberá modificar el fichero *beanRefContext.xml* para indicar su carga al inicio.

En este caso, el fichero quedaría del siguiente modo:

```
<?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-3.0.xsd">

    <bean id="wsSecurityfooWebService"
class="com.ejie.bbb.webservice.util.WsSecurityJaxWsPortProxyFactoryBean">
        <property name="serviceInterface"
value="com.ejie.bbb.webservice.servicio.ServicioPortType"/>
        <property name="wsdlDocumentUrl" value="${webservice.fooWebService.wsdl}"/>
        <property name="serviceName" value="fooWebService"/>
        <property name="portName" value="fooWebServicePort"/>
        <property name="namespaceUri" value="${webservice.fooWebService.namespace}"/>
        <property name="policyConfiguration" value="client-osb-policy.xml"/>
        <property name="keystorePath" value="${webservice.fooWebService.keystorePath}"/>
        <property name="keystorePassword" value="${webservice.fooWebService.keystorePassword}"/>
        <property name="keyAlias" value="${webservice.fooWebService.keyAlias}"/>
        <property name="keyPassword" value="${webservice.fooWebService.keyPassword}"/>
        <property name="lookupServiceOnStartup" value="false"/>
    </bean>
</beans>
```

Las propiedades que se deberán indicar en la declaración del cliente son las siguientes:

- **serviceInterface**: Interfaz del Web Service que ha sido generada por script ant utilizado para la generación de los recursos necesarios para la creación del cliente.
- **wsdlDocumentUrl**: URL del WSDL del Web Service para el que queremos generar el cliente.
- **serviceName**: Nombre del Web Service, presente en el WSDL.
- **portName**: Nombre del port, presente en el WSDL.
- **namespaceUri**: Uri del namespace, presente en el WSDL.
- **policyConfiguration**: Fichero que contiene la configuración de la política de seguridad a aplicar.
- **keystorePath**: Ruta física en la que se encuentra el almacén de claves JKS en el que se encuentran el certificado y la clave privada del cliente.
- **keystorePassword**: Contraseña del almacén de claves.
- **keyAlias**: Alias de la clave privada.
- **keyPassword**: Contraseña de la clave privada.

Por otro lado, estas serán las propiedades almacenadas en el fichero bbb.properties:

```
webservice.fooWebService.wsdl=http://desarrollo.jakina.ejiedes.net:7001/aaaWebServiceWar/fooWebService
?WSDL
webservice.fooWebService.namespace=http://com.ejie.aaa.webservice
webservice.fooWebService.keystorePath = classpath:/bbb/keystores/bbbKeystore.jks
webservice.fooWebService.keystorePassword = bbbPassword
webservice.fooWebService.keyAlias = bbbAlias
webservice.fooWebService.keyPassword = bbbPassword
```

