WebServices

The Java API for XML-based remote procedure calls (JAX-RPC) simplifies the process of building Web services that incorporate XML-based RPC. It defines mappings between Java types and XML types that attempt to hide the details of XML and provide a familiar method-call paradigm.

Axis supports 4 types of web service styles i.e. RPC, DOCUMENT, Wrapped and Message. But the ‘document/literal' type web services are most interoperable with wide range of clients. the document literal wrapped style. Web services built using this approach provide the benefit of complete self-description, standards conformance, and robustness in the face of changing requirements.

**Invocation Modes and Clients**

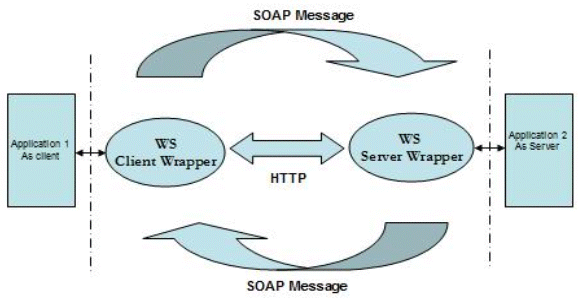
JAX-RPC supports three kinds of Web services invocation modes:

1. *Synchronous request-response:* A client invokes a remote method on a Web service and the thread blocks while it is processed by the Web service and receives a return value or an exception.
2. *One-way RPC mode:* A client invokes a remote method on a Web service in one-way mode and the thread does not block and continues execution. The client does not get any return value.
3. *Non-blocking RPC mode:* A client invokes a remote method on a Web service and continues processing in the same thread. Later, the client processes the remote method by performing a blocking receive or polling for return values.

The following types of parameters are supported by *jax-rpc*:

1. *IN type:* An *IN* parameter is passed as a copy. The value of the *IN* parameter is copied before a Web service invocation. The return value is created as a copy and returned to the Web service client.
2. *OUT type:* An *OUT* parameter is passed as a copy without any input value to the Web service method. The Web service method fills out the *OUT* parameter and then returns it back to the client.
3. *IN OUT type:* An *INOUT* parameter is passed as a copy with an input value to the Web service method. The Web service method uses the input value, process it, fills in the *INOUT* parameter with a new value and returns it back to the client.
4. The parameter passing mode for out and inout parameters uses Holder classes. The use of Holder classes enables the mapping to preserve the intended WSDL signature and parameter passing semantics. The JAX-RPC specification includes "Holder classes" for the mapping of simple XML types to the Java data types. The holder classes for primitive ones (for ex, int, float, and so forth) are available with jax-rpc implementation under the javax.xml.rpc.holders package. For the complex XML data types, the name of the Holder class is constructed by appending Holder to the name of the corresponding Java class. These generated Holder classes are packaged as part of the generated sub package named holders in the wsdl to Java mapping.

### Web Service Invocation

The Apache Tomcat-Axis combination provides a JAX-RPC 1.0 compliant runtime engine, which has both client-side and service-side libraries and deployment tools. Figure 1 elaborates the normal Web service invocation architecture for the synchronous request-response mode. The Application1 with WS client wrapper uses the JAX-RPC runtime to perform a remote procedure call to invoke a public method of Application 2 with the WS server wrapper. The client uses runtime libraries to serialize Java objects to a SOAP message and sends it to the Web service end point, using HTTP transport. As the Web service side that is deployed on Apache Tomcat receives this request, the service-side JAX-RPC runtime deserializes the SOAP message in to Java types and invokes the method on the Web service and in turn makes a call to Application 2. The Web service, after processing the request, sends response back to the client in a similar fashion.  


**Figure 1. Web Services Invocation Architecture**

# Developing JAX-RPC–Based Web Services Using Axis and SOAP

* July 18, 2003
* By [Rajesh Sumra](http://www.developer.com/author.php/64203/Rajesh--Sumra.htm)
* [Bio »](javascript:void(0))
* [Send Email »](http://www.developer.com/feedback.php/http:/www.developer.com/open/article.phpr/2237251/Developing-JAX-RPCndashBased-Web-Services-Using-Axis-and-SOAP.htm)
* [More Articles »](http://www.developer.com/author.php/64203/Rajesh--Sumra.htm)

### Sample Web Service: Order Processing Web Service

I have chosen an 'order processing' example for its proximity to real business use case. This Web service will be capable of processing and updating a given order. For this, it will have two methods: processOrder and updateOrder. processOrder will be taking an orderID string as an IN parameter and an Order object as an OUT parameter. It will return a status string as a return parameter. updateOrder will be taking the Order object as an INOUT parameter and it will update orderDate and return the Order object back to the client. As both of these methods are using a complex datatype, Order, that is also an OUT/INOUT parameter, a holder class has to be developed. The Order class and its holder class are given below in Listings 1 and 2 (For clarity, the whole code is in the sample package):

Listing 1: Class Order.

package sample;

public class Order {

// ID for order

private String orderID = null;

// date of order

private String orderDate= null;

// getter methodspublic String getOrderID() {

return orderID;

}

public String getOrderID() {

return orderID;

}

// setter methods

public void setOrderID(String orderID) {

this.orderID = orderID;

} public void setOrderDate(String orderDate) {

this. orderDate = orderDate;

}

}

Listing 2: Holder class for Class Order.

// Note that holder class is in the holders package and its name

// is derived by adding Holder as a suffix to 'Order', as per

// the JAX-RPC specification.

package sample.holders;

public class OrderHolder {

// Order's object

public Order value = null;

// default constructor

public void OrderHolder () { }

// constructor, which takes value as a parameter

public void OrderHolder (Order value) {

this.value=value;

}

}

Now, let's develop our Web service for the described functionality. The code is given in Listing 3.

