Webservice

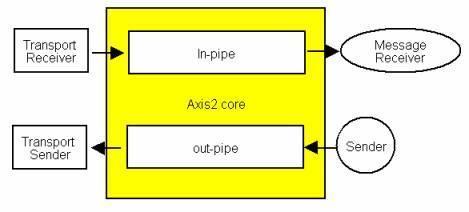
#### **What are web services?**

A web service is a business function that is self-contained and operates over the internet. The W3C defined web service as “a software system designed to support interoperable machine-to-machine interaction over a network”. XML is the base for the web services. They communicate using open protocols.

Web services are more frequently like Web APIs. These services accessibility can be from internet. A remote system can host these services for different web services. A web service can convert a stand alone / desktop application into a web based / web application. A web services are published or found and utilized through the internet. The web services works on the XML + HTTP platform.

Apache Axis2 is said to be a well-defined framework for Web services and related components developers. It has a lot of room for extending its main functionality; among them are pluggable modules, systems, even listeners and pluggable transport framework assume to be very useful. One of the most important features in Axis2 is transport independency. Axis2 does not depend on transport and Axis2 is a pure SOAP processing engine. It is the user's responsibility to select transports that match his requirements.

An asynchronous Web service is one of the main considerations in these days of Web service development, and developers are very keen on developing their Web service in an asynchrony manner. Axis2 has inbuilt capability of asynchronous Web service handling; in the meantime, none of the request-response concept has burnt into Axis2. As far as Axis2 is concerned, what it does is either process incoming messages or process outgoing messages and delivers the message to MessageReceiver or to TransportSender. In the case of an incoming message, it is the MessageReceiver; in the case of an outgoing message, it is the TransportSender. Figure 1 shows the graphical view of message processing inside Axis2 and also highlights the location of transport's fit in the flow.



**Figure 1:** Message processing inside Axis2

<BLOCKQUOTE**Note:** The sender is the one that called AxisEngine.send(MessageContext) so it could be either MessageReceiver (if there is any response for the incoming message), or OperationClient (in the case of client side).

### **Transport Receiver**

In the Axis2 world, Transport receiver is the one that calls AxisEngine.receive(MessageContext); in other words, the one that initializes the execution for an incoming message. In the case of a real-time system transport, the receiver is an application server or simply a listener that listens to a specific port.

There are two ways that someone can start a transport listener. The first way is to create an Axis2 system by itself and then start a listener (AxisServlet uses this technique). The second way is start listeners by using ListenerManager API (which will be discussed later in this article).

Irrespective of the way you start listener, it has to keep a reference to ConfigurationContext (which is the run-time representation of the system). When transport receiver receives a message, there are few steps that listener has to carry out:

1. Create a new messageContext; that will be the placeholder for the incoming SOAP message as well as related properties.
2. Set the configurationContext reference in messageContext (Transport Listener has a reference to the configuration context).
3. Extract transport-related information into messageContext (header information, input/output stream, and so forth).
4. Set the transport prefix into message context (the requirement of this will be discussed later).
5. Create a SOAPMessage for the incoming message. As a result of AXIOM being the underline message representation, it will not read the entire message into memory until someone asks to do that.
6. Add the created SOAPMessage into messageContext.
7. Create a new AxisEngine.
8. Call axisEngine.receive(message Context).

**Note:** If the transport is a two-way transport like HTTP, there is one more step left. Once the transport receiver gets the control back (Java return back), it has to check the message context to determine whether the response is written or not. If it is written, there will be a flag in the message context to indicate that msgCtx.isResponseWritten(). If it is not, it is required to send an HTTP 202 and close the incoming stream.

### **Transport Sender**

As its name implies, Transport sender is the one that sends the request message or simply writes the content to an output stream. To be a valid transport sender, it has to implement a *"TransportSender"* interface, and that interface extends the Handler interface so Transport sender is actually a handler with two more additional methods.

The simplest API that Axis2 provides to register transport senders is axis2.xml; there you can register the transport sender with the name and parameters (if it is required). As an example, the default HTTP sender in Axis2, CommonsHTTPTransportSender, has registered in axis2.xml as follows.

<transportSender name="http"

class="org.apache.axis2.transport.http.

CommonsHTTPTransportSender">

<parameter name="PROTOCOL" locked="false">HTTP/1.1</parameter>

<parameter name="Transfer-Encoding"

locked="false">chunked</parameter>

</transportSender>

**Note:** The name attribute is the name of the transport sender; it should be a unique name. There cannot be two transport senders with the same name, but it is possible to have a transport sender and a transport receiver with the same name. The parameters are to configure the sender; the class attribute represents the implementation class of the TransportSender.

When the transport receiver is compared with the transport sender, transport sender has the lesser job. It has to get the SOAP message from the message context and serialize that into the output stream of the sender. There are no two separate transport senders for server side and client side, but in some cases, the server side transport sender has to act differently than the client side sender. (The same class can be used with slightly different logics.) As a transport sender, the following are the common steps that need to be followed when sending a message.

1. If the message context has a To (target address) address, open up a connection. (On the client side, the TO address is needed, but on the server side it is not.)

String address = msgContext.getTo().getAddress();

1. Otherwise, try to find the Outputstream from the message context. (This happens on the server side when the request comes via two-way transport.)
2. Write a transport-specific header if it has any (in the case of HTTP, HTTP headers such as HTTP version, content type, soapaction, and the like).
3. Get the corresponding SOAPEnvelope from message context:

SOAPEnvelope env = msgCtx. getEnvelope();

1. Then, serialize the envelope into the output stream as follows:
2. SOAPEnvelope envelope = msgContext.getEnvelope();
3. OMElement outputMessage = envelope;
4. OMOutputFormat format = new OMOutputFormat();
5. outputMessage.serializeAndConsume(outputstream, format);
6. outputstream.flush();

### **Listener Manager**

Listener manager is a convenient way of managing transport receivers in a given system. It should be noted that the transport receivers discussed above may or may not be managed by using the listener manager API. The difference is that the receivers or listeners that Listener manager can handle has to implement a *"TransportListener"* interface, but it is not a must to implement that interface for the above-mentioned transport receivers.

#### **Transport listeners**

As discussed above, to be a listener, you must implement a TransportListener interface. It looks like the following:

public interface TransportListener {

void init(ConfigurationContext axisConf,

TransportInDescription transprtIn)

throws AxisFault;

void start() throws AxisFault;

void stop() throws AxisFault;

EndpointReference getEPRForService(String serviceName,

String ip) throws AxisFault;

}

* **init**: This method will be called when a transport listener is initialized. As discussed in the transport receiver section, a listener has to keep a reference to configuration context, so the configuration context object will be passed as an argument. **TransportInDescription** is the corresponding description object for the listener. The description object will contain the name of the transport and parameters specified in axis2.xml for this particular transport.
* **start**: Listener manager will call this method when it is asked to start a given transport, so all the transport starting logic should be here. For example, it should contain the starting server socket and the like.
* **stop**: To stop the transport, logic required to close the started socket can be written here.
* **getEPRForService** :Any service available in the system has a unique address to invoke the service, and if there are multiple transports there will be multiple addresses for a given service. The well-known name for that address is End Point Reference (EPR). From any given transport listener, it should be able to get a unique EPR for a given service. When dynamic WSDL is generated and to set the wsa:ReplyTo addresses, this method will be called. At that point, it is up to the transport receiver to create an EPR for a given service. If the transport is HTTP, the EPR will look like http://myaddress.com/foo/axis2/servives/ServiceName, and in the case of SMTP it will look like ServiceName@foo.myaddress.com. The second method parameter, IP, can be null; if it is null. the transport listener has to find the IP address somehow when it generates an EPR.

There are two ways to register a transport listener into the system. The first way is to add an entry into axis2.xml. The second way is to create TransportListener by hand and add that into Listener Manager. In the case of axis2.xml, you have to add the following XML element into axis2.xml;

<transportSender name="http"

class="org.apache.axis2.transport.http.CommonsHTTPTransportSender">

<parameter name="PROTOCOL" locked="false">HTTP/1.1</parameter>

<parameter name="Transfer-Encoding"

locked="false">chunked</parameter>

</transportSender>

Here, the name of the transport is HTTP and the listener implementation class is **CommonsHTTPTransportSender**. The corresponding description object has two parameters.

**Note:** If you want to have proxy support or port a forwarding mechanism when you are using axis2 default transports, the generated EPR can be customized by adding the following parameter into the transport tag. The parameter value should be a valid URL.

