

¿Qué ofrece Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

Spring MVC, JSF-PrimeFaces /RichFaces,
HTML5, CSS3, JavaScript-jQuery

Gestor portales (Liferay)
Gestor de contenidos (Alfresco)
Aplicaciones híbridas

Tareas programadas (Quartz)
Gestor documental (Alfresco)
Inversión de control (Spring)

Control de autenticación y
acceso (Spring Security)
UDDI
Web Services
Rest Services
Social SSO
SSO (Cas)

JPA-Hibernate, MyBatis
Motor de búsqueda empresarial (Solr)
ETL (Talend)

Dirección de Proyectos Informáticos.
Metodologías ágiles
Patrones de diseño
TDD

BPM (jBPM o Bonita)
Generación de informes (JasperReport)
ESB (Open ESB)

- A partir de un fichero WSDL (WebServices Description Language).
 - útil cuando ya tenemos este tipo de fichero, o queremos reimplementar la funcionalidad de un webservice ya publicado en internet (y que nos da igual que este en .net, java o cualquier otra tecnología).
 - No es recomendable este método para generar un webservice desde 0, puesto que es trabajo de crear el fichero .wsdl puede llegar a ser muy tedioso.
 - En la versión usada, Metro 1.1, WSDL 2.0 no está soportado.
- A partir de una clase java.
 - Perfecto para crear un webservice desde 0, o publicar ciertos métodos de una aplicación java que tenemos ya desarrollada.
 - No recomendable cuando tengamos el fichero wsdl (El webservice generado puede no ser totalmente compatible con el del wsdl que teníamos).

Veamos detalladamente cada uno de los pasos.

Crear un WebService

Para todos los ejemplos vamos a usar el webservice público definido en 'http://wscc.info/index.php?show=32031_SWE&page_anchor=http://wscc.info/p32031/p32031_swe.php', y que devuelve información de geolocalización de una ip dada. Además el webservice está realizado en .net, y así comprobaremos si WSIT funciona.

Desde un fichero WSDL

Para generar el webservice desde este tipo de ficheros vamos a usar la herramienta que trae Metro, *wsimport*, y que nos va a crear todas las clases necesarias para poder poner en funcionamiento el webservice. Un ejemplo de uso del comandos sería:

```
wsimport.sh -s src/main/java -p com.autentia.service -Xnocompile http://ws.ip2location.com/ip2locationwebservice.asmx?wsdl
```

Donde:

- -s: ruta donde se van a generar los fuentes.
- -p : paquete en el que se generará el código.
- -Xnocompile : indica a la herramienta que no compile el código generado.

y nos generará las siguientes clases:

IP2Location.java: Java bean de tipo de dato complejo.
Ip2LocationWebService.java: Proxy del webservice.
IP2LOCATION.java: Java bean de tipo de dato complejo.
Ip2LocationWebServiceSoap.java: Interfaz del servicio.
IP2LocationResponse.java: Java bean de la respuesta del servicio.
ObjectFactory.java: Factoría de objetos usados internamente por el servicio.
package-info.java: Archivo para javadoc.

Como lo que vamos a hacer es reescribir el servicio web (hacer un webservice compatible con el wsdl usado pero realizado en java), no nos quedaría nada más que crear la clase skeleton del webservice (la que implementa la lógica de negocio del webservice).

Creamos una nueva clase java que tendrá la siguiente estructura (en este ejemplo la llamaremos IP2LocationSkel.java):

```
@WebService(endpointInterface="com.autentia.service.Ip2LocationWebServiceSoap")
public class IP2LocationSkel implements Ip2LocationWebServiceSoap {
    ...
}
```

Donde veis lo simple que es: creamos la clase y hacemos que implemente la interfaz generada anteriormente (no es obligatorio, pero si muy recomendable para saber cómo es el método a implementar en tiempo de compilación).

