# The Apache Axis Project

## 1. Axis

## 1.1. WebServices - Axis

### 1.1.1. WebServices - Axis - Introduction

NEWS (April 09, 2005) : Axis C++1.5 Final is now available!

NEWS ((March 01, 2005)) : Axis Axis 1.2 RC3 release is now available!

NEWS (February 08, 2005) : Axis C++1.5 Alpha is now available!

NEWS (December 16, 2004) : Axis C++1.4 Final is now available!

NEWS (December 03, 2004) : Axis C++1.4 Alpha is now available!

NEWS (November 16, 2004) : Axis 1.2 RC2 release is now available!

NEWS (October 29, 2004) : Axis C++1.3 Final is now available!

NEWS (September 30, 2004) : Axis 1.2 RC1 release is now available!

NEWS (September 15, 2004) : Axis C++1.3 Beta is now available!

NEWS (August 18, 2004) : Axis C++1.3 Alpha is now available!

NEWS (August 17, 2004) : Axis 1.2 beta 3 release is now available!

NEWS (July 14, 2004) : Axis 1.2 beta 2 release is now available!

NEWS (July 09, 2004) : Axis C++1.2 is now available!

NEWS (June 29, 2004) : Axis C++1.2 Beta is now available!

NEWS (June 15, 2004) : Axis C++1.2 Alpha is now available!

NEWS (May 07, 2004) : Axis C++1.1.1 is now available!

NEWS (April 16, 2004) : Axis C++1.1 is now available!

NEWS (March 31, 2004) : Axis 1.2 Beta is now available.

NEWS (December 1, 2003) : Axis 1.2 Alpha is now available.

NEWS (June 16, 2003) : Axis 1.1 Final is still the most recent stable release (read the release notes)!

Apache Axis is an implementation of the SOAP ("Simple Object Access Protocol") submission to W3C.

From the draft W3C specification: SOAP is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses.

This project is a follow-on to the Apache SOAP project.

Please see the Reference Library for a list of technical resources that should prove useful.

### 1.1.2. Axis 1.2 and beyond

Axis 1.1 has proven itself to be a reliable and stable base on which to implement Java Web Services. There is a very active user community and there a many companies who use Axis for Web Service support in their products.

For Axis 1.2, we are focusing on our document/literal support to better address the WS-I Basic Profile 1.0 and JAX-RPC 1.1 specifications. And we are fixing as many bug as possible.

We can always use your help. Here are some links to help you help us:
- How do I report bugs?
- How do I submit patches to Axis?
- Where can i get snapshots of latest CVS?

### 1.1.3. Credits

The Axis Development Team

## 1.2. WebServices - Axis

### 1.2.1. WebServices - Axis - News

(April 09, 2005) : Axis C++ 1.5 Final is available!

(March 01, 2005) : Axis 1.2 RC3 release is now available!

(February 08, 2005) : Axis C++ 1.5 Alpha is available!

(December 16, 2004) : Axis C++ 1.4 Final is available!

(December 03, 2004) : Axis C++ 1.4 Alpha is available!

(November 16, 2004) : Axis 1.2 RC2 release is now available!

(October 29, 2004) : Axis C++ 1.3 Final is available!

(September 30, 2004) : Axis 1.2 RC1 release is now available!

(September 15, 2004) : Axis C++ 1.3 Beta is available!

(August 18, 2004) : Axis C++ 1.3 Alpha is available!

(August 17, 2004) : Axis 1.2 beta 3 release is now available!

(July 14, 2004) : Axis 1.2 beta 2 release is now available!

(July 09, 2004) : Axis C++ 1.2 is available!

(June 29, 2004) : Axis C++ 1.2 Beta is available!

(June 15, 2004) : Axis C++ 1.2 Alpha is available!

(May 07, 2004) : Axis C++ 1.1.1 is available!

(April 16, 2004) : Axis C++ 1.1 is available!

(December 29, 2003) : Axis C++ 1.0 is released!

(December 01, 2003) : Axis 1.2 Alpha is available!

(December 01, 2003) : Axis C++ Beta is available!

(October 31, 2003) : Axis C++ alpha has been released!

(September 10, 2003) : Axis CVS Repository has moved from xml-axis to ws-axis

(June 16, 2003) : Axis 1.1 is now available!

(March 5, 2003) : Axis 1.1 RC2 release is now available!

(February 9, 2003) : Axis 1.1 RC1 release is now available!

(October 7, 2002) : Axis 1.0 is now available!

(September 30, 2002) : Axis 1.0 RC2 release is now available!

(September 6, 2002) : Axis 1.0 RC1 release is now available!

(July 9, 2002) : The Axis beta 3 release is available!

See the Mailing Lists for more information.

The Axis Development Team

## 1.3. Get Involved

### 1.3.1. WebServices - Axis

#### 1.3.1.1. WebServices - Axis - Overview

Every volunteer project obtains its strength from the people involved in it. We invite you to participate as much or as little as you choose. The roles and responsibilities that people can assume in the project are based on merit. Everybody's input matters!

There are a variety of ways to participate. Regardless of how you choose to participate, we suggest you join some or all of our mailing lists.

Use the Products and Give Us Feedback

Using the products,reporting bugs, making feature requests, etc. is by far the most important role. It's your feedback that allows the technology to evolve.

- Join Mailing Lists
- Download Binary Builds
- Report bugs/Request additional features

Contribute Code or Documentation Patches

In this role, you participate in the actual development of the code. If this is the type of role you'd like to play, here are some steps (in addition to the ones above) to get you started:

- Read Guidelines
- Review Reference Library
- Download the Source Code
- Access CVS Repository

### 1.3.2. WebServices - Axis

#### 1.3.2.1. WebServices - Axis - CVS Repositories

Most users of the source code probably don't need to have day to day access to the source code as it changes. For these users we provide easy to unpack source code downloads via our download pages.

View the Source Tree

Latest CVS sources can be viewed at http://cvs.apache.org/viewcvs/ws-axis/

Access the Source Tree (AnonCVS)
So, you've decided that you need access to the source tree to see the latest and greatest code. There's two different forms of CVS access. The first is anonymous and anybody can use it. The second is not and you must have a login to the development server. If you don't know what this means, join the mailing list and find out.

Anyone can checkout source code from our anonymous CVS server. To do so, simply use the following commands (if you are using a GUI CVS client, configure it appropriately):

```
cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic login password: anoncvs cvs -d
:pserver:anoncvs@cvs.apache.org:/home/cvspublic checkout ws-axis
```

Full Remote CVS Access
If you are a Committer and have a login on the Apache development server, this section is for you. If you are not a Committer, but you want to submit patches or even request commit privileges, please see the Jakarta GuideLines page (we follow the same rules) for more information.

To have full access to the CVS server, you need to follow the links depending on the operating system you are using:

- Unix
- Windows

### 1.3.3. WebServices - Axis

#### 1.3.3.1. WebServices - Axis - Mailing List

Before subscribing to any of the mailing lists, please make sure you have read and understand the guidelines.

While the mailing lists are not archived on Apache they are available at other sites, for example http://marc.theaimsgroup.com is pretty good and is used for searching below.

#### 1.3.3.2. The Axis User List

Medium Traffic Subscribe Unsubscribe Subscribe(Digest) Unsubscribe(Digest) Send mail to list
This list is for developers that are using Axis in their own projects to ask questions, share knowledge, and discuss issues related to using Axis.

Search:
[] Subjects [] Authors [] Bodies for list 'axis-user'

#### 1.3.3.3. The Axis Developer List

Medium Traffic Subscribe Unsubscribe Subscribe(Digest) Unsubscribe(Digest) Send mail to list
This is the list where participating developers of the Axis project meet and discuss issues, code changes/additions, etc.

Search:
[] Subjects [] Authors [] Bodies for list 'axis-dev'

#### 1.3.3.4. The Axis C++ User List

Medium Traffic Subscribe Unsubscribe Subscribe(Digest) Unsubscribe(Digest) Send mail to list
This list is for developers that are using Axis C++ in their own projects to ask questions, share knowledge, and discuss issues related to using Axis C++.

Search:
[] Subjects [] Authors [] Bodies for list 'axis-c-user'

#### 1.3.3.5. The Axis C++ Developer List

Medium Traffic Subscribe Unsubscribe Subscribe(Digest) Unsubscribe(Digest) Send mail to list
This is the list where participating developers of the Axis C++ project meet and discuss issues, code changes/additions, etc.

Search:
[] Subjects [] Authors [] Bodies for list 'axis-c-dev'

### 1.3.4. WebServices - Axis

### 1.3.4.1. WebServices - Axis - Reference Library

The Axis Project lives or fails based on its human resources. Users and contributors alike help the project with ideas and brainpower. A common foundation of knowledge is required to effectively participate in this virtual community. The following is a list of documents that we have found helpful for us and may be helpful to you:

These resources are required reading for anybody contributing source code to the project.

SOAP Specific Resources

SOAP W3C Specification
Required reading.

SOAP Messaging with Attachments W3C Specification
SOAP combined with MIME.

SOAP Security Extensions: Digital Signature Specification
Adding security to SOAP.

Other Specifications

Web Services Description Language (WSDL) 1.1

WS-I Basic Profile Version 1.0

Java API for XML-based RPC (JAX-RPC)

Other Resources

The Java Language Specification
Written by the creators of the Java Programming Language, this online book is considered by many to be the bible for programming in Java. A must read.

Javadoc
Javadoc is the automatic software documentation generator used by Java since it was first released. All code written for this project must be documented using Javadoc conventions.

The Java Code Conventions
This Sun document specifies the de-facto standard way of formatting Java code. All code written for this project must follow these conventions.

Open Source Development with CVS
Written by Karl Fogel, this is an online version of many of the primary chapters from the dead-tree version of his book.

Introduction to CVS
Written by Jim Blandy, this brief introduction gives a first look into CVS. If you have never used CVS before, you'll want to start here.

Version Management with CVS
Written by Per Cederqvist et al, this is the main manual for CVS. It provides details on all documented CVS features.

### 1.3.5. WebServices - Axis

#### 1.3.5.1. WebServices - Axis -

New Axis bugs should be reported using JIRA (the Apache bug database).

- Please report bugs against the newest release.
- If you're not sure whether the behavior in question is a bug or a feature, please post a message to the axis-dev mailing list for clarification.
- To avoid duplicate bug reports, please query JIRA to see whether the bug has already been reported (and perhaps fixed).
- If you can't find your bug in the database, it would help if you could check out Axis from CVS, and build it locally to verify that the bug still exists.
- If you have found a new bug, please enter an Axis bug report in JIRA. Remember to include the following information:
  - Version number of Axis
  - Version number of JDK (enter "java -fullversion")
  - Instructions for how to reproduce the problem, ideally including a small testcase.

  Before you can enter your first bug report, you must submit your email address to JIRA and receive a password.

Bugs related to WSDL4J should be addressed to the Expert Group for JSR110 at http://groups.yahoo.com/group/jsr110-eg-disc.

For more information visit the following links:

- Apache JIRA

We also encourage you to write patches for problems you find and submit them to the axis-dev mailing list. If we agree the problem is a bug and the patch fixes it and does not break something else, we are likely to include the patch in the next release.

### 1.3.6. How To Build Axis Project's Website

#### 1.3.6.1. Installing Forrest

The Axis website build system requires two components to perform a build.
Ant and Forrest.

Specifically the build has been tested to work with Ant version 1.6.1 and Forrest 0.5.1. To install these products download the distributions and follow the instructions in their documentation. Make sure you don't forget to set the environment variables FORREST_HOME and ANT_HOME. The ANT_HOME/bin directory should be in the path.