<parameter name=" hostname">

http://myaddress.com/foo/axis2/services

</parameter>

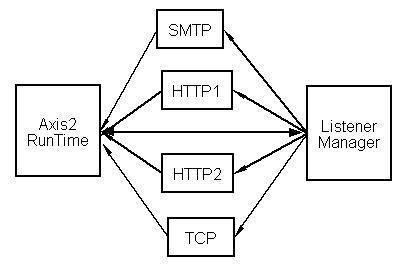
#### **API of the Listener Manager**

Listener manager provides the following APIs to mange transport listeners in a given system.

* **init**: To initialize the listener manager, you must create a configuration context and pass that as the method parameter. Inside the method, it will keep a reference to the configuration context and will set the listener manger field in the configuration context as the created listener manager.
* **start**: You can add any number of transport listeners in axis2.xml, but nothing will happen until you call this method. So, when you call the start method, it will take all the available transport listeners from axisConfiguration and then initialize and start that (will call the init and start methods, respectively).
* **startSystem**: You can just call startSystem rather than calling the two methods above. The inside method implementation will call the init and start methods.
* **stop**: This method can be used to stop all the listeners in the system.
* **addListener**: You can use this method to add a new listener into the system. When you add a listener, you have to pass two method arguments, the transport description object and boolean value. The transport description object has to have a transport listener in it, and the boolean value specifies whether the listener is running or not. If the boolean value is true, it will just add the listener into the list; otherwise, it will call both the init and start methods before adding them to the list. In the case of AxisServlet, it will use true as the boolean value because AxisServlet is running when it creates the listener manager.
* **isListenerRunning**: To check whether a given listener is up and running, if the listener is in the started listeners list, it will return true.
* **getEPRforService**: As discussed in the transport listener section, to get an EPR for a given service, listener manager itself provides an API to get the service EPR for a given service. The method takes two parameters, the name of the service and the name of the transport. Here the name of the service cannot be null but the name of the transport can be null. If the transport name is null, it will pick a transport from the started transport list and ask for the service EPR from that transport. Otherwise, if the transport name is not null, it will get the corresponding transport and ask the service EPR from that transport.

**Note:** In the case of Listener manager, all the transport receivers in the system share the same configuration context. The practical usage of this listener manager and its listeners is that one can expose same service using different transports or different addresses. A good use case could be publishing same service using HTTP, TCP, and SMTP. Isn't that cool?

The important thing in Axis2 is that it can receive a message via SMTP and a response can be send using HTTP. So, Axis2 can handle different transport combinations without having any problems. Figure 2 shows how Transport Listeners, Listener Manager, and Axis2 run time link together.



**Figure 2:** Listener Manager and Transport Listeners

#### **Running a client inside a server using server's runtime**

What does running a client inside a server mean?

Axis2 supports full asynchronous Web service invocation. There you can send the request in one transport and you can get the response via some other transport. In Axis2 terms, this is called *send/receive non blocking using two channels*. In this case, you have to have a running transport listener at the client side as well; otherwise, when someone asks to invoke two channels, the Axis2 invocation will start the corresponding transport receiver at the client side if that is present in the corresponding AxisConfiguration.

On the client side, when you create a ServiceClient, you can pass the configuration context, and when invoking a service client, you can use that configuration context as its runtime. The interesting thing is all the modules, service, properties, and transports that are available in that configuration context easily can be accessible by the service client.

When someone calls the init method of the listener manager, the corresponding field in the configuration context will be set. If the configuration context that passed to the service client does not have a listener manager object,a new one will be created and added into the configuration context and init the listener manager. Therefore, the service client always has a valid listener manager object in its configuration context.

When invoking a non-blocking, two-channel invocation, it first will try to find the incoming transport by using

String tarnsportName = Options.getTransportInProtocol ();

and then take the listener manger and check whether the given listener is running. If so, it will ask the EPR for the service using that transport. Otherwise, it will ask the listener manager to start the listener and then ask the EPR for the service.

So, if someone passes a configuration context with a listener manager when creating a service client, the service client has access to listeners in the system through the corresponding listener manager. As a result, if someone invokes a two-channel invocation, the service client can use the running transport listeners as its client-side listener for the response processing. To facilitate this, the listener manger has a static field to keep the configuration context. If you set that field at the time the listener manager is created, clients running inside the listener manager have the capability of using the server's configuration context. It should be noted that AxisServlet sets the static values; therefore, any client running inside Aixs2 war distribution can use AxisServlet itself as its transport receiver (no need to start a new listener for client response processing).

To access the server's configuration context, you can use the following line of code to create ServicClient:

ServiceClient serviceClient =

new ServiceClient(ListenerManager.defaultConfigurationContext, null)

#### **Axis2 default transport support**

Axis2 comes with a set of default transports senders and receivers; most of them use standard libraries. It is obvious that more than 90% of the Web service utilization is based on HTTP transports, but there is more of a trend of using other transports as well, especially SMTP. As a result of SOAP theoretically being the underline communication mechanism, you can use any of the transport mechanisms to communicate between two SOAP nodes. Axis2 supports the following types of transports and it has senders and receivers for them as well.

1. HTTP/HTTPS
2. SMTP/POP
3. TCP
4. JMS

### **Summary**

As mentioned earlier, Axis2 is transport independent and it does not care about the type of the senders and receivers, or all the transport-related code written using interfaces. It is very easy to implement those interfaces and extend the transport framework. Therefore, adding and removing a transport sender/receiver is not a headache; implementing a transport for a given protocol is just a matter of implementing the sender and receiver and register them in axis2.xml. At last, the Listener Manager API is very useful for applications such as BEPL, WS-Eventing, and WS-Notification, and so on.

Axis now delivers the following key features:

* **Speed.** Axis uses SAX (event-based) parsing to acheive significantly greater speed than earlier versions of Apache SOAP.
* **Flexibility.** The Axis architecture gives the developer complete freedom to insert extensions into the engine for custom header processing, system management, or anything else you can imagine.
* **Stability.** Axis defines a set of **published interfaces** which change relatively slowly compared to the rest of Axis.
* **Component-oriented deployment.** You can easily define reusable networks of Handlers to implement common patterns of processing for your applications, or to distribute to partners.
* **Transport framework.** We have a clean and simple abstraction for designing transports (i.e., senders and listeners for SOAP over various protocols such as SMTP, FTP, message-oriented middleware, etc), and the core of the engine is completely transport-independent.
* **WSDL support.** Axis supports the [Web Service Description Language](http://www.w3.org/TR/wsdl), version 1.1, which allows you to easily build stubs to access remote services, and also to automatically export machine-readable descriptions of your deployed services from Axis.
* **AXIOM** - Axis2 comes with its own light-weight object model, AXIOM, for message processing which is extensible, high performance and developer convenient
* **Hot Deployment** - Axis2 is equipped with the capability of deploying Web service & handlers while system is up and running. In other words, new services can be added to the system without having to shut down server.Drop the required Web service archive into the services directory in the repository and deployment model will automatically deploy the service and make it available for use.
* **Asynchronous Web Services** - Axis2 now supports asynchronous Web services & asynchronous Web services invocation using non-blocking clients and transports .
* **MEP Support** - Axis2 now comes handy with the flexibility to support Message Exchange Patterns (MEPs) with in-built support for basic MEPs defined in WSDL 2.0.
* **Flexibility** - The Axis2 architecture gives the developer complete freedom to insert extensions into the engine for custom header processing, system management, or *anything else you can imagine*.
* **WSDL support** - Axis2 supports the Web Service Description Language, version [1.1](http://link) and [2.0](http://link), which allows you to easily build stubs to access remote services, and also to automatically export machine-readable descriptions of your deployed services from Axis2.
* **Add-ons** Several Web services specifications have been incorporated including [WSS4J](http://window) for security (Apache Rampart), [Sandesha](http://window) for reliable messaging, [Kandula](http://window) which is an encapsulation of WS-Coordination, WS-AtomicTransaction and WS-BusinessActivity.
* **Composition and Extensibility** - modules and phases improve support for composability and extensibility. Modules supports composability and is able to add support for new WS-\* specifications in a simple and clean manner. They are however not hot deployable as they change the overall behavior of the system.

## ***Web Services Using Axis2***

Before starting, please check whether you have deployed the "axis2.war" in your servlet container and it is working properly. (See [Installation Guide](http://window)). User can select any of the  following two ways of writing Web services using Axis2.

1. Use Axis2's primary interfaces (APIs) and implement the business logic.
2. Start from the WSDL ->Code generate the Skeleton ->Implement the Business Logic.

### **Writing Web Services Using Axis2's Primary APIs**

### **Creating Web Service (MyService)**

First let's see how we can write a simple Web Service (MyService) using Axis2's primary interfaces and deploy it. For this purpose we will create a Web Service with two operations as follows.

public void ping(OMElement element){} //IN-ONLY operation, just accepts the OMElement and do some processing.

public OMElement echo(OMElement element){}//IN-OUT operation, accepts an OMElement and

// sends back the same again

Complete code for this example Web Service (MyService) can be found in the "Axis2Home/samples/userguide/src" directory under "userguide/example1" package. As you can see, the two operations are very simple and need no explanations on what they do. Now let's see how we can write the deployment descriptors for the service and deploy it.