Lo que si es obligatorio es la anotación de la clase (y lo que le da potencia y sencillez a Metro): **@WebService**. La propiedad **endpointInterface** indica la clase (nombre cualificado) de la interfaz del webservice (y es por esto por lo que no es obligatorio implementar la interfaz anterior).

Y ya está! Ya tendríamos nuestro webservice creado, en pocos pasos y de manera muy sencilla!

Automatizar la tarea

TODO

Desde java

Vamos a hacer ahora la operación inversa. Queremos realizar un webservice desde 0, no tenemos el fichero WSDL o ya tenemos una clase java de la que queremos hacer públicos alguno de sus métodos a través de webservice.

No tendríamos más que generar una clase java, que debe cumplir los siguientes requisitos:

- La clase **se debe anotar** con **@javax.jws.WebService**.

Todos los métodos que se desean exponer como operación del webservice **se**

ejemplo

2008-04-04
Banca - Genérico -
MADRID.

2008-04-03
Banca - Genérico -
MADRID.

2008-04-02
T. Información -
Analista / Programador
- MADRID.

2008-04-02
T. Información -
Analista / Programador
- MADRID.

2008-03-29
T. Información -
Analista / Programador
- MADRID.

pueden anotar con `@javax.jws.WebMethod` (si todos los métodos de la clase son operaciones del servicio web se puede omitir dicha anotación).

- Todos los métodos que se desean exponer como operación del webservice **pueden** lanzar `java.rmi.RemoteException`, aparte de todas las excepciones específicas del servicio.
- Todos los parámetros de los métodos del servicio y tipos de retorno **deben** ser compatibles con JAXB 2.0 Java to XML Schema mapping definition.
- Los parámetros de los métodos del servicio y tipos de retorno **no pueden** implementar directa o indirectamente la interfaz `java.rmi.Remote`.

Veamos una sección de la clase (una clase que implementa una pequeña calculadora):

```
@WebService
public class AddNumbersImpl {

    @WebMethod
    public int addNumbers(int number1, int number2) throws AddNumbersException {
        ...
    }
}
```

Fijaos en las anotaciones: hemos anotado la clase con `@WebService`, y el método `addNumbers` con `@WebMethod` (el único que expondremos como método del servicio web).

El siguiente paso: generar las clases de apoyo necesarias para poner en marcha el webservice, y para ello usaremos otra herramienta que nos provee Metro, y que se llama `wsgen`

```
wsgen.sh -cp -keep com.autentia.service.AddNumbersImpl
```

Donde:

- `-cp` : indica el classpath a utilizar, es decir, la ruta o rutas de .jar o carpetas que contienen los .class necesarios para la compilación.
- `-keep` : indica que no se borren los ficheros .java generados.
- `com.autentia.service.AddNumbersImpl`: para el ejemplo, es la clase que implementa el webservice (denominado *SEI: service endpoint implementation*).

Que nos generará las siguientes clases:

AddNumbersExceptionBean.java : java bean para la excepción que puede lanzar el webservice.

AddNumbers.java : Skeleton del webservice.

AddNumbersResponse.java : clase de tratamiento de la respuesta SOAP.

Ya tenemos nuestro webservice! (sencillo, eh?)

Creando una aplicación web con nuestro webservice (war)

Una vez creado nuestro webservice veamos qué recursos y configuraciones debemos incluir para que el webservice funcione como una aplicación web y la podamos publicar (yo he usado tomcat y funciona sin problemas).

Ante todo debemos meter en la carpeta WEB-INF/lib de nuestro war los .jar (webservice*.jar) que vienen en la instalación de Metro.

Dentro de WEB-INF deben ir localizados web.xml y sun-jaxws.xml

El fichero de descripción de la aplicación web, web.xml, debe contener el siguiente fragmento (se define el servlet y filtros necesarios para que funcione Metro):

web.xml:

```
...
<listener>
    <listener-class>
        com.sun.xml.ws.transport.http.servlet.WSServletContextListener
    </listener-class>
</listener>
<servlet>
    <description>JAX-WS endpoint - fromjava</description>
    <display-name>metro ws engine servlet</display-name>
    <servlet-name>metroServlet</servlet-name>
    <servlet-class>
        com.sun.xml.ws.transport.http.servlet.WSServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>metroServlet</servlet-name>
    <url-pattern>/ws/*</url-pattern>
```

...

