Axis User's Guide

1.2 Version
Feedback: axis-dev@ws.apache.org

Table of Contents

- Introduction
  - What is SOAP?
  - What is Axis?
  - What's in this release?
  - What's still to do?
- Installing Axis and Using this Guide
- Consuming Web Services with Axis
  - Basics - Getting Started
  - Naming Parameters
  - Interoperating with "untyped" servers
- Publishing Web Services with Axis
  - JWS (Java Web Service) Files - Instant Deployment
  - Custom Deployment - Introducing WSDD
  - Service Styles - RPC, Document, Wrapped, and Message
- XML <-> Java Data Mapping in Axis
  - How your Java types map to SOAP/XML types
  - Exceptions
  - What Axis can send via SOAP with restricted Interoperability
  - What Axis can not send via SOAP
  - Encoding Your Beans - the BeanSerializer
  - When Beans Are Not Enough - Custom Serialization
- Using WSDL with Axis
  - ?WSDL: Obtaining WSDL for deployed services
  - WSDL2Java: Building stubs, skeletons, and data types from WSDL
  - Java2WSDL: Building WSDL from Java
- Published Axis Interfaces
- Newbie Tips: Finding Your Way Around
  - Places to Look for Clues
  - Classes to Know
- Appendix : Using the Axis TCP Monitor (tcpmon)
- Appendix: Using the SOAP Monitor
- Glossary

Introduction

Welcome to Axis, the third generation of Apache SOAP!

What is SOAP?

SOAP is an XML-based communication protocol and encoding format for inter-application communication. Originally conceived by Microsoft and Userland software, it has evolved through several generations; the current spec is version, SOAP 1.2, though version 1.1 is more widespread. The W3C's XML Protocol working group is in charge of the specification.

SOAP is widely viewed as the backbone to a new generation of cross-platform cross-language distributed computing applications, termed Web Services.

What is Axis?

Axis is essentially a *SOAP engine* -- a framework for constructing SOAP processors such as clients, servers, gateways, etc. The current version of Axis is written in Java, but a C++ implementation of the client side of Axis is being developed.

But Axis isn't just a SOAP engine -- it also includes:

- a simple stand-alone server,
- a server which plugs into servlet engines such as Tomcat,

- extensive support for the *Web Service Description Language (WSDL)*,
- emitter tooling that generates Java classes from WSDL.
- some sample programs, and
- a tool for monitoring TCP/IP packets.

Axis is the third generation of Apache SOAP (which began at IBM as "SOAP4J"). In late 2000, the committers of Apache SOAP v2 began discussing how to make the engine much more flexible, configurable, and able to handle both SOAP and the upcoming XML Protocol specification from the W3C.

After a little while, it became clear that a ground-up rearchitecture was required. Several of the v2 committers proposed very similar designs, all based around configurable "chains" of message "handlers" which would implement small bits of functionality in a very flexible and composable manner.

After months of continued discussion and coding effort in this direction, 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, 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.

We hope you enjoy using Axis. Please note that this is an open-source effort - if you feel the code could use some new features or fixes, please get involved and lend a hand! The Axis developer community welcomes your participation. And in case you're wondering what *Axis* stands for, it's **A**pache E**X**tensible **I**nteraction **S**ystem - a fancy way of implying it's a very configurable SOAP engine.

Let us know what you think!

Please send feedback about the package to "axis-user@ws.apache.org". Also, Axis is registered in jira, the Apache bug tracking and feature-request database.

What's in this release?

This release includes the following features:

- SOAP 1.1/1.2 compliant engine
- Flexible configuration / deployment system
- Support for "drop-in" deployment of SOAP services (JWS)
- Support for all basic types, and a type mapping system for defining new serializers/deserializers
- Automatic serialization/deserialization of Java Beans, including customizable mapping of fields to XML elements/attributes
- Automatic two-way conversions between Java Collections and SOAP Arrays
- Providers for RPC and message based SOAP services
- Automatic WSDL generation from deployed services
- WSDL2Java tool for building Java proxies and skeletons from WSDL documents
- Java2WSDL tool for building WSDL from Java classes.
- Preliminary security extensions, which can integrate with Servlet 2.2 security/roles
- Support for session-oriented services, via HTTP cookies or transport-independent SOAP headers
- Preliminary support for the **SOAP with Attachments** specification
- An EJB provider for accessing EJB's as Web Services
- HTTP servlet-based transport
- JMS based transport
- Standalone version of the server (with HTTP support)
- Examples, including a client and server for the SoapBuilders community interoperability tests and experimental TCP, JMS, and file-based transports.

What's still to do?

Please click for a list of what we think needs doing - and please consider helping out if you're interested & able!

Installing Axis and Using this Guide

See the Axis Installation Guide for instructions on installing Axis as a web application on your J2EE server.

Before running the examples in this guide, you'll need to make sure that your CLASSPATH includes (Note: If you build

Axis from a CVS checkout, these will be in xml-axis/java/build/lib instead of axis-1_2/lib):

- axis-1_2/lib/axis.jar
- axis-1_2/lib/jaxrpc.jar
- axis-1_2/lib/saaj.jar
- axis-1_2/lib/commons-logging.jar
- axis-1_2/lib/commons-discovery.jar
- axis-1_2/lib/wsdl4j.jar
- axis-1_2/ *(for the sample code)*
- A JAXP-1.1 compliant XML parser such as Xerces or Crimson. We recommend Xerces, as it is the one that the product has been tested against.

Consuming Web Services with Axis

Basics - Getting Started

Let's take a look at an example Web Service client that will call the **echoString** method on the public Axis server at Apache.

```
1    import org.apache.axis.client.Call;
2    import org.apache.axis.client.Service;
3    import javax.xml.namespace.QName;
4
5    public class TestClient {
6      public static void main(String [] args) {
7        try {
8          String endpoint =
9              "http://ws.apache.org:5049/axis/services/echo";
10
11         Service  service = new Service();
12         Call     call    = (Call) service.createCall();
13
14         call.setTargetEndpointAddress( new java.net.URL(endpoint) );
15         call.setOperationName(new QName("http://soapinterop.org/", "echoString"));
16
17         String ret = (String) call.invoke( new Object[] { "Hello!" } );
18
19         System.out.println("Sent 'Hello!', got '" + ret + "'");
20       } catch (Exception e) {
21         System.err.println(e.toString());
22       }
23     }
24   }
```

(You'll find this file in samples/userguide/example1/TestClient.java)

Assuming you have a network connection active, this program can be run as follows:

```
% java samples.userguide.example1.TestClient
Sent 'Hello!', got 'Hello!'
%
```

So what's happening here? On lines 11 and 12 we create new Service and Call objects. These are the standard JAX-RPC objects that are used to store metadata about the service to invoke. On line 14, we set up our endpoint URL - this is the destination for our SOAP message. On line 15 we define the operation (method) name of the Web Service. And on line 17 we actually invoke the desired service, passing in an array of parameters - in this case just one String.

You can see what happens to the arguments by looking at the SOAP request that goes out on the wire (look at the colored sections, and notice they match the values in the call above):

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <ns1:echoString xmlns:ns1="http://soapinterop.org/">
      <arg0 xsi:type="xsd:string">Hello!</arg0>
    </ns1:echoString>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The String argument is automatically serialized into XML, and the server responds with an identical String, which we deserialize and print.

Note: To actually watch the XML flowing back and forth between a SOAP client and server, you can use the included tcpmon tool or SOAP monitor tool. See the appendix for an overview.

Naming Parameters

In the above example, you can see that Axis automatically names the XML-encoded arguments in the SOAP message "arg0", "arg1", etc. (In this case there's just "arg0") If you want to change this, it's easy! Before calling invoke() you need to call addParameter for each parameter and setReturnType for the return, like so:

```
call.addParameter("testParam",
  org.apache.axis.Constants.XSD_STRING,
  javax.xml.rpc.ParameterMode.IN);
call.setReturnType(org.apache.axis.Constants.XSD_STRING);
```

This will assign the name **testParam** to the 1st (and only) parameter on the invoke call. This will also define the type of the parameter (org.apache.axis.Constants.XSD_STRING) and whether it is an input, output or inout parameter - in this case its an input parameter. Now when you run the program you'll get a message that looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <ns1:echoString xmlns:ns1="http://soapinterop.org/">
      <testParam xsi:type="xsd:string">Hello!</testParam>
    </ns1:echoString>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Note that the param is now named "testParam" as expected.