### **How to write the Web Service?**

Writing a new Web Service with Axis2 involve four steps:

1. Write the Implementation Class
2. Write a services.xml file to explain the Web Service
3. create a \*.aar archive (Axis Archive) for the Web Service
4. Deploy the Web Service

### **Step1 :Write the Implementation Class**

Provides an implementation class that provides the business logic for the Web Service, it should have methods that match the operations in the Web Service. Unless you have data binding, the signature of the methods can have only one parameter of type OMElement.

public class MyService{

public void ping(OMElement element){

......

}

public OMElement echo(OMElement element){

......

}

}

### **Step2 :Write the services.xml file**

Axis2 uses "services.xml" to keep configurations for a Web Service. Each Web Service deployed in Axis2 needs a "services.xml" containing the configurations. "services.xml" for MyService will be as follows.

<service >

<description>

This is a sample Web Service with two operations, echo and ping.

</description>

<parameter name="ServiceClass" locked="false">userguide.example1.MyService</parameter>

<operation name="echo">

<messageReceiver class="org.apache.axis2.receivers.RawXMLINOutMessageReceiver"/>

<actionMapping>urn:echo</actionMapping>

</operation>

<operation name="ping">

<messageReceiver class="org.apache.axis2.receivers.RawXMLINOnlyMessageReceiver"/>

<actionMapping>urn:ping</actionMapping>

</operation>

</service>

*The above XML tags can be explained as follows:*

First comes the description and the service class.

<parameter name="ServiceClass" locked="false">userguide.example1.MyService</parameter>

We provide the name of the service implementation class a parameter in the services.xml file.

The next two xml tags describe the operations that are available in this service with respective message receivers.

<operation name="echo">

<messageReceiver class="org.apache.axis2.receivers.RawXMLINOutMessageReceiver"/>

<actionMapping>urn:echo</actionMapping>

</operation>

<operation name="ping">

<messageReceiver class="org.apache.axis2.receivers.RawXMLINOnlyMessageReceiver"/>

<actionMapping>urn:ping</actionMapping>

</operation>

Every operation must have a corresponding MessageReceiver class. When Axis2 engine receives a message, after the message is being processed by the handlers, it will be handed over to a MessageReceiver.   
  
For the "echo" operation we have used a **RawXMLINOutMessageReceiver** since it is an IN-OUT operation. For IN-ONLY operation "ping", we have used **RawXMLINOnlyMessageReceiver** as the message receiver.

The actionMapping is required only if you want to enable WS-Addressing. This will be used later in this user guide.

You can write a services.xml file to include a group of services instead of a single service. This makes management and deployment of a set of related services very easy. At runtime you can share information between these services within a single interaction using the ServiceGroupContext. If you hope to use this functionality, the services.xml file should have following format.

<serviceGroup>

<service name="Service1">

<!-- details for Service1 -->

</service>

<service name="Service2">

<!-- details for Service2 -->

</service>

<module ref="ModuleName" />

<parameter name="serviceGroupParam1" locked="false">value 1</parameter>

</serviceGroup>

Note : name of the service is a compulsory attribute

### **Step3 :Create the Web Service Archive**

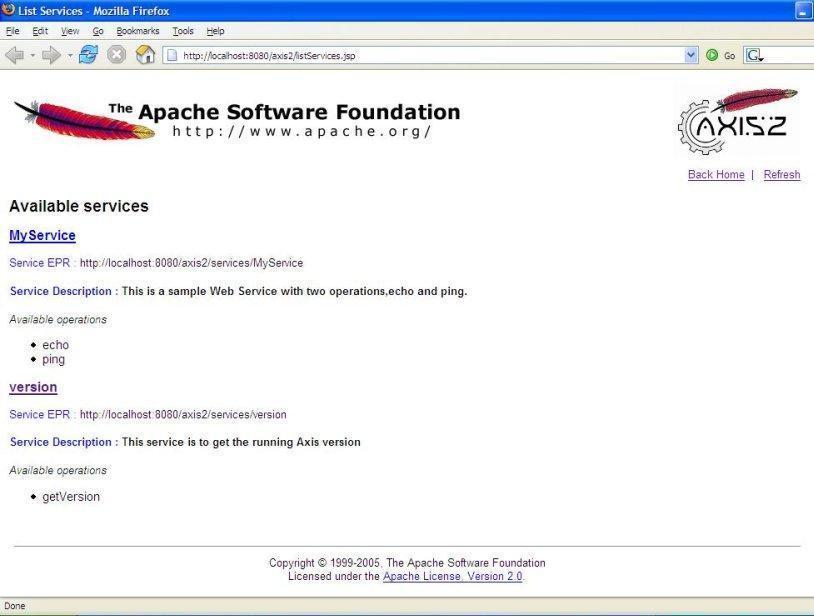
Axis2 uses ".aar" (Axis Archive) file as the deployment package for Web Services. Therefore, for MyService we will use "MyService.aar" with the "services.xml" packaged in the META-INF as shown in the following picture. (Name of the service will be the name of the archive file , if and only if the services.xml contains only one service element).



To create "MyService.aar" user can first create a jar file containing all the files necessary for the service and then rename the "jar" to "aar" so that Axis2 understands it as a service archive. This has already been created in the "Axis2Home/samples/userguide" directory. Now let's use it...

### **Step4 :Deploy the Web Service**

Deploying the service  is just a matter of dropping the ".aar" in to "services" directory that can be found in the "\webapps\axis2\WEB-INF" of your servlet container, hence copy the "MyService.aar" into the "**services**" directory. Once these steps are completed, start the servlet container (if you have not already started) and check the link "Services" on the [Home Page of Axis2 Web Application](http://window) (http://localhost:8080/axis2/index.jsp) and see whether the MyService is deployed properly. If you can see the following output then you have successfully deployed MyService on Axis2.



Note: Axis2 provides an easy way to deploy Web Services using the "Upload Service" tool on Axis2 Web Application's Administration module. (See the [Web Administration Guide](http://window) for more information on this)

### **Writing Web Services by Code Generating Skeleton**

This is the second method of writing Web Services using Axis2. Let's see how we can generate the skeleton from a given WSDL and implement the business logic using Axis2. For this we use Axis2SampleDocLit.wsdl that can be found in the **wsdl** directory under samples.

### **WSDL2Java Tool**

To generate the skeleton and the required classes you can use the WSDL2Java tool provided in Axis2. This tool is located in the bin directory of the distribution and can be executed using the provided scripts (.bat or .sh). The tool's parameter list is as follows and user can specify these values depending on their requirements.

Usage WSDL2Code -uri <Location of WSDL> : WSDL file location

-o <output Location> : output file location

-a : Generate async style code only. Default is off

-s : Generate sync style code only. Default is off. takes precedence over -a

-p <package name> : set custom package name

-l <language> : valid languages are java and csharp. Default is java

-t : Generate TestCase to test the generated code

-ss : Generate server side code (i.e. skeletons). Default is off

-sd : Generate service descriptor (i.e. services.xml). Default is off. Valid with -ss

-d <databinding> : valid databinding(s) are adb, xmlbeans and jaxme. Default is adb

-g Generates all the classes. valid only with the -ss

-pn <port\_name> : name of port in the presence of multiple ports

-sn <service\_name> : name of service in the presence of multiple services

-u : unpacks the databinding classes

-r <repository\_path> : path of the repository against which code is generated

We will use the tool with the following parameters and generate the skeleton and the other required classes.

Windows users can use the following command in the console:

WSDL2Java -uri ..\samples\wsdl\Axis2SampleDocLit.wsdl -ss -sd -d xmlbeans -o ..\samples -p org.apache.axis2.userguide

Linux users should switch the file separator:

WSDL2Java -uri ../samples/wsdl/Axis2SampleDocLit.wsdl -ss -sd -d xmlbeans -o ../samples -p org.apache.axis2.userguide

This will generate the required classes in the **src** directory inside samples, and the schema classes in **schemaorg\_apache\_xmlbeans** directory which is inside resources directory also inside samples dir. Note that these are not source files and should be available in the class path in order to compile the generated classes.

### **Implement the Business Logic**

Locate the skeleton class that can be found under src/userguide directory with the name "Axis2SampleDocLitPortTypeSkeleton.java". This is the skeleton for our Web service and we can now easily implement the business logic. The WSDL we have used has three operations:

* echoString  - Operation that echoes a String value
* echoStringArray - Operation that accept string array as the input and echoes them back
* echoStruct - Operation that accept a Struct as the input and echoes them back.