#### 1.3.6.2. Checking out ws-axis and ws-site module

Check out 'ws-axis/site/src' and 'ws-site/target/axis' module via your favorite CVS tools.
Please follow the guildeline written here.

#### 1.3.6.3. Running the Build

Here's a list of targets for the ant task. But, what you need to do is just "ant".

| Target | Description |
|---|---|
| clean | Erase all build work products (ie, everything in the build directory |
| run-forrest | Run Forrest with Jetty server to review the target |
| run-browser | Invoke a web browser (ie, Internet Explorer) |
| backcopy | Reflect the updates on the build directory to the master source on 'ws-axis/site/src' |
| forrest | Create the updated static contents |
| replace | Copy the contents to ws-site/targets/axis directory |
| build-site (default) | Do clean up and all tasks to the build site |

#### 1.3.6.4. For Committers (Highly recommended)

The procedure to make changes to http://ws.apache.org/axis/ is:

- *cd* into the local 'ws-axis/site' CVS dir
- execute "ant"
- make changes to 'build/webapp/content/xdocs'
- reload and review the contents with the autostarted browser
- close the browser and the forrest window when you are ready to finish editing the site
- cvs commit (ie, 'ws-axis/site/src' and 'ws-site/target/axis')

#### 1.3.6.5. Manual update (If you want to realize the value of ant tasks above ...)

If you just want to update the site step-by-step, the followings are the instructions.

1. Installing Forrest [Note] At this time, the version 0.5.1 of Forrest is tested version. 2. Checking out 'ws-axis/site' module [ex] 'ws-axis/site/src/documentation/content/xdocs/java/user-guide.ihtml' 3. Make changes to the target 4. Confirming the change with "forrest run" 4-1) cd into the local "ws-axis/site" CVS dir 4-2) execute "forrest run" 4-3) have an access to http://localhost:8888/ to see the site 5. Generating a static content with "forrest" 5-1) execute "forrest" in the "ws-axis/site" dir 5-2) check the generated contents in "ws-axis/site/build/site/" 6. Make commitments 6-1) commit the original source (xml/ihtml/gif/jpg) to "ws-axis" 6-2) copy the generated contents into "ws-site/targets/axis" 6-3) commit the generated contents to "ws-site" 7. (Optional) If you are in a hurry to reflect the change to the site, cd to /www/ws.apache.org, and execute "cvs update -P" on minotaur. [Note] *** VERY IMPORTANT *** YOU HAVE TO CHECK YOUR UMASK IS "002" BEFORE DOING THE COMMAND, OR THE SITE WILL BECOME A NON-UPDATABLE SITE FROM THEN ON. The site will be updated automatically twice a day 12 midnight and 12 noon PST by a cron job of dims.

#### 1.3.6.6. F.A.Q.

*Q1.*
I encountered
The <xmlcatalog> data type doesn't support the nested "catalogpath" element.
error, during the build.

*A1.*
Please make sure that your Ant version is later than 1.6 alpha. You can check the Ant version, by running "ant -version".

*Q2.*
I see an error like this regarding mirrors.pdf
[java] X [0] mirrors.pdf BROKEN ....

*A2.*
This is a known issue, but does not affect the site itself.

**( more to be come )**

## 1.4. Axis (Java)

### 1.4.1. Axis Documentation

#### 1.4.1.1. Documentation

This is the documentation for Apache Axis 1.2 If the version of Axis you are using is older or newer than this version, then this is the wrong documentation to be using. Read the version that came with your copy of Axis.

**Documentation for Axis Users**

- Installation Instructions
- User's Guide
- Client-side Axis
- Securing an Axis-based Web Service
- Axis Ant Tasks
- Reference Material
- Further Reading

**Documentation for Axis Developers**

- API Documentation
- Building Axis - Guidelines for building Axis with/without optional components.
- Developer's Guide - Collection of guidelines for developing code in Axis.
- Integration Guide - Description of APIs and development direction to allow integration into an existing web application server.
- Architecture Guide - Axis design concepts and rationale.

### 1.4.2. Axis installation instructions

#### 1.4.2.1. Axis installation instructions

**Contents**

## Introduction

This document describes how to install Apache Axis. It assumes you already know how to write and run Java code and are not afraid of XML. You should also have an application server or servlet engine and be familiar with operating and deploying to it. If you need an application server, we recommend Jakarta Tomcat. [If you are installing Tomcat, get the latest 4.1.x version, and the full distribution, not the LE version for Java 1.4, as that omits the Xerces XML parser]. Other servlet engines are supported, provided they implement version 2.2 or greater of the servlet API. Note also that Axis client and server requires Java 1.3 or later.

For more details on using Axis, please see the user guide.

## Things you have to know

A lot of problems with Axis are encountered by people who are new to Java, server-side Java and SOAP. While you can learn about SOAP as you go along, writing Axis clients and servers is not the right time to be learning foundational Java concepts, such as what an array is, or basic application server concepts such as how servlets work, and the basics of the HTTP protocol.

Things you need to know before writing a Web Service:

1. Core Java datatypes, classes and programming concepts.
2. What threads are, race conditions, thread safety and sychronization.
3. What a classloader is, what hierarchical classloaders are, and the common causes of a "ClassNotFoundException".
4. How to diagnose trouble from exception traces, what a NullPointerException (NPE) and other common exceptions are, and how to fix them.
5. What a web application is; what a servlet is, where classes, libraries and data go in a web application.
6. How to start your application server and deploy a web application on it.
7. What a network is, the core concepts of the IP protocol suite and the sockets API. Specifically, what is TCP/IP.
8. What HTTP is. The core protocol and error codes, HTTP headers and perhaps the details of basic authentication.
9. What XML is. Not necessarily how to parse it or anything, just what constitutes well-formed and valid XML.

Axis and SOAP depends on all these details. If you don't know them, Axis (or anyone else's Web Service middleware) is a dangerous place to learn. Sooner or later you will be forced to discover these details, and there are easier places to learn than Axis.

If you are completely new to Java, we recommend you start off with things like the Java Tutorials on Sun's web site, and perhaps a classic book like Thinking in Java, until you have enough of a foundation to be able to work with Axis. It is also useful to have written a simple web application, as this will give you some knowledge of how HTTP works, and how Java application servers integrate with HTTP. You may find the course notes from Mastering the World Wide Web useful in this regard, even though Axis is only introduced in lecture 28.

Be aware that there is a lot more needed to be learned in order to use Axis and SOAP effectively than the listing above. The other big area is "how to write internet scale distributed applications". Nobody knows how to do that properly yet, so that you have to learn this by doing.

## Step 0: Concepts

Apache Axis is an Open Source SOAP server and client. SOAP is a mechanism for inter-application communication between systems written in arbitrary languages, across the Internet. SOAP usually exchanges messages over HTTP: the client POSTs a SOAP request, and receives either an HTTP success code and a SOAP response or an HTTP error code. Open Source means that you get the source, but that there is no formal support organisation to help you when things go wrong.

SOAP messages are XML messages. These messages exchange structured information between SOAP systems. Messages consist of one or more SOAP elements inside an envelope, Headers and the SOAP Body. SOAP has two syntaxes for describing the data in these elements, *Section 5*, which is a clear descendant of the XML RPC system, and *XML Schema*, which is the newer (and usually better) system. Axis handles the magic of converting Java objects to SOAP data when it sends it over the wire or receives results. SOAP Faults are sent by the server when something goes wrong; Axis converts these to Java exceptions.

SOAP is intended to link disparate systems. It is not a mechanism to tightly bind Java programs written by the same team together. It can bind Java programs together, but not as tightly as RMI or Corba. If you try sending many Java objects that RMI would happily serialize, you will be disappointed at how badly Axis fails. This is by design: if Axis copied RMI and serialized Java objects to byte streams, you would be stuck to a particular version of Java everywhere.

Axis implements the JAX-RPC API, one of the standard ways to program Java services. If you look at the specification and tutorials on Sun's web site, you will understand the API. If you code to the API, your programs will work with other implementations of the API, such as those by Sun and BEA. Axis also provides extension features that in many ways extends the JAX-RPC API. You can use these to write better programs, but these will only work with the Axis implementation. But since Axis is free and you get the source, that should not matter.

Axis is compiled in the JAR file *axis.jar*; it implements the JAX-RPC API declared in the JAR files *jaxrpc.jar* and *saaj.jar*. It needs various helper libraries, for logging, WSDL processing and introspection. All these files can be packaged into a web application, *axis.war*, that can be dropped into a servlet container. Axis ships with some sample SOAP services. You can add your own by adding new compiled classes to the Axis webapp and registering them.

Before you can do that, you have to install it and get it working.

### Step 1: Preparing the webapp

Here we assume that you have a web server up and running on the localhost at port 8080. If your server is on a different port, replace references to 8080 to your own port number.

In your Application Server installation, you should find a directory into which web applications ("webapps") are to be placed. Into this directory copy the webapps/axis directory from the xml-axis distribution. You can actually name this directory anything you want, just be aware that the name you choose will form the basis for the URL by which clients will access your service. The rest of this document assumes that the default webapp name, "axis" has been used; rename these references if appropriate.

### Step 2: Setting up the libraries

In the Axis directory, you will find a WEB-INF sub-directory. This directory contains some basic configuration information, but can also be used to contain the dependencies and web services you wish to deploy.

Axis needs to be able to find an XML parser. If your application server or Java runtime does not make one visible to web applications, you need to download and add it. Java 1.4 includes the Crimson parser, so you *can* omit this stage, though the Axis team prefer Xerces.

To add an XML parser, acquire the JAXP 1.1 XML compliant parser of your choice. We recommend Xerces jars from the xml-xerces distribution, though others mostly work. Unless your JRE or app server has its own specific requirements, you can add the parser's libraries to axis/WEB-INF/lib. The examples in this guide use Xerces. This guide adds xml-apis.jar and xercesImpl.jar to the AXISCLASSPATH so that Axis can find the parser (see below).

If you get ClassNotFound errors relating to Xerces or DOM then you do not have an XML parser installed, or your CLASSPATH (or AXISCLASSPATH) variables are not correctly configured.

### Tomcat 4.x and Java 1.4

Java 1.4 changed the rules as to how packages beginning in java.* and javax.* get loaded. Specifically, they only get loaded from *endorsed* directories. jaxrpc.jar and saaj.jar contain javax packages, so they may not get picked up. If happyaxis.jsp (see below) cannot find the relevant packages, copy them from axis/WEB-INF/lib to CATALINA_HOME/common/lib and restart Tomcat.

### WebLogic 8.1

WebLogic 8.1 ships with `webservices.jar` that conflicts with Axis' `saaj.jar` and prevents Axis 1.2 from working right out of the box. This conflict exists because WebLogic uses an older definition of `javax.xml.soap.*` package from Java Web Services Developer Pack Version 1.0, whereas Axis uses a newer revision from J2EE 1.4.

However, there are two alternative configuration changes that enable Axis based web services to run on Weblogic 8.1.

- In a webapp containing Axis, set <prefer-web-inf-classes> element in `WEB-INF/weblogic.xml` to true. An example of `weblogic.xml` is shown below:
  <weblogic-web-app> <container-descriptor>
  <prefer-web-inf-classes>true</prefer-web-inf-classes> </container-descriptor>
  </weblogic-web-app>

  If set to `true`, the `<prefer-web-inf-classes>` element will force WebLogic's classloader to load classes located in the WEB-INF directory of a web application in preference to application or system classes. This is a recommended approach since it only impacts a single web module.
- In a script used to start WebLogic server, modify `CLASSPATH` property by placing Axis's `saaj.jar` library in front of WeLlogic's `webservices.jar`.

  **NOTE:** This approach impacts all applications deployed on a particular WebLogic instance and may prevent them from using WebLogic's webservices.