Por último este sería un ejemplo de invocación del Web Service desde un controller:


```
package com.ejje.bbb.control;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;

import com.ejje.bbb.remoting.jaxws.wlgen.Foo;
import com.ejje.bbb.remoting.jaxws.wlgen.FooWebServiceImpl;

@Controller
@RequestMapping(value = "/wsSecurityfooWebService")
public class SpringJaxWsWebServiceController {

    @Autowired
    private FooWebServiceImpl wsSecurityJaxWsWebService;

    @RequestMapping(value =("/{bar1})", method = RequestMethod.GET)
    public @ResponseBody
    Foo getUsuario(@PathVariable String bar1) {

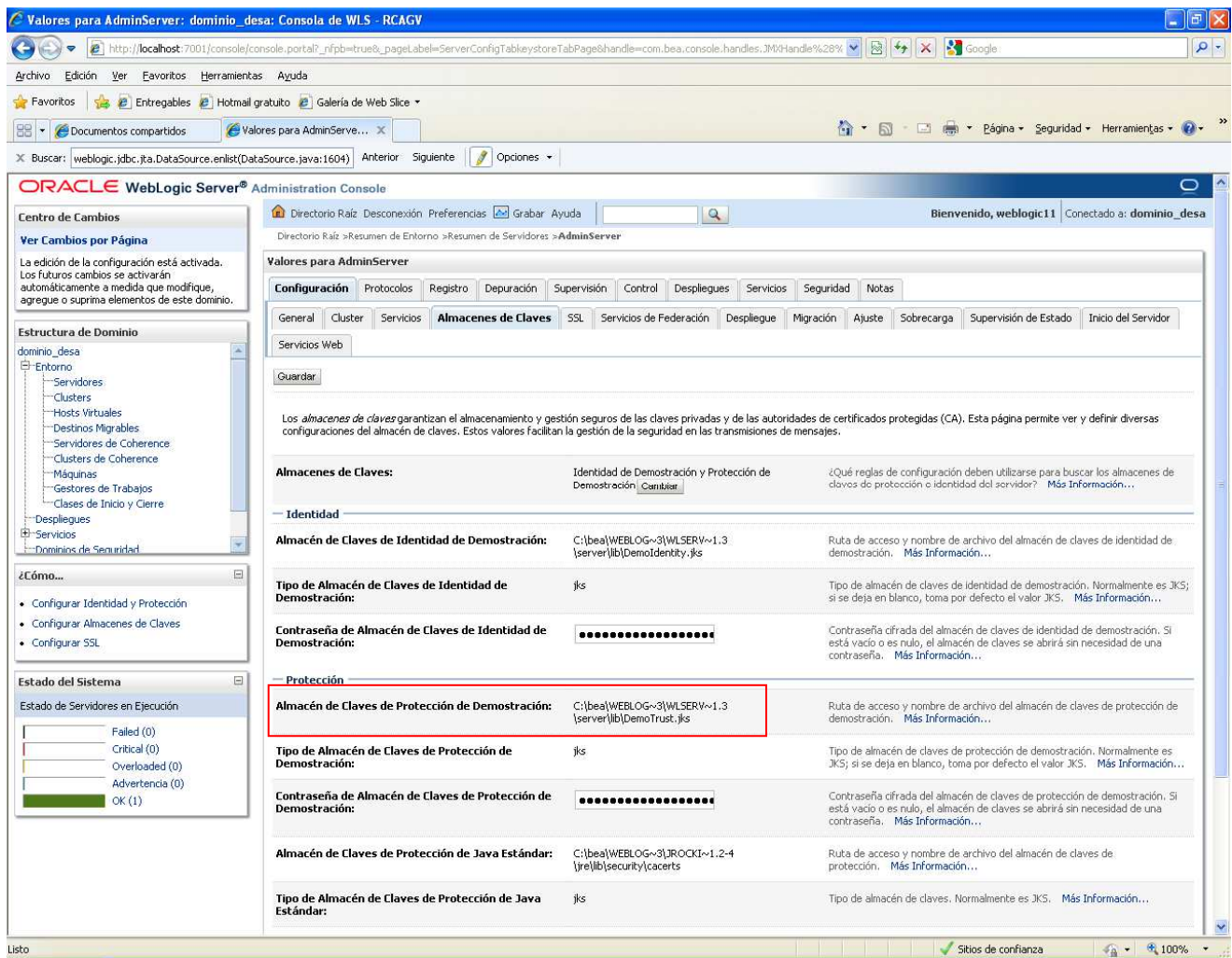
        Foo foo = new Foo();

        foo.setBar1(bar1);

        return this.wsSecurityJaxWsWebService.find(foo);
    }
}
```

Será necesario meter la ruta de certificación del certificado usado para firmar el mensaje en el almacén de certificados de confianza del servidor de Weblogic.

En un Weblogic local sería añadir los certificados raíz e intermedios en el keystore de certificados de confianza que se muestra en la pantalla siguiente.