### **echoString**

Locate the following code segment  in the "Axis2SampleDocLitPortTypeSkeleton.java"  and fill the business logic as shown below.

public org.apache.axis2.userguide.xsd.EchoStringReturnDocument echoString

(org.apache.axis2.userguide.xsd.EchoStringParamDocument param4 ){

//Todo fill this with the necessary business logic

throw new java.lang.UnsupportedOperationException();

}

Once filled with the business logic it will be as follows. The code is simple and the explanations are given as comments.

public org.apache.axis2.userguide.xsd.EchoStringReturnDocument echoString

(org.apache.axis2.userguide.xsd.EchoStringParamDocument param4) throws Exception {

//Use the factory to create the output document.

EchoStringReturnDocument retDoc = EchoStringReturnDocument.Factory.newInstance();

//send the string back.

retDoc.setEchoStringReturn(param4.getEchoStringParam());

return retDoc;

}

Similarly following code fragments shows how you can fill the business logic for our first Web service.

### **echoStringArray**

public org.apache.axis2.userguide.xsd.EchoStringArrayReturnDocument echoStringArray

(org.apache.axis2.userguide.xsd.EchoStringArrayParamDocument param0) throws Exception {

//Use the factory to create the output document.

EchoStringArrayReturnDocument retDoc = EchoStringArrayReturnDocument.Factory.newInstance();

//Get the String array from the input parameters.

String[] inParams = param0.getEchoStringArrayParam().getStringArray();

ArrayOfstringLiteral retParams = ArrayOfstringLiteral.Factory.newInstance();

//Set the input parameters to the output parameters for echoing.

for (int i = 0; i < inParams.length; i++) {

retParams.addString(inParams[i]);

}

//return the output document.

retDoc.setEchoStringArrayReturn(retParams);

return retDoc;

}

### **echoStruct**

public org.apache.axis2.userguide.xsd.EchoStructReturnDocument echoStruct

(org.apache.axis2.userguide.xsd.EchoStructParamDocument param2) throws Exception {

//Use the factory to create the output document.

EchoStructReturnDocument retDoc = EchoStructReturnDocument.Factory.newInstance();

//Get the SOAPStrcut from the incoming parameters

SOAPStruct inStruct = param2.getEchoStructParam();

//Struct for the sending back

SOAPStruct outStruct = SOAPStruct.Factory.newInstance();

//Fill the outgoing struct

outStruct.setVarFloat(inStruct.getVarFloat());

outStruct.setVarInt(inStruct.getVarInt());

outStruct.setVarString(inStruct.getVarString());

//Set the outgoing document.

retDoc.setEchoStructReturn(outStruct);

return retDoc;

}

### **services.xml**

 Axis2 uses "services.xml" to hold the configurations for a particular Web service deployed in the Axis2 engine. When we generate the skeleton using the WSDL2Java tool, it will also generate the required services.xml for this Web service as well. This can be found in the same directory as the skeleton. The generated services.xml is as follows.

<!-- This file was auto-generated from WSDL -->

<!-- by the Apache Axis2 version: #axisVersion# #today# -->

<serviceGroup>

<service name="Axis2SampleDocLitService">

<messageReceivers>

<messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"

class="org.apache.axis2.userguide.Axis2SampleDocLitServiceMessageReceiverInOut"/>

</messageReceivers>

<parameter locked="false" name="ServiceClass">

org.apache.axis2.userguide.Axis2SampleDocLitServiceSkeleton</parameter>

<operation name="echoStringArray" mep="http://www.w3.org/2004/08/wsdl/in-out">

<actionMapping>echoStringArray</actionMapping>

</operation>

<operation name="echoStruct" mep="http://www.w3.org/2004/08/wsdl/in-out">

<actionMapping>echoStruct</actionMapping>

</operation>

<operation name="echoString" mep="http://www.w3.org/2004/08/wsdl/in-out">

<actionMapping>echoString</actionMapping>

</operation>

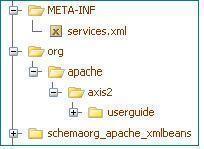
</service>

</serviceGroup>

First line of the "services.xml" gives the name of the Web Service. This is used in the URL to the service as the service name. Next comes the description and the service class. The next xml tags describe the operations that are available in this service with respective message receivers.

### **Packaging**

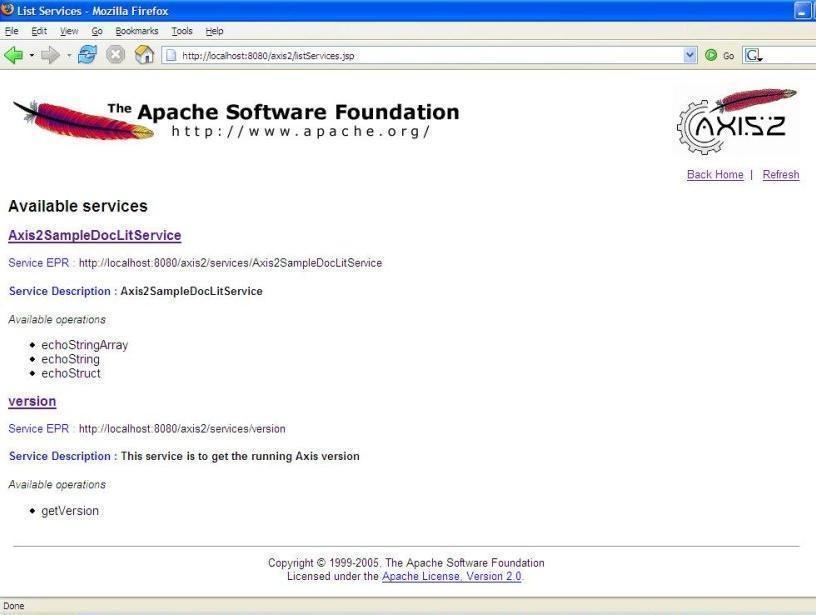
Next step in the process is to package the classes in a .aar (axis2 archive) and deploy it in Axis2. When the WSDL2Java tool generate the skeleton it will also generate the required data binding classes. These schema related classes are located in the **schemaorg\_apache\_xmlbeans** directory inside resources directory of the generated code. Copy this to your class path, compile the skeleton and the supporting classes. In order to create the .aar file, let's create the following directory structure with the required files and then simply use jar command to package it.



Go to the top level directory where you can find the class files for the above service (i.e. one level up on the directory structure shown above), then type the following command in a command line.

jar -cf Axis2SampleDocLitPortType.aar .

Deploying the service  is just a matter of dropping the ".aar" in to "services" directory that can be found in the "\webapps\axis2\WEB-INF" of your servlet container, hence copy the "echo.aar" into the "**services**" directory. Once these steps are completed, please start the servlet container (if you have not already started) and check the link "Services" on the [Home Page of Axis2 Web Application](http://window) (http://localhost:8080/axis2/index.jsp) and see whether the Axis2SampleDocLitPortType is deployed properly. If you can see the following output then you have successfully deployed Axis2SampleDocLitPortType on Axis2.



Note: Axis2 provides an easy way to deploy Web Services using the "Upload Service" tool on Axis2 Web Application's Administration module. (See the [Web Administration Guide](http://window) for more information on this)

## ***Web Service Clients Using Axis2***

Now let's see how we can write a Web Service Client to use this Web Service.

Web services can be used to provide wide range of functionality to the users ranging from simple, less time consuming  operations such as "getStockQuote"  to time consuming business services. When we utilize (invoke using client applications) these Web Service we cannot use some simple generic invocation paradigm that suites all the timing complexities involved in the service operations. For example, if we use a single transport channel (such as HTTP) to invoke a Web Service with and IN-OUT operation that take long time to complete, then most of the time we may end up with "connection time outs". On the other hand, if there are simultaneous service invocations that  we need to perform from a single client application, then the use of a "blocking" client API will degrade the performance of the client application. Similarly there are various other consequences such as One-Way transports that come in to play when we need them. Let's try to analyze some common service invocation paradigms.

Many Web service engines provide the users with a Blocking and Non-Blocking client APIs.

* **Blocking API** -Once the service invocation is called, the client application hangs and only gets control back when the operation completes, after which client receives a response or a fault. This is the simplest way of invoking Web Services and it also suites many business situations.
* **Non-Blocking API** - This is a callback or polling based API, hence once a service invocation is called, the client application immediately gets the control back and the response is retrieved using the callback object provided. This approach provides the flexibility to the client application to invoke several Web Services simultaneously without blocking the operation already invoked.