For more information on how WebLogic's class loader works, see WebLogic Server Application Classloading.

### Step 3: starting the web server

This varies on a product-by-product basis. In many cases it is as simple as double clicking on a startup icon or running a command from the command line.

### Step 4: Validate the Installation

After installing the web application and dependencies, you should make sure that the server is running the web application.

#### Look for the start page

Navigate to the start page of the webapp, usually http://127.0.0.1:8080/axis/, though of course the port may differ.
You should now see an Apache-Axis start page. If you do not, then the webapp is not actually installed, or the appserver is not running.

#### Validate Axis with happyaxis

Follow the link *Validate the local installation's configuration*
This will bring you to *happyaxis.jsp* a test page that verifies that needed and optional libraries are present. The URL for this will be something like http://localhost:8080/axis/happyaxis.jsp

If any of the needed libraries are missing, Axis will not work.
**You must not proceed until all needed libraries can be found, and this validation page is happy.**
Optional components are optional; install them as your need arises. If you see nothing but an internal server error and an exception trace, then you probably have multiple XML parsers on the CLASSPATH (or AXISCLASSPATH), and this is causing version confusion. Eliminate the extra parsers, restart the app server and try again.

#### Look for some services

From the start page, select *View the list of deployed Web services*. This will list all registered Web Services, unless the servlet is configured not to do so. On this page, you should be able to click on *(wsdl)* for each deployed Web service to make sure that your web service is up and running.

Note that the 'instant' JWS Web Services that Axis supports are not listed in this listing here. The install guide covers this topic in detail.

### Test a SOAP Endpoint

Now it's time to test a service. Although SOAP 1.1 uses HTTP POST to submit an XML request to the *endpoint*, Axis also supports a crude HTTP GET access mechanism, which is useful for testing. First let's retrieve the version of Axis from the version endpoint, calling the `getVersion` method:
http://localhost:8080/axis/services/Version?method=getVersion
This should return something like:

```
<?xml version="1.0" encoding="UTF-8" ?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
<getVersionResponse
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<getVersionReturn
xsi:type="xsd:string">
Apache Axis version: 1.1 Built on Apr 04, 2003 (01:30:37 PST)
</getVersionReturn>
</getVersionResponse>
</soapenv:Body>
</soapenv:Envelope>
```

The Axis version and build date may of course be different.

### Test a JWS Endpoint

Now let's test a JWS web service. Axis' JWS Web Services are java files you save into the Axis webapp *anywhere but the WEB-INF tree*, giving them the .jws extension. When someone requests the .jws file by giving its URL, it is compiled and executed. The user guide covers JWS pages in detail.

To test the JWS service, we make a request against a built-in example, EchoHeaders.jws (look for this in the axis/ directory).

Point your browser at http://localhost:8080/axis/EchoHeaders.jws?method=list.

This should return an XML listing of your application headers, such as

```
<?xml version="1.0" encoding="UTF-8" ?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

```
<soapenv:Body>
<listResponse
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<listReturn xsi:type="soapenc:Array"
soapenc:arrayType="xsd:string[6]"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
<item>accept:image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*</item>
<item>accept-language:en-us</item>
<item>accept-encoding:gzip, deflate</item>
<item>user-agent:Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)</item>
<item>host:localhost:8080</item>
<item>connection:Keep-Alive</item>
</listReturn>
</listResponse>
</soapenv:Body>
</soapenv:Envelope>
```

Again, the exact return values will be different, and you may need to change URLs to correct any host, port and webapp specifics.

**Step 5: Installing new Web Services**

So far you have got Axis installed and working--now it is time to add your own Web Service.

The process here boils down to (1) get the classes and libraries of your new service into the Axis WAR directory tree, and (2) tell the AxisEngine about the new file. The latter is done by submitting an XML deployment descriptor to the service via the Admin web service, which is usually done with the AdminClient program or the <axis-admin> Ant task. Both of these do the same thing: they run the Axis SOAP client to talk to the Axis administration service, which is a SOAP service in its own right. It's also a special SOAP service in one regard--it is restricted to local callers only (not remote access) and is password protected to stop random people from administrating your service. There is a default password that the client knows; if you change it then you need to pass the new password to the client.

The first step is to add your code to the server.

In the WEB-INF directory, look for (or create) a "classes" directory (i.e. axis/WEB-INF/classes ). In this directory, copy the compiled Java classes you wish to install, being careful to preserve the directory structure of the Java packages.

If your classes services are already packaged into JAR files, feel free to drop them into the WEB-INF/lib directory instead. Also add any third party libraries you depend on into the same directory.

After adding new classes or libraries to the Axis webapp, you must restart the webapp. This can be done by restarting your application server, or by using a server-specific mechanism to restart a specific webapp.

Note: If your web service uses the simple authorization handlers provided with xml-axis (this is actually not recommended as these are merely illustrations of how to write a handler than intended for production use), then you will need to copy the corresponding users.lst file into the WEB-INF directory.

## Step 6: Deploying your Web Service

The various classes and JARs you have just set up implement your new Web Service. What remains to be done is to tell Axis how to expose this web service. Axis takes a Web Service Deployment Descriptor (WSDD) file that describes in XML what the service is, what methods it exports and other aspects of the SOAP endpoint.

The users guide and reference guide cover these WSDD files; here we are going to use one from the Axis samples: the stock quote service.

### Classpath setup

In order for these examples to work, java must be able to find axis.jar, commons-discovery.jar, commons-logging.jar, jaxrpc.jar, saaj.jar, log4j-1.2.8.jar (or whatever is appropriate for your chosen logging implementation), and the XML parser jar file or files (e.g., xerces.jar). These examples do this by adding these files to AXISCLASSPATH and then specifying the AXISCLASSPATH when you run them. Also for these examples, we have copied the xml-apis.jar and xercesImpl.jar files into the AXIS_LIB directory. An alternative would be to add your XML parser's jar file directly to the AXISCLASSPATH variable or to add all these files to your CLASSPATH variable.
On Windows, this can be done via the following. For this document we assume that you have installed Axis in C:\axis. To store this information permanently in WinNT/2000/XP you will need to right click on "My Computer" and select "Properties". Click the "Advanced" tab and create the new environmental variables. It is often better to use WordPad to create the variable string and then paste it into the appropriate text field.

```
set AXIS_HOME=c:\axis
set AXIS_LIB=%AXIS_HOME%\lib
set AXISCLASSPATH=%AXIS_LIB%\axis.jar;%AXIS_LIB%\commons-discovery.jar;
%AXIS_LIB%\commons-logging.jar;%AXIS_LIB%\jaxrpc.jar;%AXIS_LIB%\saaj.jar;
%AXIS_LIB%\log4j-1.2.8.jar;%AXIS_LIB%\xml-apis.jar;%AXIS_LIB%\xercesImpl.jar
```

Unix users have to do something similar. Below we have installed AXIS into /usr/axis and are using the bash shell. See your shell's documentation for differences. To make variables permeate you will need to add them to your shell's startup (dot) files. Again, see your shell's documentation.

```
set AXIS_HOME=/usr/axis
set AXIS_LIB=$AXIS_HOME/lib
set AXISCLASSPATH=$AXIS_LIB/axis.jar:$AXIS_LIB/commons-discovery.jar:
$AXIS_LIB/commons-logging.jar:$AXIS_LIB/jaxrpc.jar:$AXIS_LIB/saaj.jar:
$AXIS_LIB/log4j-1.2.8.jar:$AXIS_LIB/xml-apis.jar:$AXIS_LIB/xercesImpl.jar
export AXIS_HOME; export AXIS_LIB; export AXISCLASSPATH
```

To use Axis client code, you can select AXISCLASSPATH when invoking Java by entering

```
java -cp %AXISCLASSPATH% ...
```

or

```
java -cp "$AXISCLASSPATH" ...
```

depending on the platform. You may omit the quotes if your CLASSPATH doesn't have spaces in it.
Also, it is probably a good time to add the AXISCLASSPATH variable to your CLASSPATH variable. This will enable you to not include the AXISCLASSPATH variable when launching the examples in this guide. This document assumes that you have NOT done this.

### Find the deployment descriptor

Look in axis/samples/stock for the file deploy.wsdd. This is the deployment descriptor we want to tell Axis about. Deployment descriptors are an Axis-specific XML file that tells Axis how to deploy (or undeploy) a Web Service, and how to configure Axis itself. The Axis Administration Web Service lets the AdminClient program and its Ant task counterpart submit a new WSDD file for interpretation. The Axis 'engine' will update its configuration, then save its state.

By default Axis saves it state into the global configuration file axis/WEB-INF/server-config.wsdd. Sometimes you see a warning message about such a file not being found--don't worry about this, because Axis auto-creates the file after you deploy something to it. You can check in the webapp to see what this file looks like--and even copy it to other systems if you want to give them identical configurations. Note that Axis needs an expanded web application *and* write access to the WEB-INF dir to save its state in this location.

### Run the admin client

Execute the following command from the samples/stock directory. If you are not in this directory you will get a "java.io.FileNotFoundException: deploy.wsdd (The system cannot find the file specified)" exception.

```
On Windows
java -cp %AXISCLASSPATH% org.apache.axis.client.AdminClient
-lhttp://localhost:8080/axis/services/AdminService deploy.wsdd
On UNIX
java -cp $AXISCLASSPATH org.apache.axis.client.AdminClient
-lhttp://localhost:8080/axis/services/AdminService deploy.wsdd
```

If you get some java client error (like ClassNotFoundException), then you haven't set up your AXISCLASSPATH (or CLASSPATH) variable right, mistyped the classname, or did some other standard error. Tracking down such problems are foundational Java development skills--if you don't know how to do these things, learn them now!

Note: You may need to replace localhost with your host name, and 8080 with the port number used by your web server. If you have renamed the web application to something other than "axis" change the URL appropriately.

If you get some AxisFault listing, then the client is working, but the deployment was unsuccessful. This is where the knowledge of the sockets API to TCP and the basics of the HTTP that Web Service development requires begins to be needed. If you got some socket error like connection refused, the computer at the far end isn't talking to you, so find the cause of that and fix it. If you get an HTTP error code back find out what the error means and correct the problem. These skills are fundamental to using web services.

The user's guide covers the AdminClient in more detail, and there is also an Ant task to automate the use of Axis in your Ant build scripts.

## Step 7: Testing

This step is optional, but highly recommended. For illustrative purposes, it is presumed that you have installed and deployed the stock quote demo.

- Change directory to the distribution directory for xml-axis and execute the following command (or its Unix equivalent):