El fichero de descripción del webservice para Metro:

sun-jaxws.xml:

```
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime"
version="2.0"
  <endpoint
    name="addnumbers-example"
    implementation="com.autentia.service.AddNumbersImpl"
    url-pattern="/ws/addnumbers"/>
</endpoints>
```

¡Ojo! El url-mapping del servlet de webservice de metro y de la definición del webservice deben ser compatibles:

Válido:

`url-pattern="/ws/addnumbers"` ← `sun-jaxws.xml`

`<url-pattern>/ws/*</url-pattern>` ← `web.xml`

No Valido:

`url-pattern="/addnumbers"` ← `sun-jaxws.xml`

`<url-pattern>/ws/*</url-pattern>` ← `web.xml`

Una vez empaquetado el código convenientemente podemos instalar la aplicación en Tomcat, y acceder a `http://localhost:8080/<war-name>/ws/addnumbers`:

Web Services	
Endpoint	Information
Service Name: {http://service.autentia.com/}AddNumbersImplService	Address: http://localhost:8080/autentiaWS/ws/addnumbers
Port Name: {http://service.autentia.com/}AddNumbersImplPort	WSDL: http://localhost:8080/autentiaWS/ws/addnumbers?wsdl
	Implementation class: com.autentia.service.AddNumbersImpl

podemos ver el wsdl generado al pulsar sobre el link que nos propone la página.

Generar el cliente

Bien, ya he creado mi web service, ya lo he publicado... y ahora cómo lo uso? O cómo me conecto a un webservice externo?. Veamos cómo crear entonces un cliente para webservice, y la manera es muy sencilla, y es 99% igual que la forma en que generábamos un servicio web a partir de un wsdl.

Ejecutando el comando (la URI del wsdl tomaremos la del webservice del ejemplo comentado al inicio):

```
wsimport.sh -s src/main/java -p com.autentia.service -Xnocompile http://ws.ip2location.com/ip2locationwebservice.asmx?wsdl
```

con el que ya vimos anteriormente qué clases se generaban automáticamente.

La diferencia entre crear un webservice a partir de un WSDL y un cliente es la clase que tenemos que crear. Ahora codificaremos una clase como sigue, que será la que se conecte y haga la llamada al webservice:

```
public class IP2LocationStub implements IP2LocationWebServiceSoap {
    public IP2LOCATION ip2Location(String ip, String license) {
        // Create Service
        IP2LocationWebService service = new IP2LocationWebService();

        // create proxy
        IP2LocationWebServiceSoap proxy = service.getIP2LocationWebServiceSoap();

        // invoke
        return proxy.ip2Location(ip, license);
    }
}
```

Donde, como al crear el webservice, no es obligatorio que la clase implemente la interfaz `Ip2LocationWebServiceSoap`, pero si es muy recomendable.

Si llamáis al webservice del ejemplo necesitáis una clave de uso, con lo que la llamada volverá con los datos vacíos y con un mensaje de error, pero eso implica que el ejemplo funciona!!!

Conclusión

Ya habéis visto lo sencillo que es crear un servicio web y un cliente de webservice con Metro (y hemos probado además que WSIT funciona!).

En mi humilde opinión el desarrollo a base de anotaciones agiliza y simplifica mucho la tarea de generar web services, además nos permite reutilizar clases que ya teníamos.

En cuanto al rendimiento, existe en http://weblogs.java.net/blog/kohsuke/archive/2007/02/jaxws_ri_21_ben.html una comparativa de Metro vs Axis2, donde se muestra que para el envío de ciertos tipos de datos, Metro puede llegar a ser el doble de rápido que axis2 (por tanto más eficiente), aunque claro, la comparativa está realizada por uno de los propios desarrolladores de Metro... A final vuestra experiencia será la que diga la última palabra.