Both these mechanisms work in the API level. Let's name the  asynchronous behavior that we can get using the **Non-Blocking API** as **API Level Asynchrony.**

Both these mechanisms use single transport connection to send the request and to receive the response. They severely lags the capability of using two transport connections for the request and the response (either One-Way of Two-Way). So both these mechanisms fail to address the problem of long running transactions (the transport connection may time-out before the operation completes). A possible solution would be to use **two separate transport connections for request and response**. The asynchronous behavior that we gain using this solution can be called **Transport Level Asynchrony**.

By combining API Level Asynchrony & Transport Level Asynchrony we can obtain four different invocation patterns for Web services as shown in the following table.

|  |  |  |
| --- | --- | --- |
| **API (Blocking/Non-Blocking)** | **Dual Transports (Yes/No)** | **Description** |
| Blocking | No | Simplest and the familiar invocation pattern |
| Non-Blocking | No | Using callbacks or polling |
| Blocking | Yes | This is useful when the service operation is IN-OUT in nature but the transport used is One-Way (e.g. SMTP) |
| Non-Blocking | Yes | This is can be used to gain the maximum asynchronous behavior. No blocking in the API level and also in the transport level |

Axis2 provides the user with all these possibilities to invoke Web Services.

Below we describe how to write Web Services Clients using Axis2. This can be done in two methods:

1. Using the Axis2's primary APIs
2. Using stubs generated with data binding support, making the life easy for developers writing Web Service client applications

### **Writing Web Service Clients Using Axis2's Primary APIs**

### **EchoBlockingClient**

Axis2 provides the user with several invocation patterns for Web Services, ranging from pure blocking single channel invocations to a non-blocking dual channel invocations. Let's first see how we can write a client to invoke "echo" operation of "MyService" using the simplest blocking invocation. The client code you need to write is as follows.

try {

OMElement payload = ClientUtil.getEchoOMElement();

Options options = new Options();

options.setTo(targetEPR); // this sets the location of MyService service

ServiceClient serviceClient = new ServiceClient();

serviceClient.setOptions(options);

OMElement result = sender.sendReceive(payload);

System.out.println(result);

} catch (AxisFault axisFault) {

axisFault.printStackTrace();

}

}

The green lines shows the set of operations that you need to perform in order to invoke a Web service. The rest is used to create the OMElement that needs to be sent and display the response OMElement. To test this client, use the provided ant build file that can be found in the "Axis2Home/samples/userguide" directory. Run the "testEchoBlockingClient" target . If you can see the response OMElement printed in your command line,  then you have successfully tested the client.

### **PingClient**

In the Web Service "MyService" we had an IN-ONLY operation with the name "ping" (see [Web Services Using Axis2](http://ws.apache.org/axis2/1_0/userguide2.html#Web_Services_Using_Axis2)). Let's write a client to invoke this operation. The client code is as follows:

try {

OMElement payload = ClientUtil.getPingOMElement ();

Options options = new Options();

options.setTo (targetEPR);

ServiceClient serviceClient = new ServiceClient();

serviceClient.setOptions (options);

serviceClient.fireAndForget (payload);

/\*\*

\* We have to block this thread untill we send the request , the problem

\* is if we go out of the main thread , then request wont send ,so

\* you have to wait some time :)

\*/

Thread.sleep(500);

}

catch (AxisFault axisFault) {

axisFault.printStackTrace();

}

Since we are accessing an IN-ONLY operation we can directly use the "fireAndForget()" in ServiceClient to invoke this operation , and that will not block the invocation, hence it will return the control immediately back to the client. You can test this client by running the target "testPingClient" of the ant build file at "Axis2Home/samples/userguide".

We have invoked the two operations in our service. Are we done? No! There are lot more to explore. Let's see some other ways to invoke the same operations...

### **EchoNonBlockingClient**

In the EchoBlockingClient once the "serviceClient.sendReceive(payload);" is called, the client is blocked till the operation is completed. This behavior is not desirable when there are many Web service invocations to be done in a single client application or within a GUI. A solution would be to use a Non-Blocking API to invoke Web services. Axis2 provides a callback based non-blocking API for users.

A sample client for this can be found under "Axis2Home/samples/userguide/src/userguide/clients" with the name EchoNonBlockingClient. If we consider the changes that user may have to do with respect to the "EchoBlockingClient" that we have already seen, it will be as follows:

serviceClient.sendReceiveNonblocking(payload, callback);

The invocation accepts a callback object as a parameter. Axis2 client API provides an abstract Callback with the following methods:

public abstract void onComplete(AsyncResult result);

public abstract void onError(Exception e);

public boolean isComplete() {}

The user is expected to implement the "onComplete " and "onError " methods of their extended call back class. Axis2 engine calls the onComplete method once the Web Service response is received by the Axis2 Client API (ServiceClient). This will eliminate the blocking nature of the Web Service invocations and provides the user with the flexibility to use Non Blocking API for Web Service Clients.

To run the sample client ( EchoNonBlockingClient) you can simply use the "testEchoNonBlockingClient" target of the ant file found at the "Axis2Home/samples" directory.

### **EchoNonBlockingDualClient**

The solution provided by the Non-Blocking API has one limitation when it comes to  Web Service invocations which takes long time to complete. The limitation is due to the use of single transport connection to invoke the Web Service and to retrieve the response. In other words, client API provides a non blocking invocation mechanism for the users, but the request and the response comes in a single transport (Two-Way transport) connection (like HTTP). Long running Web Service invocations or Web Service invocations using One-Way transports (like SMTP) cannot be utilized by simply using a non blocking invocation.

The trivial solution is to use separate transport connections (either One-Way or Two-Way) for the request and response. The next problem that needs to be solved is the correlation (correlating the request and the response). [WS-Addressing](http://window) provides a neat solution to this using <wsa:MessageID> and <wsa:RelatesTo> headers. Axis2 provides support for addressing  based correlation mechanism and a complying Client API to invoke Web Services with two transport connections. (Core of Axis2 does not depend on WS-Addressing, but contains a set of parameters like in addressing that can be populated in any means. WS-Addressing is one of the users that may populate them. Even the transports can populate these. Hence Axis2 has the flexibility to use different versions of addressing)

Users can select between Blocking or Non-Blocking APIs for the Web Service clients with two transport connections. By simply using a boolean flag, the same API can be used to invoke Web services (IN-OUT operations) using two separate transport connections. Let's see how it's done using an example. Following code fragment shows how to invoke the same "echo" operation using Non-Blocking API with two transport connections**. The ultimate asynchrony!!**

try {

OMElement payload = ClientUtil.getEchoOMElement();

Options options = new Options();

options.setTo(targetEPR);

options.setTransportInProtocol(Constants.TRANSPORT\_HTTP);

options.setUseSeparateListener(true);

options.setAction("urn:echo"); // this is the action mapping we put within the service.xml

//Callback to handle the response

Callback callback = new Callback() {

public void onComplete(AsyncResult result) {

System.out.println(result.getResponseEnvelope());

}

public void onError(Exception e) {

e.printStackTrace();

}

};

//Non-Blocking Invocation

sender = new ServiceClient();

sender.engageModule(new QName(Constants.MODULE\_ADDRESSING));

sender.setOptions(options);

sender.sendReceiveNonBlocking(payload, callback);

//Wait till the callback receives the response.

while (!callback.isComplete()) {

Thread.sleep(1000);

}

//Need to close the Client Side Listener.

} catch (AxisFault axisFault) {

axisFault.printStackTrace();

} catch (Exception ex) {

ex.printStackTrace();

} finally {

try {

sender.finalizeInvoke();

} catch (AxisFault axisFault) {

//have to ignore this

}

}

The boolean flag (value true) in the "**options.setUseSeparateListener(...)**" method informs the Axis2 engine to use separate transport connections for request and response. Finally "**serviceClient.finalizeInvoke()**" informs the Axis2 engine to stop the client side listener started to retrieve the response.

Before we run the sample client we have one more step to perform. As mentioned earlier Axis2 uses addressing based correlation mechanism, hence we need to "engage" addressing module in both client and server sides.

### **Engaging Addressing in Server Side**

According to the Axis2 architecture, addressing module put its handlers in the "**pre-dispatch**" phase (See [Architecture Guide](http://window) for more details about phases)  and hence "engaging" means simply adding module reference in the "axis2.xml" (NOT the "services.xml"). Now add the following line to the "axis2.xml" that you can find in the "/webapps/axis2/WEB-INF" directory in the servlet container.

<module ref="addressing"/>

Note: Once you change the "axis2.xml" you need to restart the servlet container.

### **Engaging Addressing in Client Side**

There are two ways of doing that.

One is to get the addressing-<version>.mar from modules folder of the std-bin distribution. And then making that available in your classpath.

The second method is to create a ConfigurationContext giving a repository location. Axis2 has the concept of a repository to keep the services and modules. You can use the extracted standard binary distribution itself as the repository as it contains the proper structure of an Axis2 repository (having services and modules folders inside it). ConfigurationContext has the runtime context information of Axis2 system.