```
On Windows
java -cp .;%AXISCLASSPATH% samples.stock.GetQuote
-lhttp://localhost:8080/axis/servlet/AxisServlet -uuser1 -wpass1 XXX On UNIX
java -cp $AXISCLASSPATH samples.stock.GetQuote
-lhttp://localhost:8080/axis/servlet/AxisServlet -uuser1 -wpass1 XXX
```

- You should get back "55.25" as a result.

Note: Again, you may need to replace localhost with your host name, and 8080 with the port number used by your web server. If you have renamed the web application to something other than "axis" change the URL appropriately.

### Advanced Installation: adding Axis to your own Webapp

If you are experienced in web application development, and especially if you wish to add web services to an existing or complex webapp, you can take an alternate approach to running Axis. Instead of adding your classes to the Axis webapp, you can add Axis to your application.

The core concepts are

1. Add axis.jar, wsdl.jar, saaj.jar, jaxrpc.jar and the other dependent libraries to your WAR file.
2. Copy all the Axis Servlet declarations and mappings from axis/WEB-INF/web.xml and add them to your own web.xml
3. Build and deploy your webapp.
4. Run the Axis AdminClient against your own webapp, instead of Axis, by changing the URL you invoke it with.

The process is also covered in Chapter 15 of Java Development with Ant, which can be downloaded as a PDF file.

### What if it doesn't work?

Axis is a complicated system to install. This is because it depends on the underlying functionality of your app server, has a fairly complex configuration, and, like all distributed applications, depends upon the network too.

We see a lot of people posting their problems on the axis-user mailing list, and other Axis users as well as the Axis developers do their best to help when they can. But before you rush to post your own problems to the mailing list, a word of caution:

Axis is free. This means nobody gets paid to man the support lines. All the help you get from the community is voluntary and comes from the kindness of their hearts. They may be other users, willing to help you get past the same hurdles they had to be helped over, or they may be the developers themselves. But it is all voluntary, so you may need to keep your expectations low!

1. Post to the user mail list, not the developer list. You may think the developer mail list is a

short cut to higher quality answers. But the developers are also on the user list along with many other skilled users--so more people will be able to answer your questions. Also, it is helpful for all user issues to be on one list to help build the searchable mailing list archive.

2. Don't ask non-Axis-related questions. The list is not the place to ask about non-Axis, non-SOAP, problems. Even questions about the MS Soap toolkit or .NET client side, don't get many positive answers--we avoid them. That also goes for the Sun Java Web Services Developer Pack, or the Jboss.net stuff that they've done with Axis.
3. Never bother posting to the soapbuilders mailing list either, that is only for people developing SOAP toolkits, not using them--all off-topic messages are pointedly ignored.
4. There is no guarantee that anyone will be able to solve your problem. The usual response in such a situation is silence, for a good reason: if everybody who didn't know the answer to a question said "I don't know", the list would be overflowed with noise. Don't take silence personally.
5. Never expect an immediate answer. Even if someone knows the answer, it can take a day or two before they read their mail. So if you don't get an answer in an hour or two, don't panic and resend. Be patient. And put the time to use by trying to solve your problems yourself.
6. Do your homework first. This document lists the foundational stuff you need to understand. It has also warned you that it can take a day to get a reply. Now imagine you get a HTTP Error '404' on a SOAP call. Should you rush to post a 'help' request, or should you try and find out what an HTTP error code is, what #404 usually means and how to use a Java debugger. We provide the source to make that debugging easier :)
7. Post meaningful subject lines. You want your message read, not deleted unread. A subject line of 'Axis problem', 'Help with Axis', etc. is not meaningful, and is not likely to get many readers.
8. Search the mailing list archives FIRST to see if someone had the same problem. This list is searchable--and may save you much time in getting an answer to your problem.
9. Use the jira database to search for Axis bugs, both open and closed.
10. Consult the Axis Wiki for its Frequently Asked Questions (FAQ), installation notes, interoperability issues lists, and other useful information.
11. Don't email people for help directly, unless you know them. It's rude and presumptuous. Messages sent over the mail list benefit the whole community--both the original posters and people who search the list. Personal messages just take up the recipients time, and are unwelcome. Usually, if not ignored outright, recipients of personal requests will just respond 'ask the mail list' anyway!
12. Know that configuration problems are hard to replicate, and so can be difficult to get help on. We have tried with the happyaxis.jsp demo to automate the diagnostics gathering for you, but it can be hard for people to be of help here, especially for obscure platforms.
13. Keep up to date with Axis releases, even the beta copies of forthcoming releases. You

wouldn't want your problem to be a bug that was already known and fixed in a more recent release. Often the common response to any question is 'have you tried the latest release'.

14. Study and use the source, and fix it when you find defects. Even fix the documentation when you find defects. It is only through the participation of Axis' users that it will ever get better.

Has this put you off joining and participating in the Axis user mail list? We hope not--this list belongs to the people who use Axis and so will be your peers as your project proceeds. We just need for you to be aware that it is not a 24x7 support line for people new to server side Java development, and that you will need to be somewhat self sufficient in this regard. It is not a silver bullet. However, knowing how to make effective use of the list will help you develop better with Axis.

## Summary

Axis is simply an implementation of SOAP which can be added to your own webapp, and a webapp which can host your own web services. Installing it can be a bit fiddly, especially given Java 1.4's stricter requirements. If you follow a methodical process, including testing along the way, using happyaxis and the bundled test services, you will find it easier to get started with Axis.

## Appendix: Enabling the SOAP Monitor

SOAP Monitor allows for the monitoring of SOAP requests and responses via a web browser with Java plug-in 1.3 or higher. For a more comprehensive explanation of its usage, read Using the SOAP Monitor in the User's Guide.

By default, the SOAP Monitor is not enabled. The basic steps for enabling it are compiling the SOAP Monitor java applet, deploying the SOAP Monitor web service and adding request and response flow definitions for each monitored web service. In more detail:

1. Go to $AXIS_HOME/webapps/axis (or %AXIS_HOME%\webapps\axis) and compile SOAPMonitorApplet.java.

**On Windows**
```
javac -classpath %AXIS_HOME%\lib\axis.jar SOAPMonitorApplet.java
```
**On Unix**
```
javac -classpath $AXIS_HOME/lib/axis.jar SOAPMonitorApplet.java
```

   Copy all resulting class files (i.e. SOAPMonitorApplet*.class) to the root directory of the web application using the SOAP Monitor (e.g. .../tomcat/webapps/axis)

2. Deploy the SOAPMonitorService web service with the admin client and the

deploy-monitor.wsdd file (shown below).
Go to the directory deploy-monitor.wsdd is located and execute the command below. The command assume that /axis is the intended web application and it is available on port 8080.

**On Windows**
```
java -cp %AXISCLASSPATH% org.apache.axis.client.AdminClient
-lhttp://localhost:8080/axis/services/AdminService deploy-monitor.wsdd
```
**On UNIX**
```
java -cp $AXISCLASSPATH org.apache.axis.client.AdminClient
-lhttp://localhost:8080/axis/services/AdminService deploy-monitor.wsdd
```

**SOAPMonitorService Deployment Descriptor (deploy-monitor.wsdd)** <deployment xmlns="http://xml.apache.org/axis/wsdd/" xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"> <handler name="soapmonitor" type="java:org.apache.axis.handlers.SOAPMonitorHandler"> <parameter name="wsdlURL" value="/axis/SOAPMonitorService-impl.wsdl"/> <parameter name="namespace" value="http://tempuri.org/wsdl/2001/12/SOAPMonitorService-impl.wsdl"/> <parameter name="serviceName" value="SOAPMonitorService"/> <parameter name="portName" value="Demo"/> </handler> <service name="SOAPMonitorService" provider="java:RPC"> <parameter name="allowedMethods" value="publishMessage"/> <parameter name="className" value="org.apache.axis.monitor.SOAPMonitorService"/> <parameter name="scope" value="Application"/> </service> </deployment>

3. For each service that is to be monitored, add request and response flow definitions to the service's deployment descriptor and deploy (or redeploy) the service. The **requestFlow**requestFlow and **responseFlow**responseFlow definitions follow the start tag of the **<service>**<service> element. If a service is already deployed, undeploy it and deploy it with the modified deployment descriptor. An example is shown below: ... <service name="xmltoday-delayed-quotes" provider="java:RPC"> <requestFlow> <handler type="soapmonitor"/> </requestFlow> <responseFlow> <handler type="soapmonitor"/> </responseFlow> ...
4. With a web browser, go to http[s]://host[:port][/webapp]/SOAPMonitor (e.g. http://localhost:8080/axis/SOAPMonitor) substituting the correct values for your web application. This will show the SOAP Monitor applet for viewing service requests and responses. Any requests to services that have been configured and deployed correctly should show up in the applet.
5. Copyright © 2001-2003, Apache Software Foundation

### 1.4.3. Axis User's Guide

### 1.4.3.1. Axis User's Guide

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

#### Table of Contents

#### 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>SK001</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"><service ... style="document"> for document style
<service ... style="wrapped"><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.

### 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.

### 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.

## What Axis can send via SOAP with restricted Interoperability

## What Axis can not send via SOAP

## 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!)

## 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)).getServ`

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.

## 1.4.4. Axis Developer's Guide

### 1.4.4.1. Axis Developer's Guide

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

#### Introduction

This guide is a collection of topics related to developing code for Axis.

#### General Guidelines

- Axis specific information (cvs repository access, mailing list info, etc.) can be found on the Axis Home Page.
- Axis uses the Jakarta Project Guidelines.
- Code changes should comply with "Code Conventions for the Java Programming Language"
- When fixing a bug, please include the href of the bug in the cvs commit message.
- Incompatible changes to published Axis interfaces should be avoided where possible. When changes are necessary, for example to maintain or improve the overall modularity of Axis, the impact on users must be considered and, preferably, documented.

- If you are making a big change that may affect interoperability, please run the echotest2 round 2 interop test to ensure that your change does not result in any new interop failures. You will also need the client_deploy.wsdd. Here are the nightly interop test results.

### Development Environment

The following packages are required for axis development:

- ant - Java based build tool. **Please Note: Version 1.5 OR HIGHER is required**
- junit - testing package
- xerces - xml processor
- Install Java 1.3.1 JDK (or later).

The Axis jar files are built in the xml-axis/java/build/lib directory. Here is an example CLASSPATH, which I use when developing code:

```
G:\xerces\xerces-1_4_2\xerces.jar
G:\junit3.7\junit.jar
G:\xml-axis\java\build\lib\commons-discovery.jar
G:\xml-axis\java\build\lib\commons-logging.jar
G:\xml-axis\java\build\lib\wsdl4j.jar
G:\xml-axis\java\build\lib\axis.jar
G:\xml-axis\java\build\lib\log4j-1.2.8.jar
G:\xml-axis\java\build\classes
```

If you access the internet via a proxy server, you'll need to set an environment variable so that the Axis tests do the same. Set ANT_OPTS to, for example:

```
      -Dhttp.proxyHost=proxy.somewhere.com
      -Dhttp.proxyPort=80
      -Dhttp.nonProxyHosts="localhost"
```

### Pluggable-Components

The Axis Architecture Guide explains the requirements for pluggable components.

#### Discovery

An Axis-specific component factory should be created of the form:

org.apache.axis.components.<componentType>.<factoryClassName>

For example, `org.apache.axis.components.logger.LogFactory` is the factory, or discovery mechanism, for the logger component/service.

The `org.apache.axis.components.image` package demonstrates both a factory, and supporting classes for different image tools used by Axis. This is representative of a pluggable component that uses external tooling, isolating it behind a 'thin' wrapper to Axis that provides only a limited interface to meet Axis minimal requirements. This allows future designers and implementors to gain an explicit understanding of the Axis's specific requirements on these tools.

## Logging/Tracing

Axis logging and tracing is based on the Logging component of the Jakarta Commons project, or the Jakarta Commons Logging (JCL) SPI. The JCL provides a Log interface with thin-wrapper implementations for other logging tools, including Log4J, Avalon LogKit, and JDK 1.4. The interface maps closely to Log4J and LogKit.

### Using the Logger SPI

To use the JCL SPI from a Java class, include the following import statements:

import org.apache.commons.logging.Log; import org.apache.axis.components.logger.LogFactory;

For each class definition, declare and initialize a `log` attribute as follows:

public class CLASS { private static Log log = LogFactory.getLog(CLASS.class); ...

Messages are logged to a *logger*, such as `log` by invoking a method corresponding to *priority*: The `Log` interface defines the following methods for use in writing log/trace messages to the log:

log.fatal(Object message); log.fatal(Object message, Throwable t); log.error(Object message); log.error(Object message, Throwable t); log.warn(Object message); log.warn(Object message, Throwable t); log.info(Object message); log.info(Object message, Throwable t); log.debug(Object message); log.debug(Object message, Throwable t); log.trace(Object message); log.trace(Object message, Throwable t);

While semantics for these methods are ultimately defined by the implementation of the Log interface, it is expected that the severity of messages is ordered as shown in the above list.

In addition to the logging methods, the following are provided:

log.isFatalEnabled(); log.isErrorEnabled(); log.isWarnEnabled(); log.isInfoEnabled(); log.isDebugEnabled(); log.isTraceEnabled();

These are typically used to guard code that only needs to execute in support of logging, and that introduces undesirable runtime overhead in the general case (logging disabled).

### Guidelines

#### Message Priorities

It is important to ensure that log message are appropriate in content and severity. The following guidelines are suggested:

- fatal - Severe errors that cause the Axis server to terminate prematurely. Expect these to be immediately visible on a console, and MUST be internationalized.
-
- error - Other runtime errors or unexpected conditions. Expect these to be immediately visible on a console, and MUST be internationalized.
-
- warn - Use of deprecated APIs, poor use of API, almost errors, other runtime situations that are undesirable or unexpected, but not necessarily "wrong". Expect these to be immediately visible on a console, and MUST be internationalized.
-
- info - Interesting runtime events (startup/shutdown). Expect these to be immediately visible on a console, so be conservative and keep to a minimum. These MUST be internationalized.
-
- debug - detailed information on flow of through the system. Expect these to be written to logs only. These NEED NOT be internationalized, but it never hurts...
-
- trace - more detailed information. Expect these to be written to logs only. These NEED NOT be internationalized, but it never hurts...

### Configuring the Logger

The Jakarta Commons Logging (JCL) SPI can be configured to use different logging toolkits. To configure which logger is used by the JCL, see the Axis System Integration Guide.

Configuration of the behavior of the JCL ultimately depends upon the logging toolkit being used. The JCL SPI (and hence Axis) uses Log4J by default if it is available (in the CLASSPATH).

### Log4J

As Log4J is the prefered/default logger for Axis, a *few* details are presented herein to get the developer going.

Configure Log4J using system properties and/or a properties file:

- **log4j.configuration=*log4j.properties***

  Use this system property to specify the name of a Log4J configuration file. If not specified, the default configuration file is *log4j.properties*. A *log4j.properties* file is provided in `axis.jar`.

This properties file can sometimes be overridden by placing a file of the same name so as to appear before `axis.jar` in the CLASSPATH. However, the precise behaviour depends on the classloader that is in use at the time, so we don't recommend this technique.

A safe way of overriding the properties file is to replace it in axis.jar. However, this isn't very convenient, especially if you want to tweak the properties during a debug session to filter out unwanted log entries. A more convenient alternative is to use an absolute file path to specify the properties file. This will even ignore web app's and their classloaders. So, for example on Linux, you could specify the system property:

log4j.configuration=file:/home/fred/log4j.props

- **log4j.debug** A good way of telling where log4j is getting its configuration from is to set this system property and look at the messages on standard output.

- **log4j.rootCategory=*priority* [, *appender*]*** Set the default (root) logger priority.

- **log4j.logger.*logger.name*=*priority*** Set the priority for the named logger and all loggers hierarchically lower than, or below, the named logger. *logger.name* corresponds to the parameter of `LogFactory.getLog(`*`logger.name`*`)`, used to create the logger instance. Priorities are: `DEBUG`, `INFO`, `WARN`, `ERROR`, or `FATAL`.

  Log4J understands hierarchical names, enabling control by package or high-level qualifiers: `log4j.logger.org.apache.axis.encoding=DEBUG` will enable debug messages for all classes in both `org.apache.axis.encoding` and `org.apache.axis.encoding.ser`. Likewise, setting `log4j.logger.org.apache.axis=DEBUG` will enable debug message for all Axis classes, but not for other Jakarta projects.

  A combination of settings will enable you to see the log events that you are interested in and omit the others. For example, the combination:

  log4j.logger.org.apache.axis=DEBUG log4j.logger.org.apache.axis.encoding=INFO log4j.logger.org.apache.axis.utils=INFO log4j.logger.org.apache.axis.message=INFO cuts down the number of a log entries produced by a single request to a manageable number.
- **log4j.appender.*appender*.Threshold=*priority*** Log4J *appenders* correspond to different output devices: console, files, sockets, and others. If appender's *threshold* is less than or equal to the message priority then the message is written by that appender. This allows different levels of detail to be appear at different log destinations.

  For example: one can capture DEBUG (and higher) level information in a logfile, while limiting console output to INFO (and higher).

## Axis Servlet Query String Plug-ins

Any servlet that is derived from the `org.apache.axis.transport.http.AxisServlet` class supports a number of standard query strings (*?list*, *?method*, and *?wsdl*) that provide information from or perform operations on a web service (for instance, *?method* is used to invoke a method on a web service and *?wsdl* is used to retrieve the WSDL document for a web service). Axis servlets are not limited to these three query strings and developers may create their own "plug-ins" by implementing the `org.apache.axis.transport.http.QSHandler` interface. There is one method in this interface that must be implemented, with the following signature:

public void invoke (MessageContext msgContext) throws AxisFault;

The `org.apache.axis.MessageContext` instance provides the developer with a number of useful objects (such as the Axis engine instance, and HTTP servlet objects) that are accessible by its `getProperty` method. The following constants can be used to retrieve various objects provided by the Axis servlet invoking the query string plug-in:

- **org.apache.axis.transport.http.HTTPConstants.PLUGIN_NAME**
  A `String` containing the name of the query string plug-in. For instance, if the query string *?wsdl* is provided, the name of the plugin is *wsdl*.
- **org.apache.axis.transport.http.HTTPConstants.PLUGIN_SERVICE_NAME**
  A `String` containing the name of the Axis servlet that inovked the query string plug-in.
- **org.apache.axis.transport.http.HTTPConstants.PLUGIN_IS_DEVELOPMENT**
  A `Boolean` containing `true` if this version of Axis is considered to be in development mode, `false` otherwise.
- **org.apache.axis.transport.http.HTTPConstants.PLUGIN_ENABLE_LIST**
  A `Boolean` containing `true` if listing of the Axis server configuration is allowed, `false` otherwise.
- **org.apache.axis.transport.http.HTTPConstants.PLUGIN_ENGINE**
  A `org.apache.axis.server.AxisServer` object containing the engine for the Axis server.
- **org.apache.axis.transport.http.HTTPConstants.MC_HTTP_SERVLETREQUEST**
  The `javax.servlet.http.HttpServletRequest` object from the Axis servlet that invoked the query string plug-in
- **org.apache.axis.transport.http.HTTPConstants.MC_HTTP_SERVLETRESPONSE**
  The `javax.servlet.http.HttpServletResponse` object from the Axis servlet that invoked the query string plug-in
- **org.apache.axis.transport.http.HTTPConstants.PLUGIN_WRITER**
  The `java.io.PrintWriter` object from the Axis servlet that invoked the query string plug-in
- **org.apache.axis.transport.http.HTTPConstants.PLUGIN_LOG**

The `org.apache.commons.logging.Log` object from the Axis servlet that invoked the query string plug-in, which is used to log messages.

- **org.apache.axis.transport.http.HTTPConstants.PLUGIN_EXCEPTION_LOG**
  The `org.apache.commons.logging.Log` object from the Axis servlet that invoked the query string plug-in, which is used to log exceptions.

Query string plug-in development is much like normal servlet development since the same basic information and methods of output are available to the developer. Below is an example query string plug-in which simply displays the value of the system clock (`import` statements have been omitted for brevity):

public class QSClockHandler implements QSHandler { public void invoke (MessageContext msgContext) throws AxisFault { PrintWriter out = (PrintWriter) msgContext.getProperty (HTTPConstants.PLUGIN_WRITER); HttpServletResponse response = (HttpServletResponse) msgContext.getProperty (HTTPConstants.MC_HTTP_SERVLETRESPONSE); response.setContentType ("text/html"); out.println ("<HTML><BODY><H1>" + System.currentTimeMillis() + "</H1></BODY></HTML>"); } }

Once a query string plug-in class has been created, the Axis server must be set up to recognize the query string which invokes it. See the section Deployment (WSDD) Reference in the Axis Reference Guide for information on how the HTTP transport section of the Axis server configuration file must be set up.

## Configuration Properties

Axis is in the process of moving away from using system properties as the primary point of internal configuration. Avoid calling `System.getProperty()`, and instead call `AxisProperties.getProperty`. `AxisProperties.getProperty` will call `System.getProperty`, and will (eventually) query other sources of configuration information.

Using this central point of access will allow the global configuration system to be redesigned to better support multiple Axis engines in a single JVM.

## Exception Handling

Guidelines for Axis exception handling are based on best-practices for exception handling. While there are details specific to Axis in these guidelines, they apply in principle to any project; they are included here for two reasons. First, because they are not listed elsewhere in the Apache/Jakarta guidelines (or haven't been found). Second, because adherence to these guidelines is considered crucial to enterprise ready middleware.

These guidelines are fundamentally independent of programming language. They are based on experience, but proper credit must be given to *More Effective C++*, by Scott Meyers, for opening the eyes of the innocent(?) many years ago.

Finally, these are guidelines. There will always be exceptions to these guidelines, in which case all that can be asked (as per these guidelines) is that they be logged in the form of comments in the code.

- Primary Rule: Only Catch An Exception If You Know What To Do With It
- If code catches an exception, it should know what to do with it at that point in the program. Any exception to this rule must be documented with a GOOD reason. Code reviewers are invited to put on their vulture beaks and peck away...

  There are a few corollaries to this rule.

  - Handle Specific Exceptions in Inner Code
  - Inner code is code *deep* within the program. Such code should catch specific exceptions, or categories of exceptions (parents in exception hierarchies), if and only if the exception can be resolved and normal flow restored to the code. Note that behaviour of this sort may be significantly different between non-interactive code versus an interactive tool.
  - Catch All Exceptions in Outermost Flow of Control
  - Ultimately, all exceptions must be dealt with at one level or another. For command-line tools, this means the `main` method or program. For a middleware component, this is the entry point(s) into the component. For Axis this is `AxisServlet` or equivalent.

    After catching specific exceptions which can be resolved internally, the outermost code must ensure that all internally generated exceptions are caught and handled. While there is generally not much that can be done, at a minimum the code should log the exception. In addition to logging, the Axis Server wraps all such exceptions in AxisFaults and returns them to the client code.

    This may seem contrary to the primary rule, but in fact we are claiming that Axis does know what to do with this type of exception: exit gracefully.

- Catching and Logging Exceptions
- When an Exception is going to cross a component boundry (client/server, or system/business logic), the exception must be caught and logged by the throwing component. It may then be rethrown, or wrapped, as described below.

  When in doubt, log the exception.

  - Catch and Throw
  - If an exception is caught and rethrown (unresolved), logging of the exception is at the discretion of the coder and reviewers. If any comments are logged, the exception

should also be logged.

When in doubt, log the exception and any related local information that can help to identify the complete context of the exception.

Log the exception as an *error* (`log.error()`) if it is known to be an unresolved or unresolvable error, otherwise log it at the *informative* level (`log.info()`).

    - Catch and Wrap
    - When exception `e` is caught and wrapped by a new exception `w`, log exception `e` before throwing `w`.

      Log the exception as an *error* (`log.error()`) if it is known to be an unresolved or unresolvable error, otherwise log it at the *informative* level (`log.info()`).

    - Catch and Resolve
    - When exception `e` is caught and resolved, logging of the exception is at the discretion of the coder and reviewers. If any comments are logged, the exception should also be logged (`log.info()`). Issues that must be balanced are performance and problem resolvability.

      Note that in many cases, ignoring the exception may be appropriate.

- Respect Component Boundries
- There are multiple aspects of this guideline. On one hand, this means that business logic should be isolated from system logic. On the other hand, this means that client's should have limited exposure/visibility to implementation details of a server - particularly when the server is published to outside parties. This implies a well designed server interface.
    - Isolate System Logic from Business Logic
    - Exceptions generated by the Axis runtime should be handled, where possible, within the Axis runtime. In the worst case the details of an exception are to be logged by the Axis runtime, and a generally descriptive Exception raised to the Business Logic.

      Exceptions raised in the business logic (this includes the server and Axis handlers) must be delivered to the client code.

    - Protect System Code from User Code
    - Protect the Axis runtime from uncontrolled user business logic. For Axis, this means that dynamically configurable `handlers`, `providers` and other user controllable hook-points must be guarded by `catch(Exception ...)`. Exceptions generated by user code and caught by system code should be:
        - Logged, and
        - Delivered to the client program
    - Isolate Visibility into Server from Client
    - Specific exceptions should be logged at the server side, and a more general exception thrown to the client. This prevents clues as to the nature of the server (such as

handlers, providers, etc) from being revealed to client code. The Axis component boundries that should be respected are:
  - Client Code <--> AxisClient
  - AxisClient <--> AxisServlet (AxisServer/AxisEngine)
  - AxisServer/AxisEngine <--> Web Service
- Throwing Exceptions in Constructors
- Before throwing an exception in a constructor, ensure that any resources owned by the object are cleaned up. For objects holding resources, this requires catching all exceptions thrown by methods called within the constructor, cleaning up, and rethrowing the exceptions.

### Compile and Run

The xml-axis/java/build.xml file is the primary 'make' file used by ant to build the application and run the tests. The build.xml file defines ant build *targets*. Read the build.xml file for more information. Here are some of the useful targets:

- compile -> compiles the source and creates xml-axis/java/build/lib/axis.jar
- javadocs -> creates the javadocs in xml-axis/java/build/javadocs
- functional-tests -> compiles and runs the functional tests
- all-tests -> compiles and runs all of the tests

To compile the source code:

```
cd xml-axis/java ant compile
```

To run the tests:

```
cd xml-axis/java ant functional-tests
```

**Note:** these tests start a server on port 8080. If this clashes with the port used by your web application server (such as Tomcat), you'll need to change one of the ports or stop your web application server when running the tests.

**Please run ant functional-tests and ant all-tests before checking in new code.**

### Internationalization

If you make changes to the source code that results in the generation of text (error messages or debug information), you must follow the following guidelines to ensure that your text is properly translated.

#### Developer Guidelines

1. Your text string should be added as a property to the resource.properties file (xml-axis/java/src/org/apache/axis/i18n/resource.properties). Note that some of the utility applications (i.e. tcpmon) have their own resource property files (tcpmon.properties).

3. The resource.properties file contains translation and usage instructions. Entries in a message resource file are of the form <key>=<message>. Here is an example message:
4. sample00=My name is {0}, and my title is {1}.
    1. sample00 is the key that the code will use to access this message.
    2. The text after the = is the message text.
    3. The {*number*} syntax defines the location for inserts.
5. The code should use the static method org.apache.axis.i18n.Messages.getMessage to obtain the text and add inserts. Here is an example usage:
6. Messages.getMessage("sample00", "Rich Scheuerle", "Software Developer");
7. All keys in the properties file should use the syntax <string><2-digit-suffix>.
8. 
    1. **Never change the message text in the properties file.** The message may be used in multiple places in the code. Plus translation is only done on new keys.
    2. If a code change requires a change to a message, create a new entry with an incremented 2-digit suffix.
    3. All new entries should be placed at the bottom of the file to ease translation.
    4. We may occasionally want to trim the properties file of old data, but this should only be done on major releases.

### Example

Consider the following statement:

```
if ( operationName == null )
throw new AxisFault( "No operation name specified" );
```

We will add an entry into org/apache/axis/i18n/resource.properties:

```
noOperation=No operation name specified.
```

And change the code to read:

```
if ( operationName == null )
throw new AxisFault(Messages.getMessage("noOperation"));
```

### Interface

Axis uses the standard Java internationalization class `java.util.ResourceBundle` to access property files and message strings, and uses `java.text.MessageFormat` to format the strings using variables. Axis provides a single class `org.apache.axis.i18n.Messages` that manages both ResourceBundle and MessageFormat classes. Messages methods are:

```
public static java.util.ResourceBundle getResourceBundle();

public static String getMessage(String key) throws java.util.MissingResourceException;

public static String getMessage(String key, String var) throws java.util.MissingResourceException;

public static String getMessage(String key, String var1, String var2) throws java.util.MissingResourceException;

public static String getMessage(String key, String[] vars) throws java.util.MissingResourceException;
```

Axis programmers can work with the resource bundle directly via a call to `Messages.getResourceBundle()`, but the `getMessage()` methods should be used instead for two reasons:

1. It's a shortcut. It is cleaner to call
2. `Messages.getMessage("myMsg00");` than `Messages.getResourceBundle().getString("myMsg00");`
3. The `getMessage` methods enable messages with variables.

#### The getMessage methods

If you have a message with no variables

```
myMsg00=This is a string.
```

then simply call

```
Messages.getMessage("myMsg00");
```

If you have a message with variables, use the syntax "{X}" where `X` is the number of the variable, starting at 0. For example:

```
myMsg00=My {0} is {1}.
```

then call:

```
Messages.getMessage("myMsg00","name", "Russell");
```

and the resulting string will be: "My name is Russell."

You could also call the String array version of getMessage:

```
Messages.getMessage("myMsg00", new String[] {"name", "Russell"});
```

The String array version of getMessage is all that is necessary, but the vast majority of messages will have 0, 1 or 2 variables, so the other getMessage methods are provided as a convenience to avoid the complexity of the String array version.

Note that the getMessage methods throw MissingResourceException if the resource cannot be found. And ParseException if there are more {X} entries than arguments. These exceptions are RuntimeException's, so the caller doesn't have to explicitly catch them.

The resource bundle properties file is org/apache/axis/i18n/resource.properties.

### Extending Message Files

Generally, within Axis all messages are placed in org.apache.axis.i18n.resource.properties. There are facilities for extending the messages without modifying this file for integration or 3rd party extensions to Axis. See the Integration Guide for details.

### Creating a WSDL Test

Here are the steps that I used to create the sequence test, which generates code from a wsdl file and runs a sequence validation test:

1. Created a xml-axis/java/test/wsdl/sequence directory.
3. Created a SequenceTest.wsdl file defining the webservice.
5. Ran the Wsdl2java emitter to create Java files:
6. java org.apache.axis.wsdl.Wsdl2java -t -s SequenceTest.wsdl

   1. The -t option causes the emitter to generate a *TestCase.java file that hooks into the test harness. This file is operational without any additional changes. Copy the *TestCase.java file into the same directory as your wsdl file. (Ideally only the Java files that are changed need to be in your directory.) So this file is not needed, but please make sure to modify your <wsdl2java ...> clause (described below) to emit a testcase.
   2. The -s option causes the emitter to generate a *SOAPBindingImpl.java file. The Java file contains empty methods for the service. You probably want to fill them in with your own logic. Copy the *SOAPBindingImpl.java file into the same directory as your wsdl file. (If no changes are needed in the Java file, you don't need to save it. But you will need to make sure that your <wsdl2java ...> clause generates a skeleton).
   3. Remove all of the Java files that don't require modification. So you should have three files in your directory (wsdl file, *TestCase.java, and *SOAPBindingImpl.java). My sequence test has an another file due to some additional logic that I needed.

7. The test/wsdl/sequence/build.xml file controls the building of this test. Locate the "compile" target. Add a clause that runs the Wsdl2java code. I would recommend stealing something from the test/wsdl/roundtrip/build.xml file (it does a LOT of wsdl2java and java2wsdl calls). Here is the one for SequenceTest:

8. <!-- Sequence Test -->
   <wsdl2java url="${axis.home}/test/wsdl/sequence/SequenceTest.wsdl"
   output="${axis.home}/build/work"
   deployscope="session"
   skeleton="yes"
   messagecontext="no"
   noimports="no"
   verbose="no"
   testcase="no">
   <mapping namespace="urn:SequenceTest2" package="test.wsdl.sequence"/>
   </wsdl2java>
9. Enable the run target in the new build.xml file. You need to choose from the execute-Component and the (soon to be introduced) execute-Simple-Test target. These control HOW the test is invoked when run as a single component. The execute-Component sets up the tcp-server and http-server prior to running the test, as well as handles deploying and services that may be needed. The execute-Simple-test simply invokes the raw test class file.
10. Done. Run ant functional-tests to verify. Check in your test.
11.

### Using tcpmon to Monitor Functional Tests.

Here is an easy way to monitor the messages while running `functional-tests` (or `all-tests`).
Start up tcpmon listening on 8080 and forwarding to a different port:

```
java org.apache.axis.utils.tcpmon 8080 localhost 8011
```

Run your tests, but use the forwarded port for the SimpleAxisServer, and indicate that functional-tests should continue if a failure occurs.

```
ant functional-tests -Dtest.functional.SimpleAxisPort=8011 -Dtest.functional.fail=no
```

The SOAP messages for all of the tests should appear in the tcpmon window.

`tcpmon` is described in more detail in the Axis User's Guide.

### Using SOAP Monitor to Monitor Functional Tests.

If you are debugging code that is running as a web application using a web application server (such as Tomcat) then you may also use the SOAP Monitor utility to view the SOAP request and response messages.
Start up the SOAP monitor utility by loading the SOAP monitor applet in your web browser window:

```
http://localhost:<port>/axis/SOAPMonitor
```

As you run your tests, the SOAP messages should appear in the SOAP monitor window.

`SOAP Monitor` is described in more detail in the Axis User's Guide.

### Running a Single Functional Test

In one window start the server:

```
java org.apache.axis.transport.http.SimpleAxisServer -p 8080
```

In another window, first deploy the service you're testing:

```
java org.apache.axis.client.AdminClient deploy.wsdd
```

Then bring up the JUnit user interface with your test. For example, to run the the multithread test case:

```
java junit.swingui.TestRunner -noloading test.wsdl.multithread.MultithreadTestCase
```

### Turning on Debug Output

This section is oriented to the Axis default logger: Log4J. For additional information on Log4J, see the section Configuring the Logger.

- Overriding Log4J properties
- The `log4j.properties` file is packaged in `axis.jar` with reasonable default settings. Subsequent items presume changes to these settings. There are multiple options open to the developer, most of which involve extracting `log4j.properties` from `axis.jar` and modifying as appropriate.
  - If you are building and executing `Java` programs from a command line or script file, include the JVM option `-Dlog4j.configuration=`*`yourConfigFile`*.
  - Set `CLASSPATH` such that your version of `log4j.properties` appears prior to `axis.jar` in the `CLASSPATH`.
  - If you are building and executing your programs using `ant` (this includes building Axis and running it's tests), set the environment variable `ANT_OPTS` to `-Dlog4j.configuration=`*`yourConfigFile`*.
  - If you are building Axis, you can change `src/log4j.properties` directly. Be sure NOT to commit your change(s).
- Turning on ALL DEBUG Output
- 
  - Set the `log4j.rootCategory` *priority* to `DEBUG`.
  - Set the *priority* threshold for an appender to `DEBUG` (The `log4j.properties` file in Axis defines two appenders: `CONSOLE` and `LOGFILE`).
- Selective DEBUG Output
- 
  - Set the `log4j.rootCategory` *priority* to `INFO` or higher.
  - Set the `log4j.logger.`*`logger.name`* *priority* to `DEBUG` for the loggers that you are interested in.

- Set the *priority* threshold for an appender to `DEBUG` (The `log4j.properties` file in Axis defines two appenders: `CONSOLE` and `LOGFILE`).
- If you are still seeing more than you want to see, you will need to use other tools to extract the information you are interested in from the log output. Use appropriate key words in log messages and use tools such as `grep` to search for them in log messages.

### Writing Temporary Output

Remember that Axis is targeted for use in a number of open-source and other web applications, and so it needs to be a good citizen. Writing output using `System.out.println` or `System.err.println` should be avoided.

Developers may be tempted to use `System.out.println` while debugging or analyzing a system. If you choose to do this, you will need to disable the `util/TestSrcContent` test, which enforces avoidance of `System.out.println` and `System.err.println`. It follows that you will need to remove your statements before checking the code back in.

As an alternative, we strongly encourage you to take a few moments and introduce debug statements: `log.debug("reasonably terse and meaningful message")`. If a debug message is useful for understanding a problem now, it may be useful again in the future to you or a peer.

## Adding Testcases

See Also: Test and Samples Structure

**Editor's Note:** We need more effort to streamline and simplify the addition of tests. We also need to think about categorizing tests as the test bucket grows.

If you make changes to Axis, please add a test that uses your change. Why?

- The test validates that your new code works.
- The test protects your change from bugs introduced by future code changes.
- The test is an example to users of the features of Axis.
- The test can be used as a starting point for new development.

Some general principles:

- Tests should be self-explanatory.
- Tests should not generate an abundance of output
- Tests should hook into the existing junit framework.
- Each test or group of related tests should have its own directory in the xml-axis/java/test directory

One way to build a test is to "cut and paste" the existing tests, and then modify the test to suit your needs. This approach is becoming more complicated as the different kinds of tests grow.

A good "non-wsdl" test for reference is test/saaj.

### Test Structure

The Test and Samples Redesign Document is here

As of Axis 1.0, RC1, we have moved to a "componentized" test structure. Instead of having one high-level large recursive function, there are smaller, simple "component" build.xml files in the leaf level of the test/** and samples/** trees.

These "component" files have a common layout. Their primary targets are:

- clean - reset the build destination(s)
- compile - javac, wsdl2java, java2wsdl instructions
- run - "executes" the test

A "sample" test xml file can be found in test/templateTest

### Adding Source Code Checks

The Axis build performs certain automated checks of the files in the source directory (java/src) to make sure certain conventions are followed such as using internationalised strings when issuing messages.

If a convention can be reduced to a regular expression match, it can be enforced at build time by updating java/test/utils/TestSrcContent.java.

All that is necessary is to add a pattern to the static FileNameContentPattern array. Each pattern has three parameters:

1. a pattern that matches filenames that are to be checked,
2. a pattern to be searched for in the chosen files, and
3. a boolean indicating whether the pattern is to be allowed (typically false indicating not allowed).

A reasonable summary of the regular expression notation is provided in the Jakarta ORO javadocs.

### JUnit and Axis

You try to run some JUnit tests on an Axis client that invokes a web service, and you always get this exception:

```
java.lang.ExceptionInInitializerError at
org.apache.axis.client.Service.<init>(Service.java:108) ... Caused by:
org.apache.commons.logging.LogConfigurationException: ...
org.apache.commons.logging.impl.Jdk14Logger does not implement Log at
org.apache.commons.logging.impl.LogFactoryImpl.newInstance
(LogFactoryImpl.java:555) ...
```

Actually, the Jdk14Logger does implement Log. What you have is a JUnit classloading issue. JUnit's graphical TestRunner has a feature where it will dynamically reload modified classes every time the user presses the "Run" button. This way, the user doesn't need to relaunch the TestRunner after every edit. For this, JUnit uses its own classloader, junit.runner.TestCaseClassLoader. As of JUnit 3.8.1, confusion can arise between TestCaseClassLoader and the system class loader as to which loader did or should load which classes.

There are two ways to avoid this problem.

- Sure and simple fix. Turn off dynamic class reloading by running junit.swingui.TestRunner with the -noloading argument.
- Finicky and fancy fix, only necessary if you want dynamic class reloading. Tell TestCaseClassLoader to ignore certain packages and their sub-packages, deferring them to the system classloader. You can do this using a file located in junit.jar, junit/runner/excluded.properties. Its content appears as follows: # # The list of excluded package paths for the TestCaseClassLoader # excluded.0=sun.* excluded.1=com.sun.* excluded.2=org.omg.* excluded.3=javax.* excluded.4=sunw.* excluded.5=java.* excluded.6=org.w3c.dom.* excluded.7=org.xml.sax.* excluded.8=net.jini.*

Copy this file, preserving the directory path, into another location, e.g. deployDir. So the copied properties file's path will be deployDir/junit/runner/excluded.properties. Add an extra entry to the end of this file:

```
excluded.9=org.apache.*
```

Edit your classpath so that deployDir appears before junit.jar. This way, the modified excluded.properties will be used, rather than the default. (Don't add the path to excluded.properties itself to the classpath.)

This fix will prevent the commons-logging exception. However, other classloading problems might still arise. For example:

```
Dec 10, 2002 7:16:16 PM org.apache.axis.encoding.ser.BeanPropertyTarget set
SEVERE: Could not convert [Lfoo.bar.Child; to bean field 'childrenAsArray', type
[Lfoo.bar.Child; Dec 10, 2002 7:16:16 PM org.apache.axis.client.Call invoke
SEVERE: Exception: java.lang.IllegalArgumentException: argument type mismatch
at org.apache.axis.encoding.ser.BeanPropertyTarget.set
(BeanPropertyTarget.java:182) at
org.apache.axis.encoding.DeserializerImpl.valueComplete
(DeserializerImpl.java:284) ...
```

In this case, you have no choice but to give up on dynamic class reloading and use the -noloading argument.

One other heads-up about JUnit testing of an Axis web service. Suppose you have run JUnit tests locally on the component that you want to expose as a web service. You press the "Run" button to initiate a series of tests. Between each test, all your data structures are re-initialized. Your tests produce a long green bar. Good.

Suppose you now want to run JUnit tests on an Axis client that is connecting to an application server running the Axis web application and with it your web service. Between each test, JUnit will automatically re-initialize your client.

Your server-side data structures are a different matter. If you're checking your server data at the end of each test (as you should be) and you run more than one test at a time, the second and later tests will fail because they are generating cumulative data on the Axis server based on preceding tests rather than fresh data based only on the current one.

This means that, for each test, you must manually re-initialize your web service. One way to accomplish this is to add to your web service interface a re-initialize operation. Then have the client call that operation at the start of each test.

#### Debugging

#### Running the JAX-RPC Compatibility Tests

As well as a specification, JAX-RPC has a Technology Compatibility Kit (TCK) which is available to members of the JAX-RPC Expert Group (and others?).

The kit comes as a zip file which you should unzip into a directory of your choosing. The installation instructions are in the JAX-RPC Release Notes document which is stored in the docs directory. If you open the index.html file in the docs directory using a web browser, you'll see a list of all the documents supplied with the kit.

Note that the kit includes the JavaTest test harness which is used for running the compatibility tests.

If any more information is needed about running these tests, please add it here!

### 1.4.5. Axis System Integration Guide

#### 1.4.5.1. Axis System Integration Guide

*1.2 Version*

*Feedback: axis-dev@ws.apache.org*

## Introduction

The primary purpose of this guide is to present how Axis can be integrated into an existing web application server, such as Tomcat or WebSphere, for example. Axis has a number of Pluggable APIs that are necessary for such an integration.

The reader may find useful background information in the Architecture Guide.

## Pluggable APIs

The following are the points that are pluggable in order to integrate Axis into a web application server. The first subsection details a number of pluggable components in general. More details are provided for other components in the remaining subsections.

### Components

This section describes in general how to plug specializations of various components into Axis.

### General Strategy

To override the default behavior for a pluggable component:

- Develop implementation of components interface
- 
- Define the implementation class to Axis by either creating a service definition file (prefered) or by setting a system property.
  - **PREFERED:** To create a service definition file:
  - - The name of the service definition file is derived from the interface or abstract class which the service implements/extends:
      `/META-INF/services/<componentPackage>.<interfaceName>`.
    - Put the fully qualified class name of the implementation class on a line by itself in the service definition file.
  - Set system property:
    - The name of the system property is the name of the interface.
    - The value of the system property is the name of the implementation.
    - The optional system property name (in table, below) may be also be used.
    - 
    - Setting a system property is not prefered, particularly in a J2EE or other application hosting environment, because it imposes a directive across all

applications. This may or may not be appropriate behavior. If it is to be done, it should never be done from within a Web Application at runtime.

- Package the implementation class and, if used, the service definition file in a JAR file and/or place it where it can be picked up by a class loader (CLASSPATH).

**Example 1**

To override the default behavior for the Java Compiler:

- An implementation of the `Compiler` interface is already provided for the `Jikes` compiler.
- 
- Create the service definition file named:

  ```
  /META-INF/services/org.apache.axis.components.compiler.Compiler
  ```
- 
- Add the following line to the service definition file:
  ```
  org.apache.axis.components.compiler.Jikes
  ```
- 
- Since `org.apache.axis.components.compiler.Jikes` is packaged with Axis, all that needs to be done is to ensure that the service definition file is loadable by a class loader.

**Example 2**

To override the default behavior for the SocketFactory in an environment that does not allow resources to be located/loaded appropriately, or where the behavior needs to be forced to a specific implementation:

- Provide an implementation of the `SocketFactory` interface, for example `your.package.YourSocketFactory`
- 
- Set the system property named
  ```
  org.apache.axis.components.net.SocketFactory
  ```
  to the value
  ```
  your.package.YourSocketFactory
  ```
  This can be done by using the JVM commandline