Listing 3: Order Processing Web service.

package sample;

public class OrderProcessingService {

// Method 1: processes a order given ID as input and

// return status and Order object as an OUT parameter

public String processOrder(String orderID,

OrderHolder orderHolder ) {

String status = "pending";

// perform business logic here

// for simplicity just filling the Order object

Order order = new Order();

order.setOrderID(orderID);

order.setOrderDate("03 March 2003");

// set the Holder value to the order.

orderHolder.value = order;

//set the status

status = "complete" ;

return status;

}

// Method 2: updates a order given Order as an INOUT

// parameter and returns status.

public String updateOrder(OrderHolder orderHolder) {

String status = "pending";

// perform update here

Order order = orderHolder.value;

order.setOrderDate("03 April 2003");

// Note that orderID is not changed.

// It will be same as the passed one.

// set the Holder value to the order.

orderHolder.value = order;

//set the status

status = "complete" ;

return status;

}

}

Now that we have finished with Web service development, the next step is to compile and deploy it on a Tomcat-Axis platform. After compilation, we need to deploy the preceding Web service on Tomcat-Axis, using a deployment descriptor. The sample 'deploy.wsdd' is shown in Listing 4 (for more details, please refer to the Tomcat-Axis Manual).

Listing 4: deploy.wsdd for deployment.

<deployment xmlns="http://xml.apache.org/axis/wsdd/"

xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

<service name=" OrderProcessingService" provider="java:RPC">

<parameter name="className"

value="sample.OrderProcessingService "/>

<parameter name="allowedMethods" value="\*"/>

<operation name="processOrder">

<parameter name="arg1" mode="IN"/>

<parameter name="arg2" mode="OUT"/>

</operation>

<operation name="updateOrder">

<parameter name="arg1" mode="INOUT"/>

</operation>

</service>

</deployment>

The above deployment descriptor actually tells the server about the Web service, such as methods it exposes, parameters it is expecting, and the return types. To deploy the OrderProcessingService, we need to invoke the Axis admin service, passing this "deploy.wsdd". The admin service, which is also running on the same server, will process the descriptor and deploy the Web service, so that it is ready to be invoked by clients.

Execute the following command from where you have kept the deploy.wsdd:

java -cp %AXISCLASSPATH% org.apache.axis.client.AdminClient

-lhttp://localhost:8080/axis/services/AdminService deploy.wsdd

Where AXISCLASSPATH is used to set the environment for Axis (for details, see the [axis installation guide](http://ws.apache.org/axis/)).

The OrderProcessing service will be available at the following URL:  
http://<your\_machine\_name>:<port-num>/<contextURI>/<serviceURI>.

In our case, it may look like:  
[http://localhost:8080/axis/services/OrderProcessing](http://www.developer.com/open/article.php/10930_2237251_2/%3ehttp:/localhost:8080/axis/services/OrderProcessing)

### OrderProcessing Web Service Client

#### Dynamic client

Dynamic client is analogous to looking up and invoking the Java class methods using the reflection APIs.

Here, all the information, such as target endpoint, method parameters, and so forth has to be set explicitly. The code shown in Listing 5 will tell how to write a dynamic client for invoking the updateOrder method, in the OrderProcessing Web service.

Listing 5: Dynamic Client

package sample.client;

import org.apache.axis.client.Call;

import org.apache.axis.client.Service;

import org.apache.axis.encoding.XMLType;

import javax.xml.rpc.ParameterMode;

import javax.xml.rpc.encoding.\*;

import javax.xml.namespace.QName;

import java.util.\*;

import sample.\*;

/\*\*

\* This class illustrates how to use the JAX-RPC API to invoke

\* the Order Processing Web service dynamically

\*/

public class DynamicClient {

public static void main(String[] args) throws Exception {

// create service factory

ServiceFactory factory = ServiceFactory.newInstance();

// define qnames

String targetNamespace = "OrderProcessingService";

QName serviceName = new QName(targetNamespace,

"OrderProcessingService");

QName portName = new QName(targetNamespace,

"OrderProcessingService");

QName operationName = new QName(targetNamespace, "updateOrder");

// create service

Service service = new Service();

Call call = (Call) service.createCall();

Qname qn = new Qname(targetNamespace, "OrderHolder");

call.registerTypeMapping(OrderHolder.class, qn,

new org.apache.axis.encoding.ser.BeanSerializerFactory

(OrderHolder.class, qn),

new org.apache.axis.encoding.ser.BeanDeserializerFactory

(TicketHolder.class, qn));

// set port and operation name

call.setPortTypeName(portName);

call.setOperationName(operationName);

// add parameters

call.addParameter( "arg1", serviceName, ParameterMode.INOUT );

call.setReturnType( XMLType.XSD\_STRING );

Order order = new Order ();

order.setOrderID("Order001");

order.setOrderDate("03 March 2003");

// set end point address

call.setTargetEndpointAddress(

"http://localhost:8080/axis/services/OrderProcessing");

// Invoke the WebService

String result = (String) call.invoke( new Object[] { order } );

System.out.println("result : " +result);

Map outparams = call.getOutputParams();

System.out.println("Got the outparams");

}

### Running the Client

To run the client, use this command:

<Prompt>java -cp %AXISCLASSPATH% sample.client.DynamicClient

The result will be:

Got the outparams (as mentioned in client we developed)