Interoperating with "untyped" servers

In the above examples, we've been casting the return type of invoke(), which is Object, to the appropriate "real" type - for instance, we know that the echoString method returns a String, so we expect to get one back from client.invoke(). Let's take a moment and investigate how this happens, which sheds light on a potential problem (to which, of course, we have a solution - so don't fret :)).

Here's what a typical response might look like to the echoString method:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <ns1:echoStringResponse xmlns:ns1="http://soapinterop.org/">
      <result xsi:type="xsd:string">Hello!</result>
    </ns1:echoStringResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Take a look at the section which we've highlighted in red - that attribute is a schema **type declaration**, which Axis uses to figure out that the contents of that element are, in this case, deserializable into a Java String object. Many toolkits put this kind of explicit typing information in the XML to make the message "self-describing". On the other hand, some toolkits return responses that look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <ns1:echoStringResponse xmlns:ns1="http://soapinterop.org/">
      <result>Hello, I'm a string!</result>
    </ns1:echoStringResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

There's no type in the message, so how do we know what Java object we should deserialize the <result> element into? The answer is **metadata** - data about data. In this case, we need a **description** of the service that tells us what to expect as the return type. Here's how to do it on the client side in Axis:

```
call.setReturnType( org.apache.axis.Constants.XSD_STRING );
```

This method will tell the Axis client that if the return element is not typed then it should act as if the return value has an xsi:type attribute set to the predefined SOAP String type. (You can see an example of this in action in the interop echo-test client - samples/echo/TestClient.java.)

There is also a similar method which allows you to specify the Java class of the expected return type:

call.setReturnClass(String.class);

OK - so now you know the basics of accessing SOAP services as a client. But how do you publish your own services?

Publishing Web Services with Axis

Let's say we have a simple class like the following:

```
public class Calculator {
  public int add(int i1, int i2) {
    return i1 + i2;
  }

  public int subtract(int i1, int i2) {
    return i1 - i2;
  }
}
```

(You'll find this very class in samples/userguide/example2/Calculator.java.)

How do we go about making this class available via SOAP? There are a couple of answers to that question, but we'll start with the easiest way Axis provides to do this, which takes almost no effort at all!

JWS (Java Web Service) Files - Instant Deployment

OK, here's step 1 : copy the above .java file into your webapp directory, and rename it "Calculator.jws". So you might do something like this:

```
% copy Calculator.java <your-webapp-root>/axis/Calculator.jws
```

Now for step 2... hm, wait a minute. You're done! You should now be able to access the service at the following URL (assuming your Axis web application is on port 8080):

http://localhost:8080/axis/Calculator.jws

Axis automatically locates the file, compiles the class, and converts SOAP calls correctly into Java invocations of your service class. Try it out - there's a calculator client in samples/userguide/example2/CalcClient.java, which you can use like this:

```
% java samples.userguide.example2.CalcClient -p8080 add 2 5
Got result : 7
% java samples.userguide.example2.CalcClient -p8080 subtract 10 9
Got result : 1
%
```

(Note that you may need to replace the "-p8080" with whatever port your J2EE server is running on)

Important: JWS web services are intended for simple web services. You cannot use packages in the pages, and as the code is compiled at run time you can not find out about errors until after deployment. Production quality web services should use Java classes with custom deployment.

Custom Deployment - Introducing WSDD

JWS files are great quick ways to get your classes out there as Web Services, but they're not always the best choice. For one thing, you need the source code - there might be times when you want to expose a pre-existing class on your system without source. Also, the amount of configuration you can do as to how the service gets accessed is pretty limited - you can't specify custom type mappings, or control which Handlers get invoked when people are using your service. *(Note for the future : the Axis team, and the Java SOAP community at large, are thinking about ways to be able to embed this sort of metadata into your source files if desired - stay tuned!)*

Deploying via descriptors

To really use the flexibility available to you in Axis, you should get familiar with the Axis **Web Service Deployment Descriptor (WSDD)** format. A deployment descriptor contains a bunch of things you want to "deploy" into Axis - i.e. make available to the Axis engine. The most common thing to deploy is a Web Service, so let's start by taking a look at a deployment descriptor for a basic service (this file is samples/userguide/example3/deploy.wsdd):

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

```
  <service name="MyService" provider="java:RPC">
    <parameter name="className" value="samples.userguide.example3.MyService"/>
    <parameter name="allowedMethods" value="*"/>
  </service>
</deployment>
```

Pretty simple, really - the outermost element tells the engine that this is a WSDD deployment, and defines the "java" namespace. Then the service element actually defines the service for us. A service is a **targeted chain** (see the Architecture Guide), which means it may have any/all of: a request flow, a pivot Handler (which for a service is called a "provider"), and a response flow. In this case, our provider is "java:RPC", which is built into Axis, and indicates a Java RPC service. The actual class which handles this is **org.apache.axis.providers.java.RPCProvider**. We'll go into more detail later on the different styles of services and their providers.

We need to tell the RPCProvider that it should instantiate and call the correct class (e.g. samples.userguide.example3.MyService), and we do so by including <parameter> tags, giving the service one parameter to configure the class name, and another to tell the engine that any public method on that class may be called via SOAP (that's what the "*" means; we could also have restricted the SOAP-accessible methods by using a space or comma separated list of available method names).

Advanced WSDD - specifying more options

WSDD descriptors can also contain other information about services, and also other pieces of Axis called "Handlers" which we'll cover in a later section.

Scoped Services

Axis supports scoping service objects (the actual Java objects which implement your methods) three ways. "Request" scope, the default, will create a new object each time a SOAP request comes in for your service. "Application" scope will create a singleton shared object to service **all** requests. "Session" scope will create a new object for each session-enabled client who accesses your service. To specify the scope option, you add a <parameter> to your service like this (where "*value*" is request, session, or application):

```
<service name="MyService"...>
  <parameter name="scope" value="value"/>
  ...
</service>
```

Using the AdminClient

Once we have this file, we need to send it to an Axis server in order to actually deploy the described service. We do this with the AdminClient, or the "org.apache.axis.client.AdminClient" class. If you have deployed Axis on a server other than Tomcat, you may need to use the -p <*port*> argument. The default port is 8080. A typical invocation of the AdminClient looks like this:

```
% java org.apache.axis.client.AdminClient deploy.wsdd
<Admin>Done processing</Admin>
```

This command has now made our service accessible via SOAP. Check it out by running the Client class - it should look like this:

```
% java samples.userguide.example3.Client
  -lhttp://localhost:8080/axis/services/MyService "test me!"
You typed  :  test me!
%
```

If you want to prove to yourself that the deployment really worked, try undeploying the service and calling it again. There's an "undeploy.wsdd" file in the example3/ directory which you can use just as you did the deploy.wsdd file above. Run the AdminClient on that file, then try the service Client again and see what happens.

You can also use the AdminClient to get a listing of all the deployed components in the server:

```
% java org.apache.axis.client.AdminClient list
<big XML document returned here>
```

In there you'll see services, handlers, transports, etc. Note that this listing is an exact copy of the server's "server-config.wsdd" file, which we'll talk about in more detail a little later.

More deployment - Handlers and Chains

Now let's start to explore some of the more powerful features of the Axis engine. Let's say you want to track how many times your service has been called. We've included a sample handler in the samples/log directory to do just this. To use a handler class like this, you first need to deploy the Handler itself, and then use the name that you give it in deploying

a service. Here's a sample deploy.wsdd file (this is example 4 in samples/userguide):

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <!-- define the logging handler configuration -->
  <handler name="track" type="java:samples.userguide.example4.LogHandler">
    <parameter name="filename" value="MyService.log"/>
   </handler>

  <!-- define the service, using the log handler we just defined -->
  <service name="LogTestService" provider="java:RPC">
    <requestFlow>
      <handler type="track"/>
    </requestFlow>

    <parameter name="className" value="samples.userguide.example4.Service"/>
    <parameter name="allowedMethods" value="*"/>
  </service>
</deployment>
```

The first section defines a Handler called "track" that is implemented by the class samples.userguide.example4.LogHandler. We give this Handler an option to let it know which file to write its messages into.

Then we define a service, LogTestService, which is an RPC service just like we saw above in our first example. The difference is the <requestFlow> element inside the <service> - this indicates a set of Handlers that should be invoked when the service is invoked, before the provider. By inserting a reference to "track", we ensure that the message will be logged each time this service is invoked.

Remote Administration

Note that by default, the Axis server is configured to only accept administration requests from the machine on which it resides - if you wish to enable remote administration, you must set the "enableRemoteAdmin" property of the AdminService to **true**. To do this, find the "server-config.wsdd" file in your webapp's WEB-INF directory. In it, you'll see a deployment for the AdminService. Add an option as follows:

```
<service name="AdminService" provider="java:MSG">
  <parameter name="className" value="org.apache.axis.util.Admin"/>
  <parameter name="allowedMethods" value="*"/>
  <parameter name="enableRemoteAdmin" value="true"/>
</service>
```

WARNING: enabling remote administration may give unauthorized parties access to your machine. If you do this, please make sure to add security to your configuration!

Service Styles - RPC, Document, Wrapped, and Message

There are four "styles" of service in Axis. **RPC** services use the SOAP RPC conventions, and also the SOAP "section 5" encoding. **Document** services do not use any encoding (so in particular, you won't see multiref object serialization or SOAP-style arrays on the wire) but DO still do XML<->Java databinding. **Wrapped** services are just like document services, except that rather than binding the entire SOAP body into one big structure, they "unwrap" it into individual parameters. **Message** services receive and return arbitrary XML in the SOAP Envelope without any type mapping / data binding. If you want to work with the raw XML of the incoming and outgoing SOAP Envelopes, write a message service.

RPC services

RPC services are the default in Axis. They are what you get when you deploy services with <service ... provider="java:RPC"> or <service ... style="RPC">. RPC services follow the SOAP RPC and encoding rules, which means that the XML for an RPC service will look like the "echoString" example above - each RPC invocation is modeled as an outer element which matches the operation name, containing inner elements each of which maps to a parameter of the operation. Axis will deserialize XML into Java objects which can be fed to your service, and will serialize the returned Java object(s) from your service back into XML. Since RPC services default to the soap section 5 encoding rules, objects will be encoded via "multi-ref" serialization, which allows object graphs to be encoded. (See the SOAP spec for more on multi-ref serialization.)

Document / Wrapped services

Document services and wrapped services are similar in that neither uses the SOAP encoding for data; it's just plain old XML schema. In both cases, however, Axis still "binds" Java representations to the XML (see the databinding section for more), so you end up dealing with Java objects, not directly with XML constructs.

A good place to start in describing the difference between document and wrapped services is with a sample SOAP message containing a purchase order:

```
<soap:Envelope xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <soap:Body>
    <myNS:PurchaseOrder xmlns:myNS="http://commerce.com/PO">
      <item>SKU 1</item>
      <quantity>1</quantity>
      <description>Sushi Knife</description>
    </myNS:PurchaseOrder>
  </soap:Body>
</soap:Envelope>
```

The relevant schema for the PurchaseOrder looks like this:

```
<schema targetNamespace="http://commerce.com/PO">
  <complexType name="POType">
    <sequence>
      <element name="item" type="xsd:string"/>
      <element name="quantity" type="xsd:int"/>
      <element name="description" type="xsd:string"/>
    </sequence>
  </complexType>
  <element name="PurchaseOrder" type="POType"/>
</schema>
```

For a **document** style service, this would map to a method like this:

public void method(PurchaseOrder po)

In other words, the ENTIRE <PurchaseOrder> element would be handed to your method as a single bean with three fields inside it. On the other hand, for a **wrapped** style service, it would map to a method like this:

public void purchaseOrder(String item, int quantity, String description)

Note that in the "wrapped" case, the <PurchaseOrder> element is a "wrapper" (hence the name) which only serves to indicate the correct operation. The arguments to our method are what we find when we "unwrap" the outer element and take each of the inner ones as a parameter.

The document or wrapped style is indicated in WSDD as follows:

<service ... style="document"> for document style
<service ... style="wrapped"> for wrapped style

In most cases you won't need to worry about document or wrapped services if you are starting from a WSDL document (see below).

Message services

Finally, we arrive at "Message" style services, which should be used when you want Axis to step back and let your code at the actual XML instead of turning it into Java objects. There are four valid signatures for your message-style service methods:

public Element [] method(Element [] bodies);
public SOAPBodyElement [] method (SOAPBodyElement [] bodies);
public Document method(Document body);
public void method(SOAPEnvelope req, SOAPEnvelope resp);

The first two will pass your method arrays of either DOM Elements or SOAPBodyElements - the arrays will contain one element for each XML element inside the <soap:body> in the envelope.

The third signature will pass you a DOM Document representing the <soap:body>, and expects the same in return.

The fourth signature passes you two SOAPEnvelope objects representing the request and response messages. This is the signature to use if you need to look at or modify headers in your service method. Whatever you put into the response envelope will automatically be sent back to the caller when you return. Note that the response envelope may already contain headers which have been inserted by other Handlers.

Message Example

A sample message service can be found in samples/message/MessageService.java. The service class, MessageService, has one public method, echoElements, which matches the first of the three method signatures above:

public Element[] echoElements(Element [] elems)

The MsgProvider handler calls the method with an array of org.w3c.dom.Element objects that correspond to the

immediate children of the incoming message's SOAP Body. Often, this array will contain a single Element (perhaps the root element of some XML document conforming to some agreed-upon schema), but the SOAP Body can handle any number of children. The method returns an Element[] array to be returned in the SOAP body of the response message.

Message services must be deployed with a WSDD file. Here is the full WSDD for the MessageService class:

```
<deployment name="test" xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
    xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance">
  <service name="MessageService" style="message">
    <parameter name="className" value="samples.message.MessageService"/>
    <parameter name="allowedMethods" value="echoElements"/>
  </service>
</deployment>
```

Note that the "style" attribute is different from the RPC deployment example. The "message" style tells Axis that this service is to be handled by org.apache.axis.providers.java.MsgProvider rather than org.apache.axis.providers.java.RPCProvider.

You can test this service by deploying it, then running samples.message.TestMsg (look at the source to see what the test driver does).

XML <-> Java Data Mapping in Axis

How your Java types map to SOAP/XML types

Interoperability, *interop*, is an ongoing challenge between SOAP implementations. If you want your service to work with other platforms and implementations, you do need to understand the issues. There are some external articles on the subject that act as a good starting place. The basic mapping between Java types and WSDL/XSD/SOAP in Axis is determined by the JAX-RPC specification. Read chapters 4 and 5 of the specification to fully understand how things are converted. Here are some of the salient points.

Standard mappings from WSDL to Java

xsd:base64Binary	byte[]
xsd:boolean	boolean
xsd:byte	byte
xsd:dateTime	java.util.Calendar
xsd:decimal	java.math.BigDecimal
xsd:double	double
xsd:float	float
xsd:hexBinary	byte[]
xsd:int	int
xsd:integer	java.math.BigInteger
xsd:long	long
xsd:QName	javax.xml.namespace.QName
xsd:short	short
xsd:string	java.lang.String

If the WSDL says that an object can be nillable, that is the caller may choose to return a value of nil, then the primitive data types are replaced by their wrapper classes, such as Byte, Double, Boolean, etc.

SOAP Encoding Datatypes

Alongside the XSD datatypes are the SOAP 'Section 5' datatypes that are all nillable, and so only ever map to the wrapper classes. These types exist because they all support the "ID" and "HREF" attributes, and so will be used when in an RPC-encoded context to support multi-ref serialization.

Exceptions

This is an area which causes plenty of confusion, and indeed, the author of this section is not entirely sure how everything works, especially from an interop perspective. This means treat this section as incomplete and potentially inaccurate. See also section 5.5.5 and chapter 14 in the JAX-RPC specification

RemoteExceptions map to SOAP Faults

If the server method throws a java.rmi.RemoteException then this will be mapped into a SOAP Fault. The faultcode of this will contain the classname of the fault. The recipient is expected to deserialize the body of the fault against the classname.

Obviously, if the recipient does not know how to create an instance of the received fault, this mechanism does not work. Unless you include information about the exception class in the WSDL description of the service, or sender and receiver share the implementation, you can only reliably throw java.rmi.RemoteException instances, rather than subclasses.

When an implementation in another language receives such an exception, it should see the name of the class as the faultCode, but still be left to parse the body of the exception. You need to experiment to find out what happens there.

Exceptions are represented as wsdl:fault elements

If a method is marked as throwing an Exception that is not an instance or a subclass of java.rmi.RemoteException, then things are subtly different. The exception is no longer a SOAP Fault, but described as a wsdl:fault in the WSDL of the method. According to the JAX-RPC specification, your subclass of Exception must have accessor methods to access all the fields in the object to be marshalled *and* a constructor that takes as parameters all the same fields (i.e, arguments of the same name and type). This is a kind of immutable variant of a normal JavaBean. The fields in the object must be of the datatypes that can be reliably mapped into WSDL.

If your exception meets this specification, then the WSDL describing the method will describe the exception too, enabling callers to create stub implementations of the exception, regardless of platform.

Again, to be sure of interoperability, you need to be experiment a bit. Remember, the calling language may not have the notion of Exceptions, or at least not be as rigorous as Java in the rules as to how exceptions must be handled.

What Axis can send via SOAP with restricted Interoperability

Java Collections

Some of the Collection classes, such as Hashtable, do have serializers, but there is no formal interoperability with other SOAP implementations, and nothing in the SOAP specifications which covers complex objects. The most reliable way to send aggregate objects is to use arrays. In particular, .NET cannot handle them, though many Java SOAP implementations can marshall and unmarshall hash tables.

What Axis can not send via SOAP

Arbitrary Objects without Pre-Registration

You cannot send arbitrary Java objects over the wire and expect them to be understood at the far end. With RMI you can send and receive Serializable Java objects, but that is because you are running Java at both ends. **Axis will only send objects for which there is a registered Axis serializer.** This document shows below how to use the BeanSerializer to serialize any class that follows the JavaBean pattern of accessor and mutator. To serve up objects you must either register your classes with this BeanSerializer, or there must be serialization support built in to Axis.

Remote References

Remote references are neither part of the SOAP specification, nor the JAX-RPC specification. You cannot return some object reference and expect the caller to be able to use it as an endpoint for SOAP calls or as a parameter in other calls. Instead you must use some other reference mechanism, such as storing them in a HashMap with numeric or string keys that can be passed over the wire.

Encoding Your Beans - the BeanSerializer

Axis includes the ability to serialize/deserialize, without writing any code, arbitrary Java classes which follow the standard JavaBean pattern of get/set accessors. All you need to do is tell Axis which Java classes map to which XML Schema types. Configuring a bean mapping looks like this:

```
<beanMapping qname="ns:local" xmlns:ns="someNamespace"
    languageSpecificType="java:my.java.thingy"/>
```

The <beanMapping> tag maps a Java class (presumably a bean) to an XML QName. You'll note that it has two important attributes, **qname** and **languageSpecificType**. So in this case, we'd be mapping the "my.java.thingy" class to the XML QName [someNamespace]:[local].

Let's take a look at how this works in practice. Go look at samples/userguide/example5/BeanService.java. The key thing to notice is that the argument to the service method is an Order object. Since Order is not a basic type which Axis understands by default, trying to run this service without a type mapping will result in a fault. (If you want to try this for yourself, you can use the bad-deploy.wsdd file in the example5 directory.) But if we put a beanMapping into our

deployment, all will be well. Here's how to run this example (from the example5 directory):

```
% java org.apache.axis.client.AdminClient -llocal:///AdminService deploy.wsdd
<Admin>Done processing</Admin>

% java samples.userguide.example5.Client -llocal://
Hi, Glen Daniels!

You seem to have ordered the following:

1 of item : mp3jukebox
4 of item : 1600mahBattery

If this had been a real order processing system, we'd probably have charged
you about now.
%
```

When Beans Are Not Enough - Custom Serialization

Just as JWS deployment is sometimes not flexible enough to meet all needs, the default bean serialization model isn't robust enough to handle every case either. At times there will be non-bean Java classes (especially in the case of pre-existing assets) which you need to map to/from XML, and there also may be some custom XML schema types which you want to map into Java in particular ways. Axis gives you the ability to write custom serializers/deserializers, and some tools to help make your life easier when you do so.

TBD - this section will be expanded in a future version! For now look at the DataSer/DataDeser classes (in samples/encoding). Also look at the BeanSerializer, BeanDeserializer, ArraySerializer, ArrayDeserializer and other classes in the org.apache.axis.encoding.ser package.

Deploying custom mappings - the <typeMapping> tag

Now that you've built your serializers and deserializers, you need to tell Axis which types they should be used for. You do this with a typeMapping tag in WSDD, which looks like this:

```
<typeMapping qname="ns:local" xmlns:ns="someNamespace"
    languageSpecificType="java:my.java.thingy"
    serializer="my.java.Serializer"
    deserializer="my.java.DeserializerFactory"
    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
```

This looks a lot like the <beanMapping> tag we saw earlier, but there are three extra attributes. One, **serializer**, is the Java class name of the Serializer *factory* which gets the serializer to be used to marshal an object of the specified Java class (i.e., my.java.thingy) into XML. Another, **deserializer**, is the class name of a Deserializer *factory* that gets the deserializer to be used to unmarshall XML into the correct Java class. The final attribute, the **encodingStyle**, which is SOAP encoding.

(The <beanMapping> tag is really just shorthand for a <typeMapping> tag with serializer="org.apache.axis.encoding.ser.BeanSerializerFactory", deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory", and encodingStyle="http://schemas.xmlsoap.org/soap/encoding/", but clearly it can save a lot of typing!)

Deploying array mappings - the <arrayMapping> tag

nother variation around typeMapping is arrayMapping. The arrayMapping tag is useful for advanced users wanting to exatly control how their arrays are serialized throught the wire.

```
<arrayMapping qname="ns:ArrayOfthingy" xmlns:ns="someNamespaceURI"
              languageSpecificType="java:my.java.array.thingy[]"
                           innerType="ns2:thingy" xmlns:ns2="anotherNamespaceURI"
              encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
```

No need here to specify a serializer/deserializer couple, the arrayMapping tag is only about arrays (no List, ...). The added attribute (*innerType*) is used to tell Axis what precisely is the item type of the Array.

Using WSDL with Axis

The Web Service Description Language is a specification authored by IBM and Microsoft, and supported by many other organizations. WSDL serves to describe Web Services in a structured way. A WSDL description of a service tells us, in a machine-understandable way, the interface to the service, the data types it uses, and where the service is located. Please see the spec (follow the link in the first sentence) for details about WSDL's format and options.

Axis supports WSDL in three ways:

1. When you deploy a service in Axis, users may then access your service's URL with a standard web browser and by appending "?WSDL" to the end of the URL, they will obtain an automatically-generated WSDL document which describes your service.
2. We provide a "WSDL2Java" tool which will build Java proxies and skeletons for services with WSDL descriptions.
3. We provide a "Java2WSDL" tool which will build WSDL from Java classes.

?WSDL: Obtaining WSDL for deployed services

When you make a service available using Axis, there is typically a unique URL associated with that service. For JWS files, that URL is simply the path to the JWS file itself. For non-JWS services, this is usually the URL "http://<host>/axis/services/<service-name>".

If you access the service URL in a browser, you'll see a message indicating that the endpoint is an Axis service, and that you should usually access it using SOAP. However, if you tack on "?wsdl" to the end of the URL, Axis will automatically generate a service description for the deployed service, and return it as XML in your browser (try it!). The resulting description may be saved or used as input to proxy-generation, described next. You can give the WSDL-generation URL to your online partners, and they'll be able to use it to access your service with toolkits like .NET, SOAP::Lite, or any other software which supports using WSDL.

You can also generate WSDL files from existing Java classes (see Java2WSDL: Building WSDL from Java).

WSDL2Java: Building stubs, skeletons, and data types from WSDL

Client-side bindings

You'll find the Axis WSDL-to-Java tool in "org.apache.axis.wsdl.WSDL2Java". The basic invocation form looks like this:

```
% java org.apache.axis.wsdl.WSDL2Java (WSDL-file-URL)
```

This will generate only those bindings necessary for the client. Axis follows the JAX-RPC specification when generating Java client bindings from WSDL. For this discussion, assume we executed the following:

```
% cd samples/addr
% java org.apache.axis.wsdl.WSDL2Java AddressBook.wsdl
```

The generated files will reside in the directory "AddressFetcher2". They are put here because that is the target namespace from the WSDL and namespaces map to Java packages. Namespaces will be discussed in detail later.

WSDL clause	Java class(es) generated
For each entry in the type section	A java class
	A holder if this type is used as an inout/out parameter
For each portType	A java interface
For each binding	A stub class
For each service	A service interface
	A service implementation (the locator)

There is an Ant Task to integrate this action with an Ant based build process.

Types

The Java class generated from a WSDL type will be named from the WSDL type. This class will typically, though not always, be a bean. For example, given the WSDL (the WSDL used throughout the WSDL2Java discussion is from the Address Book sample):