No os perdáis los próximos tutoriales sobre metro donde veremos cómo automatizar las tareas que realizábamos 'a mano' mediante maven2, y otro que nos enseñará a usar RESTFull con Metro.

- Puedes opinar sobre este tutorial haciendo clic aquí.
- Puedes firmar en nuestro libro de visitas haciendo clic aquí.
- Puedes asociarte al grupo AdictosAlTrabajo en XING haciendo clic aquí.
- Añadir a favoritos Technorati.



Esta obra está licenciada bajo licencia Creative Commons de Reconocimiento-No comercial-Sin obras derivadas 2.5

Recuerda

Autentia te regala la mayoría del conocimiento aquí compartido (Ver todos los tutoriales). Somos expertos en: J2EE, Struts, JSF, C++, OOP, UML, UP, Patrones de diseño ... y muchas otras cosas.

¿Nos vas a tener en cuenta cuando necesites consultoría o formación en tu empresa?, ¿Vas a ser tan generoso con nosotros como lo tratamos de ser con vosotros?

Somos pocos, somos buenos, estamos motivados y nos gusta lo que hacemos ...

Autentia = Soporte a Desarrollo & Formación.

info@autentia.com

Servicio de notificaciones:

Si deseas que te enviemos un correo electrónico cuando introduzcamos nuevos tutoriales.

Formulario de suscripción a novedades:

E-mail

Tutoriales recomendados

Nombre	Resumen	Fecha	Visitas	pdf
Como hacer visible en toda la red nuestra máquina virtual con KVM, en Debian GNU/Linux	En este tutorial Alex no enseñara cómo hacer visible en toda la red nuestra máquina virtual con KVM, en Debian GNU/Linux	2007-10-22	1329	pdf
XMLBeans, una forma de mapear un XML en objetos Java	En este tutorial vamos a ver una introducción a XMLBeans para ver como podemos obtener, a partir de un DTD o un XSD, las clases Java que procesan los XML que cumplen ese DTD o ese XSD.	2007-08-20	3106	pdf
Primeros pasos con PostgreSQL en Debian	En este tutorial Germán nos enseña a dar nuestros primeros pasos con postgresQL en Debian.	2008-02-20	553	pdf
EJB 3.0, un ejemplo práctico con Maven y JBoss	Este tutorial presenta un ejemplo sencillo donde se verá como desarrollar EJBs de sesión y de entidad, inyección de dependencias, llamar a los EJBs desde una aplicación Web, definición de un DataSource, y como configurarlo y hacerlo funcionar en JBoss, y	2007-08-06	4073	pdf
WebServices con Axis y JBoss	En este tutorial os mostramos como realizar servicios web utilizando Axis y el contenedor de aplicaciones web JBoss	2006-04-03	16051	pdf
Creación e invocación de Webservices por SSL	En este tutorial se pretende enseñar a desplegar un webservice usando SSL y a invocarlo correctamente	2006-11-29	8081	pdf
Q, un plugin de Eclipse para gestionar Maven 2	en este tutorial aprendemos a instalar y usar Q, un plugin de Eclipse para gestionar Maven 2	2007-10-24	1826	pdf
Generador automático de Webservices	Os mostramos como crear un servicio Web a partir de una clases, gracias a generadores automáticos de código y NetBeans	2003-10-16	33801	pdf

Nota:

Los tutoriales mostrados en este Web tienen como objetivo la difusión del conocimiento. Los contenidos y comentarios de los tutoriales son responsabilidad de sus respectivos autores. En algún caso se puede hacer referencia a marcas o nombres cuya propiedad y derechos es de sus respectivos dueños. Si algún afectado desea que incorporemos alguna reseña específica, no tiene más que solicitarlo. Si alguien encuentra algún problema con la información publicada en este Web, rogamos que informe al administrador rcanales@adictosaltrabajo.com para su resolución.