If you have extracted the standard binary distribution to, say, $user\_home/axis2/dist, then put the following line just before sender = new ServiceClient();

ConfigurationContext configContext = ConfigurationContextFactory.createConfigurationContextFromFileSystem(< Axis2RepositoryLocation >, null);

Then replace "sender = new ServiceClient();" line with "sender = new ServiceClient(configContext, null);"

This will enable addressing in both client and server sides. Now you can test the "TestEchoNonBlockingDualClient" using the "testEchoNonBlockingDualClient" target of the ant file found at "Axis2Home/samples/userguide" directory. If you see the response OMElement printed in the client side, then you have successfully tested the Non Blocking API with two transport channels at the client side.

### **EchoBlockingDualClient**

This is again a Two-Way transport request/response client, but this time, we use a Blocking API in the client code. Sample code for this can be found in the "Axis2Home/samples/userguide/src/userguide/clients/" directory and the explanation is similar to the EchoNonBlockingDualClient, except that here we do not use a callback object to handle response. This is a very useful mechanism when the service invocation is IN-OUT in nature and the transports are One-Way (e.g. SMTP). For the sample client we use two HTTP connections for request and response. User can test this client using the "echoBlockingDualClient" target of the ant build file found in the "Axis2Home/samples/userguide" directory.

See [Configuring Transports](http://window) for use different transports.

### **Writing Web Service Clients using Code Generation with Data Binding Support**

Axis2 provides the data binding support for Web Service client as well. The user can generate the required stubs from a given WSDL with the other supporting classes. Let's generate stubs for the WSDL used earlier to generate the skeleton for the "Axis2SampleDocLitPortType". Simply run the WSDL2Java tool that can be found in the bin directory of the Axis2 distribution using the following command:

WSDL2Java -uri ..\samples\wsdl\Axis2SampleDocLit.wsdl -o ..\samples\src -p org.apache.axis2.userguide

This will generate the required stub "Axis2SampleDocLitPortTypeStub.java" that can be used to invoke the Web Service Axis2SampleDocLitPortType. Let's see how we can use this stub to write Web Service clients to utilize the Web Service Axis2SampleDocLitPortType (the service that we have already deployed).

### **Client for echoVoid Operation**

Following code fragment shows the necessary code for utilizing the echoVoid operation of the Axis2SampleDocLitPortType that we have already deployed. In this operation, a blank SOAP body element is sent to the Web Service and the same SOAP envelope is echoed back.

try {

//Create the stub by passing the AXIS\_HOME and target EPR.

//We pass null to the AXIS\_HOME and hence the stub will use the current directory as the AXIS\_HOME

Axis2SampleDocLitPortTypeStub stub = new Axis2SampleDocLitPortTypeStub(null,

"http://localhost:8080/axis2/services/Axis2SampleDocLitPortType");

stub.echoVoid();

} catch (Exception e) {

e.printStackTrace();

}

### **Client for echoString Operation**

Following code fragment shows the necessary code for utilizing the echoString operation of the Axis2SampleDocLitPortType that we have already deployed. The code is very simple to understand and the explanations are in the form of comments.

try {

//Create the stub by passing the AXIS\_HOME and target EPR.

//We pass null to the AXIS\_HOME and hence the stub will use the current directory as the AXIS\_HOME

Axis2SampleDocLitPortTypeStub stub= new Axis2SampleDocLitPortTypeStub(null,

"http://localhost:8080/axis2/services/Axis2SampleDocLitPortType");

//Create the request document to be sent.

EchoStringParamDocument reqDoc= EchoStringParamDocument.Factory.newInstance();

reqDoc.setEchoStringParam("Axis2 Echo");

//invokes the Web service.

EchoStringReturnDocument resDoc=stub.echoString(reqDoc);

System.out.println(resDoc.getEchoStringReturn());

} catch (Exception e) {

e.printStackTrace();

}

Similarly following code fragments show client side code for echoStringArray operation and echoStruct operation respectively.

### **Client for echoStringArray Operation**

try {

//Create the stub by passing the AXIS\_HOME and target EPR.

//We pass null to the AXIS\_HOME and hence the stub will use the current directory as the AXIS\_HOME

Axis2SampleDocLitPortTypeStub stub = new Axis2SampleDocLitPortTypeStub(null,

"http://localhost:8080/axis2/services/Axis2SampleDocLitPortType");

//Create the request document to be sent.

EchoStringArrayParamDocument reqDoc = EchoStringArrayParamDocument.Factory.newInstance();

ArrayOfstringLiteral paramArray = ArrayOfstringLiteral.Factory.newInstance();

paramArray.addString("Axis2");

paramArray.addString("Echo");

reqDoc.setEchoStringArrayParam(paramArray);

EchoStringArrayReturnDocument resDoc = stub.echoStringArray(reqDoc);

//Get the response params

String[] resParams = resDoc.getEchoStringArrayReturn().getStringArray();

for (int i = 0; i < resParams.length; i++) {

System.out.println(resParams[i]);

}

} catch (Exception e) {

e.printStackTrace();

}

### **Client for echoStruct Operation**

try {

//Create the stub by passing the AXIS\_HOME and target EPR.

//We pass null to the AXIS\_HOME and hence the stub will use the current directory as the AXIS\_HOME

Axis2SampleDocLitPortTypeStub stub = new Axis2SampleDocLitPortTypeStub(null,

"http://localhost:8080/axis2/services/Axis2SampleDocLitPortType");

//Create the request Document

EchoStructParamDocument reqDoc = EchoStructParamDocument.Factory.newInstance();

//Create the complex type

SOAPStruct reqStruct = SOAPStruct.Factory.newInstance();

reqStruct.setVarFloat(100.50F);

reqStruct.setVarInt(10);

reqStruct.setVarString("High");

reqDoc.setEchoStructParam(reqStruct);

//Service invocation

EchoStructReturnDocument resDoc = stub.echoStruct(reqDoc);

SOAPStruct resStruct = resDoc.getEchoStructReturn();

System.out.println("floot Value :" + resStruct.getVarFloat());

System.out.println("int Value :" + resStruct.getVarInt());

System.out.println("String Value :" + resStruct.getVarString());

} catch (Exception e) {

e.printStackTrace();

}

#### **Explain each web service technologies - SOAP, WSDL, UDDI, eBXML and JAX pack.**

**SOAP:** Simple Object Access Protocol is a protocol that is used to exchange structured information at the time of implementing a web service. SOAP is relied on XML. Message format of SOAP usually relies on another protocol of different application layers. Among these the most notable application layer is Remote Procedure Call and HTTP. SOAP forms the foundation layer for web services protocol stack. This stack provides the basic framework for messaging on which the web services are built.

**WSDL:** Web Service Definition Language is used to describe a web service based on XML. WSDL is used for describing Web Services and to locate the services. WSDL consists of the information on what the service is all about, its residing location and the way of invocation the service.

**UDDI Universal Discovery Description :** To publish and discover the information about web services, UDDI is a specification. It is an XML based standard. This standard is used for describing, publishing, and finding services. These services are found in a distributed environment through the use of a server called registry server.

**Electronic Business using XML:** EBXML is one from XML family that is based on the standards of OASIS and UN/CEFACT. The mission of this standard is to provide an open xml-based infrastructure which could enable the global use of e-business in an interoperable, secure and consistent manner by all of the partners of trading. This is a unique architecture with unique concepts that are part theory and part implemented within the existing EBXML standards.

**JAX PACK:** A java API for xml pack that integrates all of the programming interfaces by SUN for different web services development. All these interfaces are made as a single download. JAX PACK is a bundle of JAXB, JAXM, JAX-RPC, and JAXR. Jax pack includes the documentations for support for the SAX, DOM.SOAP, WSDL, XSLT, EBXML, UDDI standards.

#### **Explain the technologies included within JAX pack, i.e. JAXP, JAXB, JAXM, JAX-RPC, JAXR.**

**JAXP:** Java API for xml processing. It provides the validation capability and parsing XML documents. There are three basic parsing interfaces in JAXP are DOM, SAX and Streaming API for XML STAX.

**JAXB:** Java Architecture for XML Binding: The java classes are mapped to XML representations. The two main features of JAXB are the ability to marshal Java objects into XML and unmarshal XML back to Java objects.

The Java API for XML Messaging (JAXM) enables distributed software applications to communicate using XML (and SOAP). JAXM supports both asynchronous and synchronous messaging.

**JAX-RPC:** Java API for XML based RPC. Allows a java based web service that is to be invoked by a Java application provided the description, still being consistent with WSDL description. This can resemble as Java RMI over web services.

Allowing a web service to be implemented at server side as a servlet/jsp or EJB container is the advantage of JAX-RPC.