```
<xsd:complexType name="phone">
  <xsd:all>
    <xsd:element name="areaCode" type="xsd:int"/>
    <xsd:element name="exchange" type="xsd:string"/>
    <xsd:element name="number" type="xsd:string"/>
  </xsd:all>
</xsd:complexType>
```

WSDL2Java will generate:

```
public class Phone implements java.io.Serializable {
  public Phone() {...}
  public int getAreaCode() {...}
  public void setAreaCode(int areaCode) {...}
  public java.lang.String getExchange() {...}
```

```
  public void setExchange(java.lang.String exchange) {...}
  public java.lang.String getNumber() {...}
  public void setNumber(java.lang.String number) {...}
  public boolean equals(Object obj) {...}
  public int hashCode() {...}
}
```

Mapping XML to Java types : Metadata

Notice in the mapping above, the XML type name is "phone" and the generated Java class is "Phone" - the capitalization of the first letter has changed to match the Java coding convention that classes begin with an uppercase letter. This sort of thing happens a lot, because the rules for expressing XML names/identifiers are much less restrictive than those for Java. For example, if one of the sub-elements in the "phone" type above was named "new", we couldn't just generate a Java field called "new", since that is a reserved word and the resultant source code would fail to compile.

To support this kind of mapping, and also to enable the serialization/deserialization of XML attributes, we have a **type metadata** system which allows us to associate Java data classes with descriptors which control these things.

When the WSDL2Java tool creates a data bean like the Phone class above, it notices if the schema contains any attributes, or any names which do not map directly to Java field/property names. If it finds any of these things, it will generate a static piece of code to supply a **type descriptor** for the class. The type descriptor is essentially a collection of **field descriptors**, each of which maps a Java field/property to an XML element or attribute.

To see an example of this kind of metadata, look at the "test.encoding.AttributeBean" class in the Axis source, or generate your own bean from XML which uses attributes or names which would be illegal in Java.

Holders

This type may be used as an inout or out parameter. Java does not have the concept of inout/out parameters. In order to achieve this behavior, JAX-RPC specifies the use of holder classes. A holder class is simply a class that contains an instance of its type. For example, the holder for the Phone class would be:

```
package samples.addr.holders;

public final class PhoneHolder implements javax.xml.rpc.holders.Holder {
  public samples.addr.Phone value;

  public PhoneHolder() {
  }

  public PhoneHolder(samples.addr.Phone value) {
    this.value = value;
  }
}
```

A holder class is **only** generated for a type if that type is used as an inout or out parameter. Note that the holder class has the suffix "Holder" appended to the class name, and it is generated in a sub-package with the "holders".

The holder classes for the primitive types can be found in javax.xml.rpc.holders.

PortTypes

The Service Definition Interface (SDI) is the interface that's derived from a WSDL's portType. This is the interface you use to access the operations on the service. For example, given the WSDL:

```
<message name="empty">
<message name="AddEntryRequest">
  <part name="name" type="xsd:string"/>
  <part name="address" type="typens:address"/>
</message>
<portType name="AddressBook">
  <operation name="addEntry">
    <input message="tns:AddEntryRequest"/>
    <output message="tns:empty"/>
  </operation>
</portType>
```

WSDL2Java will generate:

```
public interface AddressBook extends java.rmi.Remote {
  public void addEntry(String name, Address address) throws
```

```
        java.rmi.RemoteException;
}
```

A note about the name of the SDI. The name of the SDI is typically the name of the portType. However, to construct the SDI, WSDL2Java needs information from both the portType **and** the binding. (This is unfortunate and is a topic of discussion for WSDL version 2.)

JAX-RPC says (section 4.3.3): "The name of the Java interface is mapped from the name attribute of the wsdl:portType element. ... If the mapping to a service definition interface uses elements of the wsdl:binding ..., then the name of the service definition interface is mapped from the name of the wsdl:binding element."

Note the name of the spec. It contains the string "RPC". So this spec, and WSDL2Java, assumes that the interface generated from the portType is an RPC interface. If information from the binding tells us otherwise (in other words, we use elements of the wsdl:binding), then the name of the interface is derived instead from the binding.

Why? We could have one portType - pt - and two bindings - bRPC and bDoc. Since document/literal changes what the interface looks like, we cannot use a single interface for both of these bindings, so we end up with two interfaces - one named pt and another named bDoc - and two stubs - bRPCStub (which implements pt) and bDocStub (which implements bDoc).

Ugly, isn't it? But you can see why it's necessary. Since document/literal changes what the interface looks like, and we could have more than one binding referring to a single portType, we have to create more than one interface, and each interface must have a unique name.

Bindings

A Stub class implements the SDI. Its name is the binding name with the suffix "Stub". It contains the code which turns the method invocations into SOAP calls using the Axis Service and Call objects. It stands in as a **proxy** (another term for the same idea) for the remote service, letting you call it exactly as if it were a local object. In other words, you don't need to deal with the endpoint URL, namespace, or parameter arrays which are involved in dynamic invocation via the Service and Call objects. The stub hides all that work for you.

Given the following WSDL snippet:

```
<binding  name="AddressBookSOAPBinding" type="tns:AddressBook">
  ...
</binding>
```

WSDL2Java will generate:

```
public class AddressBookSOAPBindingStub extends org.apache.axis.client.Stub
    implements AddressBook {
  public AddressBookSOAPBindingStub() throws org.apache.axis.AxisFault {...}

  public AddressBookSOAPBindingStub(URL endpointURL,
      javax.xml.rpc.Service service) throws org.apache.axis.AxisFault {...}

  public AddressBookSOAPBindingStub(javax.xml.rpc.Service service)
      throws org.apache.axis.AxisFault {...}

  public void addEntry(String name, Address address)
      throws RemoteException {...}
}
```

Services

Normally, a client program would not instantiate a stub directly. It would instead instantiate a service locator and call a get method which returns a stub. This locator is derived from the service clause in the WSDL. WSDL2Java generates two objects from a service clause. For example, given the WSDL:

```
<service name="AddressBookService">
  <port name="AddressBook" binding="tns:AddressBookSOAPBinding">
    <soap:address location="http://localhost:8080/axis/services/AddressBook"/>
  </port>
</service>
```

WSDL2Java will generate the service interface:

```
public interface AddressBookService extends javax.xml.rpc.Service {
  public String getAddressBookAddress();

  public AddressBook getAddressBook() throws javax.xml.rpc.ServiceException;
```

```
  public AddressBook getAddressBook(URL portAddress)
      throws javax.xml.rpc.ServiceException;
}
```

WSDL2Java will also generate the locator which implements this interface:

```
public class AddressBookServiceLocator extends org.apache.axis.client.Service
    implements AddressBookService {
  ...
}
```

The service interface defines a get method for each port listed in the service element of the WSDL. The locator is the implementation of this service interface. It implements these get methods. It serves as a locator for obtaining Stub instances. The Service class will by default make a Stub which points to the endpoint URL described in the WSDL file, but you may also specify a different URL when you ask for the PortType.

A typical usage of the stub classes would be as follows:

```
public class Tester {
  public static void main(String [] args) throws Exception {
    // Make a service
    AddressBookService service = new AddressBookServiceLocator();

    // Now use the service to get a stub which implements the SDI.
    AddressBook port = service.getAddressBook();

    // Make the actual call
    Address address = new Address(...);
    port.addEntry("Russell Butek", address);
  }
}
```

Server-side bindings

Just as a stub is the client side of a Web Service represented in Java, a **skeleton** is a Java framework for the server side. To make skeleton classes, you just specify the "--server-side --skeletonDeploy true" options to WSDL2Java. For instance, using the AddressBook.wsdl as we had above:

```
% java org.apache.axis.wsdl.WSDL2Java --server-side
    --skeletonDeploy true AddressBook.wsdl
```

You will see that WSDL2Java generates all the classes that were generated before for the client, but it generates a few new files:

WSDL clause	Java class(es) generated
For each binding	A skeleton class
	An implementation template class
For all services	One deploy.wsdd file
	One undeploy.wsdd file

If you don't specify the "--skeletonDeploy true" option, a skeleton will not be generated. Instead, the generated deploy.wsdd will indicate that the implementation class is deployed directly. In such cases, the deploy.wsdd contains extra meta data describing the operations and parameters of the implementation class. Here is how you run WSDL2Java to deploy directly to the implementation:

```
% java org.apache.axis.wsdl.WSDL2Java --server-side AddressBook.wsdl
```

And here are the server side files that are generated:

WSDL clause	Java class(es) generated
For each binding	An implementation template class
For all services	One deploy.wsdd file with operation meta data
	One undeploy.wsdd file

Bindings

Skeleton Description (for Skeleton Deployment)

The skeleton class is the class that sits between the Axis engine and the actual service implementation. Its name is the binding name with suffix "Skeleton". For example, for the AddressBook binding, WSDL2Java will generate:

```
public class AddressBookSOAPBindingSkeleton implements AddressBook,
    org.apache.axis.wsdl.Skeleton {
  private AddressBook impl;

  public AddressBookSOAPBindingSkeleton() {
    this.impl = new AddressBookSOAPBindingImpl();
  }

  public AddressBookSOAPBindingSkeleton(AddressBook impl) {
    this.impl = impl;
  }

  public void addEntry(java.lang.String name, Address address)
      throws java.rmi.RemoteException {
    impl.addEntry(name, address);
  }
}
```

(The real skeleton is actually much richer. For brevity we just show you the basic skeleton.)

The skeleton contains an implementation of the AddressBook service. This implementation is either passed into the skeleton on construction, or an instance of the generated implementation is created. When the Axis engine calls the skeleton's addEntry method, it simply delegates the invocation to the real implementation's addEntry method.

Implementation Template Description

WSDL2Java also generates an implementation template from the binding:

```
public class AddressBookSOAPBindingImpl implements AddressBook {
  public void addEntry(String name, Address address)
      throws java.rmi.RemoteException {
  }
}
```

This template could actually be used as a test implementation but, as you can see, it doesn't do anything. It is intended that the service writer fill out the implementation from this template.

When WSDL2Java is asked to generate the implementation template (via the --server-side flag), it will ONLY generate it if it does not already exist. If this implementation already exists, it will not be overwritten.

Services

The tool also builds you a "deploy.wsdd" and an "undeploy.wsdd" for each service for use with the AdminClient. These files may be used to deploy the service once you've filled in the methods of the Implementation class, compiled the code, and made the classes available to your Axis engine.

Java2WSDL: Building WSDL from Java

The Java2WSDL and WSDL2Java emitters make it easy to develop a new web service. The following sections describe the steps in building a web service from a Java interface.

Step 1: Provide a Java interface or class

Write and compile a Java interface (or class) that describes the web service interface. Here is an example interface that describes a web services that can be used to set/query the price of widgets (samples/userguide/example6/WidgetPrice.java):

```
package samples.userguide.example6;

/**
 * Interface describing a web service to set and get Widget prices.
 **/
public interface WidgetPrice {
  public void setWidgetPrice(String widgetName, String price);
  public String getWidgetPrice(String widgetName);
}
```

Note: If you compile your class with debug information, Java2WSDL will use the debug information to obtain the method parameter names.

Step 2: Create WSDL using Java2WSDL

Use the Java2WSDL tool to create a WSDL file from the interface above.

Here is an example invocation that produces the wsdl file (wp.wsdl) from the interface described in the previous section:

```
% java org.apache.axis.wsdl.Java2WSDL -o wp.wsdl
    -l"http://localhost:8080/axis/services/WidgetPrice"
    -n "urn:Example6" -p"samples.userguide.example6" "urn:Example6"
    samples.userguide.example6.WidgetPrice
```

Where:

- -o indicates the name of the ***output WSDL*** file
- -l indicates the***location of the service***
- -n is the target ***namespace*** of the WSDL file
- -p indicates a mapping from the ***package to a namespace***. There may be multiple mappings.
- the class specified contains the interface of the webservice.

The output WSDL document will contain the appropriate WSDL types, messages, portType, bindings and service descriptions to support a SOAP rpc, encoding web service. If your specified interface methods reference other classes, the Java2WSDL tool will generate the appropriate xml types to represent the classes and any nested/inherited types. The tool supports JAX-RPC complex types (bean classes), extension classes, enumeration classes, arrays and Holder classes.

The Java2WSDL tool has many additional options which are detailed in the reference guide. There is an Ant Task to integrate this action with an Ant based build process.

Step 3: Create Bindings using WSDL2Java

Use the generated WSDL file to build the appropriate client/server bindings for the web service (see WSDL2Java):