6.3 Cliente wss4j

Para realizar la implementación de un cliente de webservice sin utilizar las clases propietarias de Weblogic (apartado 6.2) se puede hacer uso de la librería wss4j, para realizar las operaciones de seguridad. De este modo el cliente no será dependiente del servidor de aplicación Weblogic.

Como ejemplo de esta solución, se va a mostrar la creación de un cliente de webservice que hace uso de la librería wss4j mediante Spring-ws.

La configuración se deberá realizar en el fichero webservice-config.xml. En caso de no existir, deberá ser creado junto al resto de ficheros de configuración de spring en el EARClasses y se deberá modificar el fichero *beanRefContext.xml* para indicar su carga al inicio.

Este sería un ejemplo de la configuración que se deberá de incluir:

```
<?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:ctx="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
```

```

    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <ctx:property-placeholder location="classpath:/bbb/bbb.properties" />

    <bean id="marshaller" class="org.springframework.xml.jaxb.Jaxb2Marshaller">
        <property name="classesToBeBound">
            <list>
                <value>com.ejie.bbb.webservice.foo.webservice.Foo</value>
            </list>
        </property>
    </bean>

    <bean id="servicioClient" class="com.ejie.bbb.webservice.servicio.ServicioClient">
        <property name="template">
            <bean class="org.springframework.ws.client.core.WebServiceTemplate">
                <property name="defaultUri"
value="${webservice.fooWebService.endpointUri}" />
                <property name="marshaller" ref="marshaller" />
                <property name="unmarshaller" ref="marshaller" />
                <property name="interceptors">
                    <list>
                        <ref bean="wss4jWebServiceInterceptor" />
                    </list>
                </property>
            </bean>
        </property>
    </bean>

    <bean id="wss4jWebServiceInterceptor"
class="org.springframework.ws.soap.security.wss4j.Wss4jSecurityInterceptor">
        <property name="securementActions" value="Timestamp Signature" />
        <property name="securementSignatureKeyIdentifier" value="DirectReference" />
        <property name="securementUsername" value="${webservice.fooWebService.alias}" />
        <property name="securementPassword" value="${webservice.fooWebService.alias_pwd}" />
        <property name="securementSignatureParts">
            <value>
                {Element}{http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
utility-1.0.xsd}Timestamp;{}{http://schemas.xmlsoap.org/soap/envelope/}Body;Token
            </value>
        </property>
        <property name="securementSignatureCrypto">
            <bean
class="org.springframework.ws.soap.security.wss4j.support.CryptoFactoryBean">
                <property name="keyStorePassword"
value="${webservice.fooWebService.keystorePassword}" />
                <property name="keyStoreLocation"
value="${webservice.fooWebService.keystoreResource}" />
            </bean>
        </property>
        <property name="validationActions" value="NoSecurity" />
        <property name="enableSignatureConfirmation" value="true" />
        <property name="validationSignatureCrypto">
            <bean class="org.springframework.ws.soap.security.wss4j.support.CryptoFactoryBean">
                <property name="keyStorePassword"
value="${webservice.fooWebService.truststorePassword}" />
                <property name="keyStoreLocation"
value="${webservice.fooWebService.truststoreResource}" />
            </bean>
        </property>
    </bean>
</beans>

```

Para la creación del cliente de Web Service utilizaremos la clase *WebServiceTemplate*, clase principal para realizar el acceso a servicios web. Esta clase permite tanto el envío y recepción de mensajes así como la posibilidad de realizar el marshall de los objetos a XML antes del envío y el unmarshall de la respuesta XML en un objeto.