#### **Explain the web services architecture.**

The operations between different software applications, which are running on a variety of platforms and frameworks are supported by a standard called Web services. The web services architecture provides the concepts, model and understanding web services and relationships among the components.

The WSA specifies the minimal characteristics that are very common for all web services and a number of characteristics to the needed web services. WSA is called interoperability architecture that means the global elements of a global web service network are identified by this architecture in order to perform the interoperability between the web services.

#### **What are smart web services?**

A smart web service understands the situational context and capable of sharing the context with other services. The result of this web service is based on the needs like who, what, when, where it was called.

The customer’s identity, role of a customer, security policies, physical location of a customer, type of client device are some of the number of the situational circumstances of that a smart web service is aware of.

**SOAP: Simple Object Oriented Protocol**

It is a stateless, platform independent, XML based generic lightweight protocol that uses HTTP as its transport medium and can be used for developing distributed complex computing environments.

Application can communicate directly with each other over the internet using SOAP.

It allows exchange of data between heterogeneous web applications.   
SOAP supports RPC like the DCOM or CORBA but it uses the XML Open Standard for the purpose of data exchange between homogenous or heterogeneous distributed applications.

Because of its platform independence, language independence and the usage og message communication, the SOAP protocol is robust and standardized mechanism over homogenous or heterogeneous networks.   
In the Remote Procedure Call of SOAP, the client sends a request message to the server. The server in turn sends the response message to the client.

XML is the foundation of the SOAP activities. All SOAP messages are transmitted in XML format and is a standard for the representation and interchange of data in structured form across systems.

#### **Why is SOAP required?**

The protocols like DCOM, RPC, IIOP are limited to a homogenous network. However, distributed applications comprise of a heterogeneous network components which necessitates the information to be transferred in common data format across the platforms.

### **Explain JAXR**

**Answer**JARX is a standard API that are used to access XML registries (list of services available on the web) from the JAVA platform. Client application can use JARX API to query the registries. It acts as a pluggable layer that allows access to registries implemented on different standards such as UDDI.

### **Explain JAX-RPC.**

**Answer**JAX-RPC uses SOAP to call remote procedures. JAX-RPC enables JAX-RPC clients to invoke web services developed across heterogeneous platform

### **Define UDDI, DISCO and WSDL.**

**Answer**UDDI, Universal description, discovery and integration  
It is the directory that is used to publish and discover public web services.

DISCO, Discovery  
Commonly known as Discovery. Discovery clubs together common services and exposes schema document of the web services.

WSDL, Web Service description language.  
This is used to describe web services. The description includes  
URL of web services  
Method and properties supported by web services  
Data type it supports.  
Protocol detail it supports.

### **What are the steps to get a proxy object of a web service at the client side?**

**Answer**Following are the steps to get a proxy object of a web service at the client side.

Access UDDI node for a list of web services.  
Services thus responded by UDDI have URL pointing to DISCO or WSDL document.  
Parse DISCO and WSDL document and build a proxy object which can communicate with the web service

### **Explain JAXM messaging models.**

**Answer**JAXM messaging models has two types of messaging model, synchronous and asynchronous.

Synchronous messaging model  
In this type of model, client directly interacts with the source. The client sends a request and waits for the response.

Asynchronous messaging model  
In this model, client sends message to the messaging provider and returns back. Messaging provider then performs the routing of message to the end source.

Simple Object Oriented Protocol (SOAP) is an Open Standard protocol and used XMl which has already been accepted as a standard form of information transfer which makes the usage of SOAP a solution to the complexity.

#### **Advantages of SOAP**

The following are some of the many advantages that SOAP provides.

* Language neutrality:   
  SOAP can be developed using any language.
* Interoperability and Platform Independence:  
  SOAP can be implemented in any language and can be executed in any platform.
* Simplicity:  
  SOAP messages are in very simple XML format.
* Scalability:   
  SOAP uses HTTP protocol for transport due to which it becomes scalable.

#### **Disadvantages of SOAP.**

The following are the disadvantages of SOAP.

* Slow  
  SOAP uses the XML format which needs to be parsed and is lengthier too which makes SOAP slower than CORBA, RMI or IIOP.
* WSDL Dependence  
  It depends on WSDL and does not have any standardized mechanism for dynamic discovery of the services.

#### **Composition of SOAP.**

As per the SOAP specification, SOAP is typically composed of the following three parts:

* A framework: It describes the way for the construction and processing of the messages.
* A set of encoding rules: These are used for exchanging the data types
* A convention and a procedure for representing the Remote Procedure Calls.

#### **What is SOAP? Explain its purpose.**

Simple Object Access Protocol is a XML based protocol that enables application to communicate with each other. SOAP has a standard format for sending messages. SOAP allows applications to communicate with each other over http, when different application running on different types of operating systems and using different technologies; SOAP can be used to enable communication between them.

#### **Give examples where SOAP is used.**

* Different application running on different types of operating systems and using different technologies.
* Example: To find company details, a SOAP request GetCompanyDetail() is sent to the server with the company id as the parameter. In response, details of company are returned via XML.

**SOAP request**

<?xml version="1.0"?>  
<soap:Envelope  
xmlns:soap=<http://www.w3.org/2001/12/soap-envelope>  
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">  
<soap:Body xmlns:m="http://www.example.org/ company ">  
<m: GetCompanyDetail >  
<m:CompanyID>1234></m:CompanyID >  
</m: GetCompanyDetail >  
</soap:Body>  
<soap:Envelope>

**SOAP response**

<?xml version="1.0"?>  
<soap:Envelope  
xmlns:soap=<http://www.w3.org/2001/12/soap-envelope>  
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">  
<soap:Body xmlns:m="http://www.example.org/company">  
<m: GetCompanyDetailResponse>  
<m:name>ABC</m:name>  
<m:revenue>20000</m:revenue>  
</m: GetCompanyDetailResponse >  
</soap:Body>  
</soap:Envelope>

#### **What are Transport methods in SOAP?**

* HTTP – The most common and preferred method of transport. HTTP is simple and universally accepted. The mechanism for sending a SOAP message over HTTP is the standard HTTP POST method. An HTTP POST sends a block of data to a particular URI on the web server.
* SMTP is also used as a transport medium.
* HTTPS may also be used for a secured communication.

#### **Explain the role of XML in SOAP.**

XML is used as a message format to communicate. Due to the open source nature of XML, it is widely accepted.

#### **What are the elements that should be contained in SOAP message?**

* Envelope – Translates the XML document to a SOAP message. It is the root element.
* Header – Contains the header message. Contains the application specific information.
* Body – Contains the call and response message.
* Fault element – Used for communicating errors. If present, it appears as a child element of the body and can appear only once.

#### **What is SOAP Envelope element?**

SOAP envelope element is the root element used to define the XML document as a SOAP message.

Example:

<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope" soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">  
...  
Message information goes here  
...  
</soap:Envelope>

#### **What is SOAP actor element?**

The header element may pass through different endpoints before it reaches the receiver. SOAP actor element is specifically used to address the header element to a specific endpoint.

Syntax:

Soap:actor=”URL”

#### **What is SOAP body element?**

SOAP body element is used to enclose the actual message. It is a required element that is enclosed within the Envelope tag.

Example - Request

<soap: Body>  
<m: GetCompanyDetail xmlns:m="http://www.w3schools.com/company ">  
<m:name>Apple</m:name>  
</m: GetCompanyDetail>  
</soap:Body>

#### **What do you mean by SOAP encoding?**

A SOAP message has no default encoding. Hence, in order to define data types used in the document encodingStyle attribute is used. It can appear in any SOAP element.

**Syntax:**

soap:encodingStyle="URI"

#### **Why Encrypt SOAP messages?**

The main purpose of SOAP is to exchange messages over HTTP. For communication it uses XML. The messages exchanged if done in plain text can be potentially viewed by anyone across the internet. SOAP over HTTPS is secured. The entire HTTP message, including both the headers and the body of the HTTP message is encrypted using public asymmetric encryption algorithms.

Web services are software applications that are based on a collection of industry standards and can be shared by and used as components of distributed Web-based applications. They can be identified by an URI and their public interfaces can be described and discovered using XML.

With the rising needs of interoperability and application integration, Web services serve as an optimal business solution because they are based on standard Internet protocols. Web services allow applications to process requests to remote and differential systems by speaking a common, non-proprietary language and using common transport protocols (such as HTTP and SMTP).

In this article, the focus will be on solving the previously mentioned problems by building inexpensive Web services using Axis, an open source SOAP implementation provided by Apache. For justification, the Web services build will be jax-rpc based, and in coming articles, I will deal with other methodologies, too.

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. This article elaborates how developers can use JAX-RPC to implement and call SOAP-based Web services described by the Web Services Description Language (WSDL) using Apache's open source tools—Apache Tomcat for deployment and Apache Axis for SOAP implementation.