```
% java org.apache.axis.wsdl.WSDL2Java -o . -d Session -s -S true
    -Nurn:Example6 samples.userguide.example6 wp.wsdl
```

This will generate the following files:

- **WidgetPriceSoapBindingImpl.java** : Java file containing the default server implementation of the WidgetPrice web service.
  You will need to modify the *SoapBindingImpl file to add your implementation (see samples/userguide/example6/WidgetPriceSoapBindingImpl.java).
- **WidgetPrice.java**: New interface file that contains the appropriate **java.rmi.Remote** usages.
- **WidgetPriceService.java**: Java file containing the client side service interface.
- **WidgetPriceServiceLocator.java**: Java file containing the client side service implementation class.
- **WidgetPriceSoapBindingSkeleton.java**: Server side skeleton.
- **WidgetPriceSoapBindingStub.java**: Client side stub.
- **deploy.wsdd**: Deployment descriptor
- **undeploy.wsdd**: Undeployment descriptor
- (data types): Java files will be produced for all of the other types and holders necessary for the web service. There are no additional files for the WidgetPrice web service.

Now you have all of the necessary files to build your client/server side code and deploy the web service!

Published Axis Interfaces

Although you may use any of the interfaces and classes present in Axis, you need to be aware that some are more stable than others since there is a continuing need to refactor Axis to maintain and improve its modularity.

Hence certain interfaces are designated as *published*, which means that they are relatively stable. As Axis is refactored, the Axis developers will try to avoid changing published interfaces unnecessarily and will certainly consider the impact on users of any modifications.

So if you stick to using only published interfaces, you'll minimise the pain of migrating between releases of Axis. On the other hand, if you decide to use unpublished interfaces, migrating between releases could be an interesting exercise! If you would like an interface to be published, you should make the case for this on the axis-user mailing list.

The current list of published interfaces is as follows:

- JAX-RPC interfaces. These interfaces are from JAX-RPC 1.0 specification, and will change only when new versions of the specification are released.
  - javax.xml.messaging.Endpoint
  - javax.xml.messaging.URLEndpoint
  - javax.xml.rpc.Call

    - javax.xml.rpc.FaultException
    - javax.xml.rpc.JAXRPCException
    - javax.xml.rpc.ParameterMode
    - javax.xml.rpc.Service
    - javax.xml.rpc.ServiceException
    - javax.xml.rpc.ServiceFactory
    - javax.xml.rpc.Stub
    - javax.xml.rpc.encoding.DeserializationContext
    - javax.xml.rpc.encoding.Deserializer
    - javax.xml.rpc.encoding.DeserializerFactory
    - javax.xml.rpc.encoding.SerializationContext
    - javax.xml.rpc.encoding.Serializer
    - javax.xml.rpc.encoding.SerializerFactory
    - javax.xml.rpc.encoding.TypeMapping
    - javax.xml.rpc.encoding.TypeMappingRegistry
    - javax.xml.rpc.handler.Handler
    - javax.xml.rpc.handler.HandlerChain
    - javax.xml.rpc.handler.HandlerInfo
    - javax.xml.rpc.handler.HandlerRegistry
    - javax.xml.rpc.handler.MessageContext
    - javax.xml.rpc.handler.soap.SOAPMessageContext
    - javax.xml.rpc.holders.BigDecimalHolder
    - javax.xml.rpc.holders.BigIntegerHolder
    - javax.xml.rpc.holders.BooleanHolder
    - javax.xml.rpc.holders.BooleanWrapperHolder
    - javax.xml.rpc.holders.ByteArrayHolder
    - javax.xml.rpc.holders.ByteHolder
    - javax.xml.rpc.holders.ByteWrapperArrayHolder
    - javax.xml.rpc.holders.ByteWrapperHolder
    - javax.xml.rpc.holders.CalendarHolder
    - javax.xml.rpc.holders.DateHolder
    - javax.xml.rpc.holders.DoubleHolder
    - javax.xml.rpc.holders.DoubleWrapperHolder
    - javax.xml.rpc.holders.FloatHolder
    - javax.xml.rpc.holders.FloatWrapperHolder
    - javax.xml.rpc.holders.Holder
    - javax.xml.rpc.holders.IntHolder
    - javax.xml.rpc.holders.IntegerWrapperHolder
    - javax.xml.rpc.holders.LongHolder
    - javax.xml.rpc.holders.LongWrapperHolder
    - javax.xml.rpc.holders.ObjectHolder
    - javax.xml.rpc.holders.QNameHolder
    - javax.xml.rpc.holders.ShortHolder
    - javax.xml.rpc.holders.ShortWrapperHolder
    - javax.xml.rpc.holders.StringHolder
    - javax.xml.rpc.namespace.QName
    - javax.xml.rpc.server.ServiceLifecycle
    - javax.xml.rpc.soap.SOAPFault
    - javax.xml.rpc.soap.SOAPHeaderFault
    - javax.xml.transform.Source
- Axis interfaces. These have less guarantees of stability:
    - org.apache.axis.AxisFault
    - org.apache.axis.Handler
    - org.apache.axis.DefaultEngineConfigurationFactory
    - org.apache.axis.EngineConfiguration
    - org.apache.axis.EngineConfigurationFactory
    - org.apache.axis.Message
    - org.apache.axis.MessageContext
    - org.apache.axis.SOAPPart
    - org.apache.axis.client.Call
    - org.apache.axis.client.Service
    - org.apache.axis.client.ServiceFactory
    - org.apache.axis.client.Stub
    - org.apache.axis.client.Transport
    - org.apache.axis.description.TypeDesc
    - org.apache.axis.description.AttributeDesc
    - org.apache.aixs.description.ElementDesc

- org.apache.axis.encoding.DeserializationContext
- org.apache.axis.encoding.Deserializer
- org.apache.axis.encoding.DeserializerFactory
- org.apache.axis.encoding.DeserializerTarget
- org.apache.axis.encoding.FieldTarget
- org.apache.axis.encoding.MethodTarget
- org.apache.axis.encoding.SerializationContext
- org.apache.axis.encoding.Serializer
- org.apache.axis.encoding.SerializerFactory
- org.apache.axis.encoding.SimpleType
- org.apache.axis.encoding.Target
- org.apache.axis.encoding.TypeMapping
- org.apache.axis.encoding.TypeMappingRegistry
- org.apache.axis.encoding.ser.BaseDeserializerFactory
- org.apache.axis.encoding.ser.BaseSerializerFactory
- org.apache.axis.encoding.ser.BeanPropertyTarget
- org.apache.axis.encoding.ser.SimpleSerializer
- org.apache.axis.encoding.ser.SimpleDeserializer
- org.apache.axis.session.Session
- org.apache.axis.transport.http.SimpleAxisServer
- org.apache.axis.transport.jms.SimpleJMSListener
- org.apache.axis.utils.BeanProperty
- org.apache.axis.wsdl.WSDL2Java
- org.apache.axis.wsdl.Java2WSDL

Newbie Tips: Finding Your Way Around

So you've skimmed the User's Guide and written your first .jws service, and everything went perfectly! Now it's time to get to work on a real project, and you have something specific you need to do that the User's Guide didn't cover. It's a simple thing, and you know it must be in Axis *somewhere*, but you don't know what it's called or how to get at it. This section is meant to give you some starting points for your search.

Places to Look for Clues

Here are the big categories.

- **The samples.** These examples are complete with deployment descriptors and often contain both client and server code.
- **The Javadocs.** Full Javadocs are included with the binary distribution. The Javadocs can be intimidating at first, but once you know the major user classes, they are one of the fastest ways to an answer.
- **The mailing list archives.** If you know what you want but don't know what it's called in Axis, this is the best place to look. Chances are someone has wanted the same thing and someone else has used (or developed) Axis long enough know the name.
- Consult the Axis web site for updated documentation and the Axis Wiki for its Frequently Asked Questions (FAQ), installation notes, interoperability issues lists, and other useful information.
- **WSDL2Java.** Point WSDL2Java at a known webservice that does some of the things you want to do. See what comes out. This is useful even if you will be writing the actual service or client from scratch. If you want nice, human-readable descriptions of existing web services, try http://www.xmethods.net.

Classes to Know

org.apache.axis.MessageContext

The answer to most "where do I find..." questions for an Axis web service is "in the MessageContext." Essentially everything Axis knows about a given request/response can be retrieved via the MessageContext. Here Axis stores:

- A reference to the AxisEngine
- The request and response messages (org.apache.axis.Message objects available via getter and setter methods)
- Information about statefulness and service scope (whether the service is maintaining session information, etc.)
- The current status of processing (whether or not the "pivot" has been passed, which determines whether the request or response is the current message)
- Authentication information (username and password, which can be provided by a servlet container or other means)
- Properties galore. Almost anything you would want to know about the message can be retrieved via MessageContext.getProperty(). You only need to know the name of the property. This can be tricky, but it is usually a constant, like those defined in org.apache.axis.transport.http.HTTPConstants. So, for example, to retrieve the ServletContext for the Axis Servlet, you would want:
  ((HttpServlet)msgC.getProperty(HTTPConstants.MC_HTTP_SERVLET)).getServletContext();

From within your service, the current MessageContext object is always available via the static method MessageContext.getCurrentContext(). This allows you to do any needed customization of the request and response

methods, even from within an RPC service that has no explicit reference to the MessageContext.

org.apache.axis.Message

An org.apache.axis.Message object is Axis's representation of a SOAP message. The request and response messages can be retrieved from the MessageContext as described above. A Message has:

- MIME headers (if the message itself has MIME information)
- Attachments (if the message itself has attachments)
- A SOAPPart (and a convenience method for quick retrieval of the SOAPPart's SOAPEnvelope). The SOAPPart gives you access to the SOAP "guts" of the message (everything inside the <soap:Envelope> tags)

org.apache.axis.SOAPEnvelope

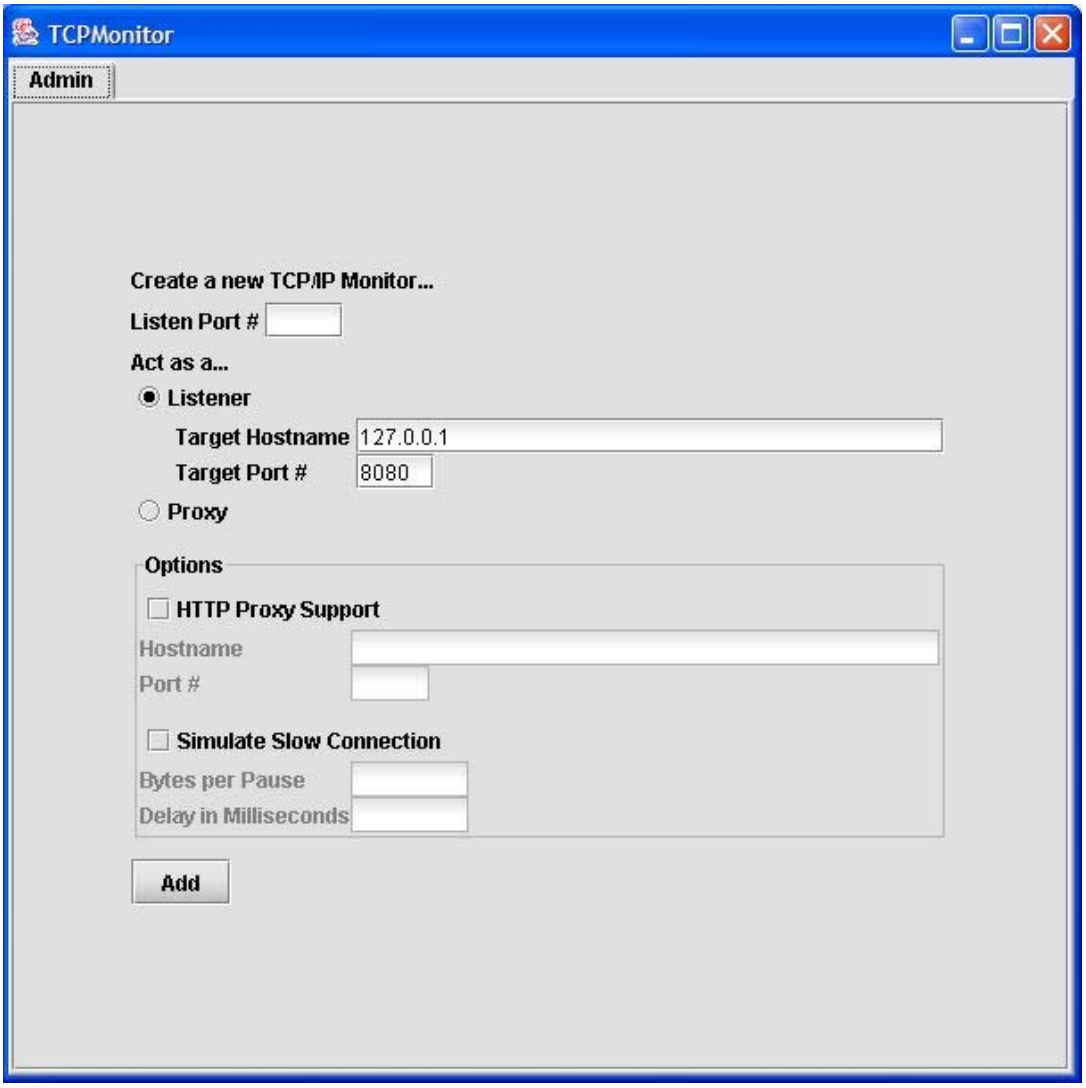
As you can see, starting with the MessageContext lets you work your way down through the API, discovering all the information available to you about a single request/response exchange. A MessageContext has two Messages, which each have a SOAPPart that contains a SOAPEnvelope. The SOAPEnvelope, in turn, holds a full representation of the SOAP Envelope that is sent over the wire. From here you can get and set the contents of the SOAP Header and the SOAP Body. See the Javadocs for a full list of the properties available.

Appendix : Using the Axis TCP Monitor (tcpmon)

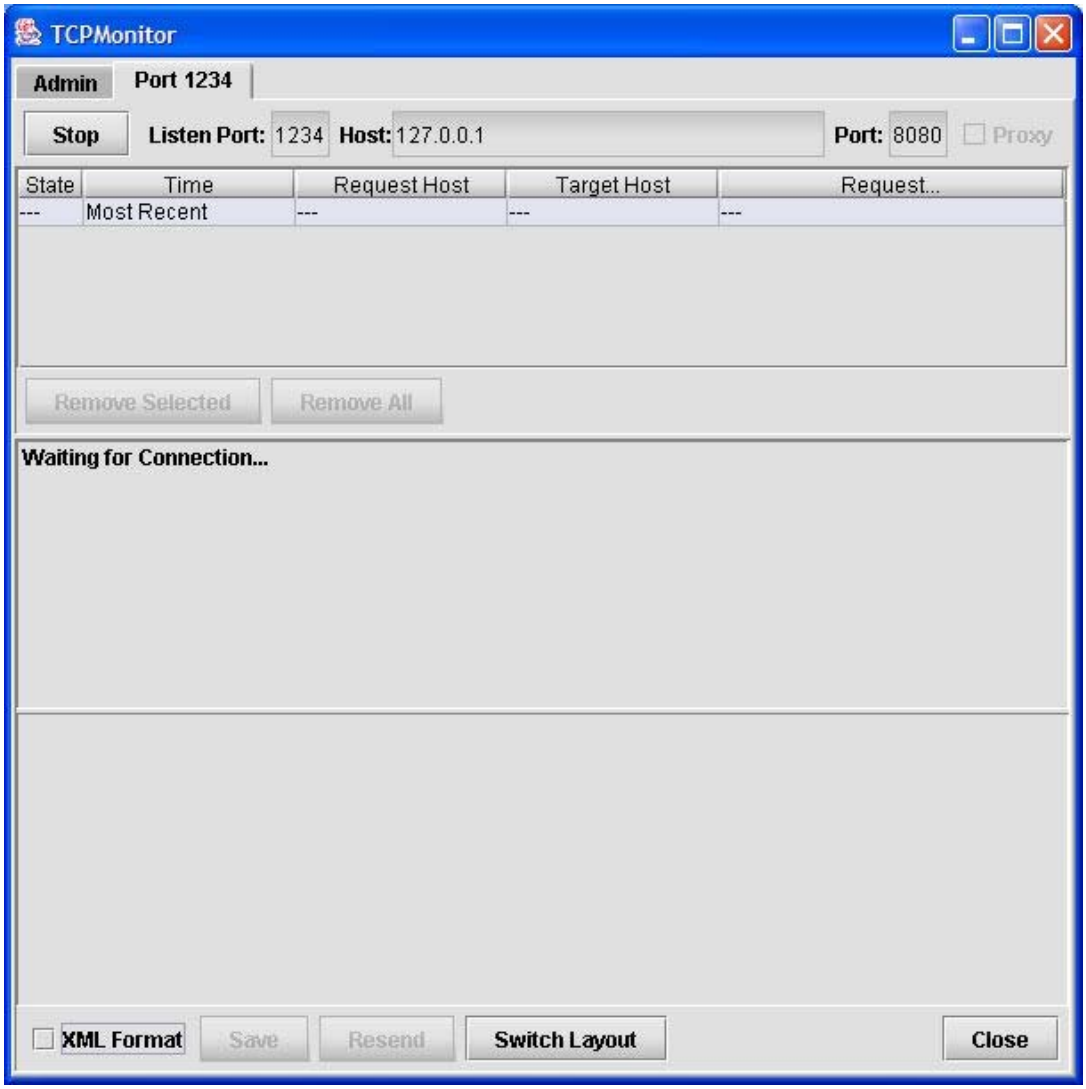
The included "tcpmon" utility can be found in the org.apache.axis.utils package. To run it from the command line:

```
% java org.apache.axis.utils.tcpmon [listenPort targetHost targetPort]
```

Without any of the optional arguments, you will get a gui which looks like this:



To use the program, you should select a local port which tcpmon will monitor for incoming connections, a target host where it will forward such connections, and the port number on the target machine which should be "tunneled" to. Then click "add". You should then notice another tab appearing in the window for your new tunneled connection. Looking at that panel, you'll see something like this:



Now each time a SOAP connection is made to the local port, you will see the request appear in the "Request" panel, and the response from the server in the "Response" panel. Tcpmon keeps a log of all request/response pairs, and allows you to view any particular pair by selecting an entry in the top panel. You may also remove selected entries, or all of them, or choose to save to a file for later viewing.

The "resend" button will resend the request you are currently viewing, and record a new response. This is particularly handy in that you can edit the XML in the request window before resending - so you can use this as a great tool for testing the effects of different XML on SOAP servers. Note that you may need to change the content-length HTTP header value before resending an edited request.

Appendix: Using the SOAP Monitor

Web service developers often have the need to see the SOAP messages being used to invoke web services along with the results of those messages. The goal of the SOAP Monitor utility is to provide a way for these developers to monitor the SOAP messages being used without requiring any special configuration or restarting of the server.

In this utility, a handler has been written and added to the global handler chain. As SOAP requests and responses are received, the SOAP message information is forwarded to a SOAP monitor service where it can be displayed using a web browser interface.

The SOAP message information is accessed with a web browser by going to http://localhost:<port>/axis/SOAPMonitor (where <port> is the port number where the application server is running).

The SOAP message information is displayed through a web browser by using an applet that opens a socket connection to the SOAP monitor service. This applet requires a Java plug-in 1.3 or higher to be installed in your browser. If you do not have a correct plug-in, the browser should prompt you to install one.

The port used by the SOAP monitor service to comminicate with applets is configurable. Edit the web.xml file for the Axis web application to change the port to be used.

Note: The SOAP Monitor is NOT enabled by default for security reasons. To enable it, read Enabling the SOAP Monitor in the Installation instructions.

Glossary

Handler

A reusable class which is responsible for processing a MessageContext in some custom way. The Axis engine invokes a series of Handlers whenever a request comes in from a client or a transport listener.

SOAP

The Simple Object Access Protocol (yes, despite the fact that it sometimes doesn't seem so simple, and doesn't have anything to do with objects... :)). You can read the SOAP 1.1 specification at http://www.w3.org/TR/soap. The W3C is currently in the midst of work on SOAP 1.2, under the auspices of the XML Protocol Group.

Provider

A provider is the "back-end" Handler which is responsible for actually performing the "meat" of the desired SOAP operation. Typically this means calling a method on some back-end service object. The two commonly used providers are RPCProvider and MsgProvider, both in the org.apache.axis.providers.java package.



PDF