La definición se realizaría en el fichero *webservice-config.xml* del siguiente modo:

```
<bean id="servicioClient" class="com.ejie.bbb.webservice.servicio.ServicioClient">
  <property name="template">
    <bean class="org.springframework.ws.client.core.WebServiceTemplate">
      <property name="defaultUri" value="{webService.fooWebService.endpointUri}" />
      <property name="marshaller" ref="marshaller" />
      <property name="unmarshaller" ref="marshaller" />
      <property name="interceptors">
        <list>
          <ref bean="wss4jWebServiceInterceptor" />
        </list>
      </property>
    </bean>
  </property>
</bean>
```

Estas serían las propiedades para la configuración del *WebServiceTemplate*:

- **defaultUri**: Uri del Web Service que se desea invocar.
- **marshaller**: Bean que se va a utilizar para realizar el marshall de un objeto en un XML antes del envío de la petición.
- **unmarshaller**: Bean que se va a utilizar para realizar el unmarshall del XML de respuesta en un objeto sobre la respuesta de la invocación al Web Service.
- **interceptors**: Mediante esta propiedad podremos definir los interceptores que deberán ejecutarse al realizar la invocación al Web Service. En nuestro caso, especificaremos un interceptor que nos permitirá realizar las operaciones de seguridad requeridas.

Este sería un ejemplo de la definición del bean que realiza las operaciones de marshall y unmarshall:

```
<bean id="marshaller" class="org.springframework.xml.jaxb.Jaxb2Marshaller">
  <property name="classesToBeBound">
    <list>
      <value>com.ejie.bbb.webservice.fooWebService.Foo</value>
    </list>
  </property>
</bean>
```

La declaración del interceptor encargado de realizar las operaciones de seguridad se realiza del siguiente modo:

```
<bean id="wss4jWebServiceInterceptor"
class="org.springframework.ws.soap.security.wss4j.Wss4jSecurityInterceptor">
  <property name="securementActions" value="Timestamp Signature" />
  <property name="securementSignatureKeyIdentifier" value="DirectReference" />
  <property name="securementUsername" value="{webService.fooWebService.alias}" />
  <property name="securementPassword" value="{webService.fooWebService.alias_pwd}" />
  <property name="securementSignatureParts">
    <value>
      {Element}{http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd}Timestamp;{http://schemas.xmlsoap.org/soap/envelope/}Body;Token
    </value>
  </property>
  <property name="securementSignatureCrypto">
    <bean class="org.springframework.ws.soap.security.wss4j.support.CryptoFactoryBean">
      <property name="keyStorePassword"
value="{webService.fooWebService.keystorePassword}" />
      <property name="keyStoreLocation"
value="{webService.fooWebService.keystoreResource}" />
    </bean>
  </property>
```

```
<property name="validationActions" value="NoSecurity"/>
<property name="enableSignatureConfirmation" value="true"/>
<property name="validationSignatureCrypto">
  <bean class="org.springframework.ws.soap.security.wss4j.support.CryptoFactoryBean">
    <property name="keyStorePassword" value="\${webService.fooWebService.truststorePassword}"/>
    <property name="keyStoreLocation" value="\${webService.fooWebService.trustStoreResource}"/>
  </bean>
</property>
</bean>
```

Por último, estas serían las propiedades correspondientes incluidas en el fichero *bbb.properties*:

```
webService.fooWebService.alias=bbbAlias
webService.fooWebService.alias_pwd=bbbAliasPassword
webService.fooWebService.endpointUri=http://desarrollo.jakina.ejiedes.net:7001/aaaWebServiceWar/fooWebService
webService.fooWebService.keystoreResource=classpath:/bbb/keystores/bbbKeystore.jks
webService.fooWebService.keystorePassword=keystorePassword
webService.fooWebService.keystoreType=JKS
webService.fooWebService.trustStoreResource=classpath:/bbb/keystores/bbbTruststore.jks
webService.fooWebService.truststorePassword=truststorePassword
```

7 Referencias externas

Spring Web Services

<http://static.springsource.org/spring-ws/site/>

Oracle Desarrollo de clientes Weblogic 11 con WS-Security

http://download.oracle.com/docs/cd/E12840_01/wls/docs103/webserv_sec/message.html#wp237755

WSS4J

<http://ws.apache.org/wss4j/>

WS-SecurityPolicy 1.2

<http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html>