### **JAX-RPC–Based Web Services**

JAX-RPC fully embraces the heterogeneous nature of Web services. It allows a JAX-RPC client to talk to another Web service deployed on a different platform and coded in a different language. *JAX-RPC* provides the specification for invocation modes, client generation, parameter modes, and type-mappings for Java to WSDL and WSDL to Java and client side APIs for invoking the Web service.

### **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.

In this article with above modes, we will see how the Java clients can be written or generated using synchronous request-response mode on a Tomcat-Axis combination.

### **Parameter Modes**

The Web service invocation based on *jax-rpc* uses pass-by-copy semantics for parameter passing. It does not support the pass-by-reference way of parameter passing.

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.

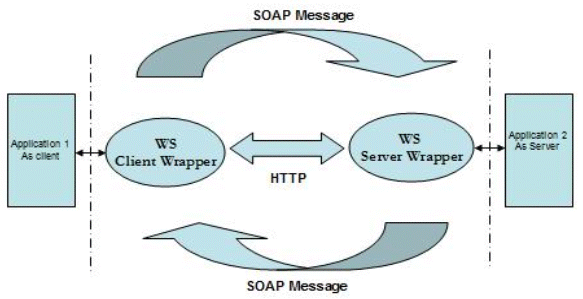
The parameter passing mode for *out* and *inout* parameters uses *Holder* classes. The use of Holder classes enables the mapping to preserve the intended *WSDL* signature and parameter passing semantics. The JAX-RPC specification includes "Holder classes" for the mapping of simple XML types to the Java 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.

Each *Holder* class provides the following methods and fields:

1. A public field named 'value'. The type of value is the mapped Java type.
2. A default constructor that initializes the value field to a default value.
3. A constructor that sets the value field to the passed parameter.

### **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.

 [*Click here for a larger image.*](http://www.developer.com/img/articles/2003/07/18/DevelopingJAX.gif)

**Figure 1. Web Services Invocation Architecture**

As of now, we understood the simple Web service invocation based on JAX-RPC. The next part of this article briefly develops a sample Web service, a dynamic client for it on Apache Tomcat with Axis.

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

### **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)

#### **What is Service-Oriented Architecture?**

SOA is an IT architecture strategy for business solution (and infrastructure solution) delivery based on the concept of service-orientation.

It is a set of components which can be invoked, and whose interface descriptions can be published and discovered.   
It aims at building systems that are extendible, flexible and fit with legacy systems.   
It promotes the re-use of basic components called services.

#### **Why SOA?**

Service is the important concept. Services can be published, discovered and used in a technology neutral, standard form by the set of protocols of the web services.

Other than being just architecture, SOA is the policies, practices, and frameworks by which it is ensure the right services are provided and consumed.

It becomes critical to implement processes that ensure that there are at least two different and separate processes— one for provider and the other for consumer, using SOA.

The Business Service Bus is starting point for developers that guide them to a coherent set that has been assembled for their domain.   
This is better than leaving developers to discover individual services and put them into context.

#### **Challenges faced in SOA adoption**

One of the challenges faced by SOA is managing services metadata.

Second biggest challenge is the lack of testing in SOA space.

Another challenge is providing appropriate levels of security.

Interoperability is another important aspect in the SOA implementations.

Vendor hype concerns SOA because it can create expectations that may not be fulfilled.

#### **What is SOA governance? What are its functions?**

Service-Oriented Architecture (SOA) governance is a concept used for activities related to exercising control over services in an SOA

Some key activities that are often mentioned as being part of SOA governance are:

* Managing the portfolio of services: This includes planning development of new services and updating current services.
* Managing the service lifecycle: This is meant to ensure that updates of services do not disturb current services to the consumers.
* Using policies to restrict behavior: Consistency of services can be ensured by having the rules applied to all the created services.
* Monitoring performance of services: The consequences of service downtime or underperformance can be severe because of service composition. Therefore action can be taken instantly when a problem occurs by monitoring service performance and availability.

#### **Business Benefits of Service-Oriented Architecture**

SOA can help businesses respond more quickly and economically to changing market conditions.

SOA can be considered an architectural evolution. It captures many of the best practices of previous software architectures.

The goal of separating users from the service implementations is promoted by SOA.

The goals like increased interoperability, increased federation and increased business & technology domain alignment can be achieved by SOA due to its architectural and design discipline.

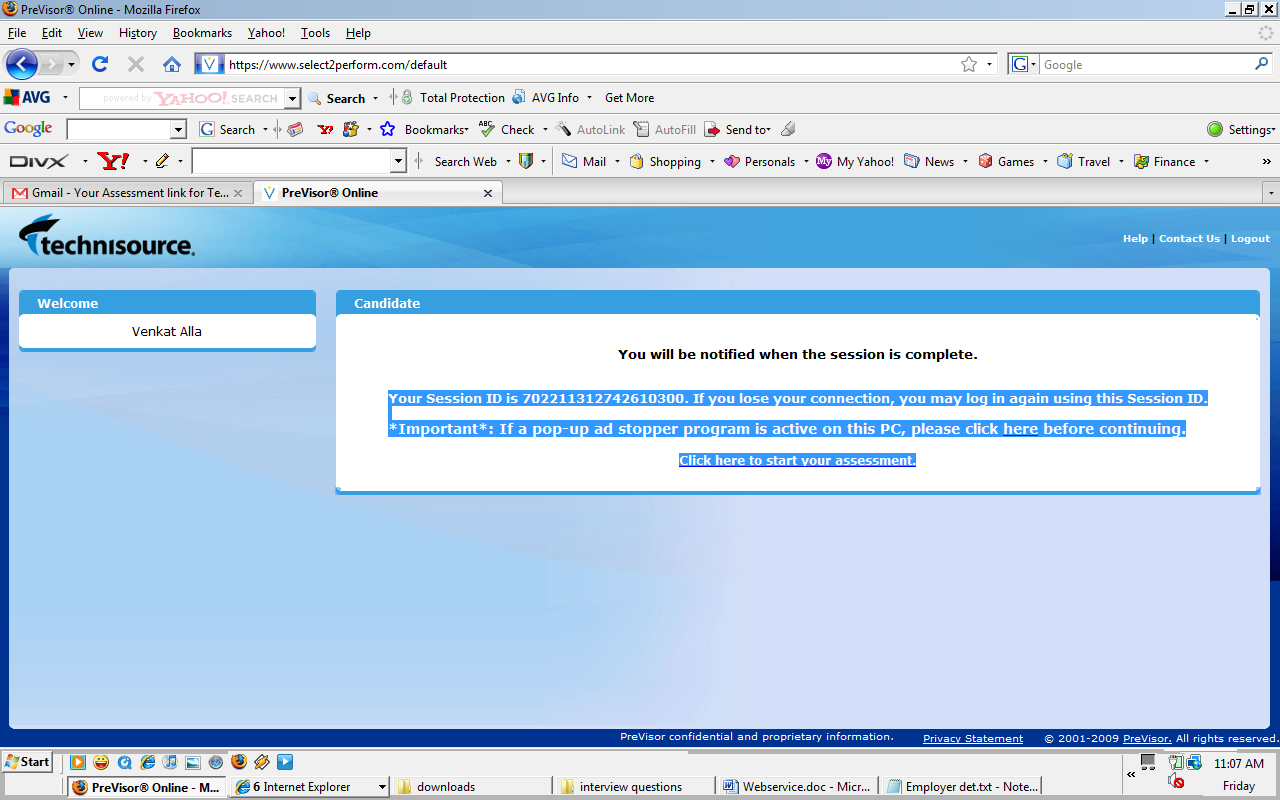
SOA is an architectural approach for constructing complex software-intensive systems from services.

SOA realizes its business and IT benefits through utilizing an analysis and design methodology when creating services.

#### **IT Benefits of Service-Oriented Architecture**

IT benefits of SOA are:

* The ability to build composite applications is provided.
* Business services are offered across the platforms.
* A self-healing infrastructure that reduces management costs is created.
* Location independence is provided
* Provides truly real-time decision-making applications.
* Reliability is enhanced
* It is not necessary that Services be at a particular system or network
* The approach is completely loosely coupled
* Hardware acquisition costs are reduced
* At every level there’s Authentication and authorization support
* Existing development skills are leveraged
* Provides a data bridge between incompatible technologies
* The search and connectivity to other services is dynamic



|  |
| --- |
| **You will be notified when the session is complete.** |

|  |
| --- |
| **Your Session ID is 702211312742610300. If you lose your connection, you may log in again using this Session ID.** |
|  |
| **\*Important\*: If a pop-up ad stopper program is active on this PC, please click** [**here**](http://support.select2perform.com/105662) **before continuing.** |