  ```
  -Dorg.apache.axis.components.net.SocketFactory=your.package.YourSocket
  ```

- Ensure that the implementation class is loadable by a class loader.

### Reference

(Component/Package: org.apache.axis.components.*)

| Component Package | Factory | Interface | Optional System Property | Default Implementation |
|---|---|---|---|---|
| compiler | CompilerFactory getCompiler() | Compiler | axis.Compiler | Javac |
| image | ImageIOFactory getImageIO() | ImageIO | axis.ImageIO | MerlinIO, JimiIO, JDK13IO |
| jms | JMSVendorAdapterFactory getJMSVendorAdapter() | JMSVendorAdapter | | JNDIVendorAdapter |
| net | SocketFactoryFactory getFactory() | SocketFactory | axis.socketFactory | DefaultSocketFactory |
| net | SocketFactoryFactory getSecureFactory() | SecureSocketFactory | axis.socketSecureFactory | JSSESocketFactory |

### Logging/Tracing

Axis logging and tracing is based on the Logging component of the Jakarta Commons project, or the Jakarta Commons Logging (JCL) SPI. The JCL provides a Log interface with thin-wrapper implementations for other logging tools, including Log4J, Avalon LogKit, and JDK 1.4. The interface maps closely to Log4J and LogKit.

### Justification/Rationale

A pluggable logging/trace facility enables Axis to direct logging/trace messages to a host web application server's logging facility. A central logging facility with a single point of configuration/control is superior to distinct logging mechanisms for each of a multitude of middleware components that are to be integrated into a web application server.

### Integration

The minimum requirement to integrate with another logger is to provide an implementation of the `org.apache.commons.logging.Log` interface. In addition, an implementation of the `org.apache.commons.logging.LogFactory` interface can be provided to meet specific requirements for connecting to, or instantiating, a logger.

- org.apache.commons.logging.Log
- The `Log` interface defines the following methods for use in writing log/trace messages to

the log: log.fatal(Object message); log.fatal(Object message, Throwable t); log.error(Object message); log.error(Object message, Throwable t); log.warn(Object message); log.warn(Object message, Throwable t); log.info(Object message); log.info(Object message, Throwable t); log.debug(Object message); log.debug(Object message, Throwable t); log.trace(Object message); log.trace(Object message, Throwable t); log.isFatalEnabled(); log.isErrorEnabled(); log.isWarnEnabled(); log.isInfoEnabled(); log.isDebugEnabled(); log.isTraceEnabled();

Semantics for these methods are such that it is expected that the severity of messages is ordered, from highest to lowest:

- fatal - Consider logging to console and system log.
- error - Consider logging to console and system log.
- warn - Consider logging to console and system log.
- info - Consider logging to console and system log.
- debug - Log to system log, if enabled.
- trace - Log to system log, if enabled.

- org.apache.commons.logging.LogFactory
- If desired, the default implementation of the `org.apache.commons.logging.LogFactory` interface can be overridden, allowing the JDK 1.3 Service Provider discovery process to locate and create a LogFactory specific to the needs of the application. Review the Javadoc for the `LogFactoryImpl.java` for details.

**Mechanism**

- Life cycle
- The JCL LogFactory implementation must assume responsibility for either connecting/disconnecting to a logging toolkit, or instantiating/initializing/destroying a logging toolkit.

- Exception handling
- The JCL Log interface doesn't specify any exceptions to be handled, the implementation must catch any exceptions.

- Multiple threads
- The JCL Log and LogFactory implementations must ensure that any synchronization required by the logging toolkit is met.

**Logger Configuration**

- Log

- The default `LogFactory` provided by JCL can be configured to instantiate a specific implementation of the `org.apache.commons.logging.Log` interface by setting the property `org.apache.commons.logging.Log`. This property can be specified as a system property, or in the `commons-logging.properties` file, which must exist in the CLASSPATH.

- Default logger if not plugged
- The Jakarta Commons Logging SPI uses the implementation of the `org.apache.commons.logging.Log` interface specified by the system property `org.apache.commons.logging.Log`. If the property is not specified or the class is not available then the JCL provides access to a default logging toolkit by searching the CLASSPATH for the following toolkits, in order of preference:
  - Log4J
  - JDK 1.4
  - JCL SimpleLog

**Configuration**

The internal data model used by Axis is based on an Axis specific data model: Web Services Deployment Descriptor (WSDD). Axis initially obtains the WSDD information for a service from an instance of `org.apache.axis.EngineConfiguration`.

The EngineConfiguration is provided by an implementation of the interface `org.apache.axis.EngineConfigurationFactory`, which currently provides methods that return client and server configurations.

Our focus will be how to define the implementation class for `EngineConfigurationFactory`.

- Justification/Rationale
- While the default behaviour is sufficient for general use of Axis, integrating Axis into an existing application server may require an alternate deployment model. A customized implementation of the EngineConfigurationFactory would map from the hosts deployment model to Axis's internal deployment model.

- Mechanism
- The relevant sequence of instructions used to obtain configuration information and initialize Axis is as follows:

```
EngineConfigurationFactory factory =
EngineConfigurationFactoryFinder(someContext);
EngineCongfiguration config =
factory.getClientEngineConfig();
```

`AxisClient = new AxisClient(config);`
The details may vary (server versus client, whether other factories are involved, etc). Regardless, the point is that integration code is responsible for calling `EngineConfigurationFactoryFinder(someContext)` and ensuring that the results are handed to Axis. `someContext` is key to how the factory finder locates the appropriate implementation of EngineConfigurationFactory to be used, if any.

EngineConfigurationFactoryFinder works as follows:

- Obtain a list of classes that implement `org.apache.axis.EngineConfigurationFactory`, in the following order:
    - The value of the system property `axis.EngineConfigFactory`.
    - The value of the system property `org.apache.axis.EngineConfigurationFactory`.
    - Locate all resources named `META-INF/services/org.apache.axis.EngineConfigurationFactory`. Each line of such a resource identifies the name of a class implementing the interface ('#' comments, through end-of-line).
    - `org.apache.axis.configuration.EngineConfigurationFactoryServlet`
    - `org.apache.axis.configuration.EngineConfigurationFactoryDefault`
- Classes implementing EngineConfigurationFactory are required to provide the method
  `public static EngineConfigurationFactory newFactory(Object)`
  This method is called, passing `someContext` as the parameter.
- 
- The `newFactory` method is required to check the `someContext` parameter to determine if it is meaningfull to the class (at a minimum, verify that it is of an expected type, or class) and may, in addition, examine the overall runtime environment. If the environment can provide information required by an EngineConfigurationFactory, then the `newFactory()` may return in instance of that factory. Otherwise, `newFactory()` must return null.
- 
- EngineConfigurationFactoryFinder returns the first non-null factory it obtains.

- Default behavior
- The default behaviour is provided by the last two elements of the list of implementing classes, as described above:
    - `org.apache.axis.configuration.EngineConfigurationFactoryServlet`
      `newFactory(obj)` is called. If `obj instanceof`

`javax.servlet.ServletContext` is true, then an instance of this class is returned.

The default Servlet factory is expected to function as a server (as a client it will incorrectly attempt to load the WSDD file `client-config.wsdd` from the current working directory!).

The default Servlet factory will open the Web Application resource `/WEB-INF/server-config.wsdd` (The name of this file may be changed using the system property `axis.ServerConfigFile`):

- If it exists as an accessible file (i.e. not in a JAR/WAR file), then it opens it as a file. This allows changes to be saved, if changes are allowed & made using the Admin tools.
- If it does not exist as a file, then an attempt is made to access it as a resource stream (getResourceAsStream), which works for JAR/WAR file contents.
- If the resource is simply not available, an attempt is made to create it as a file.
- If all above attempts fail, a final attempt is made to access `org.apache.axis.server.server-config.wsdd` as a data stream.

- 
- `org.apache.axis.configuration.EngineConfigurationFactoryDefault` `newFactory(obj)` is called. If `obj` is null then an instance of this class is returned. A non-null `obj` is presumed to require a non-default factory.

  The default factory will load the WSDD files `client-config.wsdd` or `server-config.wsdd`, as appropriate, from the current working directory. The names of these files may be changed using the system properties `axis.ClientConfigFile` and `axis.ServerConfigFile`, respectively.

### Handlers

See the Architecture Guide for current information on Handlers.

### Internationalization

Axis supports internationalization by providing both a property file of the strings used in Axis, and an extension mechanism that facilitates accessing internal Axis messages and extending the messages available to integration code based on existing Axis code.

#### Translation

- Justification/Rationale
- In order for readers of languages other than English to be comfortable with Axis, we provide a mechanism for the strings used in Axis to be translated. We do not provide any

translations in Axis; we merely provide a means by which translators can easily plug in their translations.

- Mechanism
- Axis provides english messages in the Java resource named org.apache.axis.i18n.resource.properties (in the source tree, the file is named xml-axis/java/src/org/apache/axis/i18n/resource.properties).

  Axis makes use of the Java internationalization mechanism - i.e., a java.util.ResourceBundle backed by a properties file - and the java.text.MessageFormat class to substitute parameters into the message text.

  - java.util.ResourceBundle retrieves message text from a property file using a key provided by the program. Entries in a message resource file are of the form <key>=<message>.
  -
  - java.text.MessageFormat substitutes variables for markers in the message text. Markers use the syntax "{X}" where `X` is the number of the variable, starting at 0.

  For example: `myMsg00=My {0} is {1}.`

  Translation requires creating an alternate version of the property file provided by Axis for a target language. The JavaDoc for `java.utils.ResourceBundle` provides details on how to identify different property files for different locales.

  For details on using Axis's internationalization tools, see the Developer's Guide.

- Default behavior
- The default behavior, meaning what happens when a translated file doesn't exist for a given locale, is to fall back on the English-language properties file. If that file doesn't exist (unlikely unless something is seriously wrong), Axis will throw an exception with an English-language reason message.

### Extending Message Files

Axis provides a Message file extension mechanism that allows Axis-based code to use Axis message keys, as well as new message keys unique to the extended code.

- Justification/Rationale
- Axis provides pluggable interfaces for various Axis entities, including EngineConfigurationFactory's, Provides, and Handlers. Axis also provides a variety of implementations of these entities. It is convenient to use Axis source code for such implementations as starting points for developing extentions and customizations that

fulfill the unique needs of the end user.

- Procedure
- To extend the Axis message file:
  - Copy the Axis source file `java/src/org/apache/axis/i18n/Messages.java` to your project/package, say `my/project/package/path/Messages.java`.
  - 
    - Set the `package` declaration in the copied file to the correct package name.
    - Set the private attribute `projectName` to `"my.project"`: the portion of the package name that is common to your project. `projectName` must be equal to or be a prefix of the copied Messages package name.
  - Create the file `my/project/package/path/resource.properties`. Add new message key/value pairs to this file.
  - 
  - As you copy Axis source files over to your project, change the `import org.apache.axis.i18n.Messages` statement to `import my.project.package.path.Messages`.
  - Use the methods provided by the class Messages, as discussed in the Developer's Guide, to access the new messages.
- Behavior
- 
  - Local Search
  - `Messages` begins looking for a key's value in the `resources.properties` resource in it's (Messages) package.
  - Hierarchical Search
  - If `Messages` cannot locate either the key, or the resource file, it walks up the package hierarchy until it finds it. The top of the hierarchy, above which it will not search, is defined by the `projectName` attribute, set above.
  - Default behavior
  - If the key cannot be found in the package hierarchy then a default resource is used. The default behaviour is determined by the `parent` attribute of the `Messages` class copied to your extensions directory.

    Unless changed, the default behavior, meaning what happens when a key isn't defined in the new properties file, is to fall back to the Axis properties file (org.apache.axis.i18n.resource.properties).

### Performance Monitoring

Axis does not yet include specific Performance Monitoring Plugs.

### Encoding

Axis does not yet include an Encoding Plug.

### WSDL Parser and Code Generator Framework

WSDL2Java is Axis's tool to generate Java artifacts from WSDL. This tool is extensible. If users of Axis wish to extend Axis, then they may also need to extend or change the generated artifacts. For example, if Axis is inserted into some product which has an existing deployment model that's different than Axis's deployment model, then that product's version of WSDL2Java will be required to generate deployment descriptors other than Axis's deploy.wsdd.

What follows immediately is a description of the framework. If you would rather dive down into the dirt of examples, you could learn a good deal just from them. Then you could come back up here and learn the gory details.

There are three parts to WSDL2Java:

1. The symbol table
2. The parser front end with a generator framework
3. The code generator back end (WSDL2Java itself)

### Symbol Table

The symbol table, found in org.apache.axis.wsdl.symbolTable, will contain all the symbols from a WSDL document, both the symbols from the WSDL constructs themselves (portType, binding, etc), and also the XML schema types that the WSDL refers to.

NOTE: Needs lots of description here.

The symbol table is not extensible, but you **can** add fields to it by using the Dynamic Variables construct:

- You must have some constant object for a dynamic variable key. For example: public static final String MY_KEY = "my key";
- You set the value of the variable in your GeneratorFactory.generatorPass: entry.setDynamicVar(MY_KEY, myValue);
- You get the value of the variable in your generators: Object myValue = entry.getDynamicVar(MY_KEY);

## Parser Front End and Generator Framework

The parser front end and generator framework is located in org.apache.axis.wsdl.gen. The parser front end consists of two files:

- Parser
- public class Parser {

```
    public Parser();
    public boolean isDebug();
    public void setDebug(boolean);
    public boolean isImports();
    public void setImports(boolean);
    public boolean isVerbose();
    public void setVerbose(boolean);
    public long getTimeout();
    public void setTimeout(long);
    public java.lang.String getUsername();
    public void setUsername(java.lang.String);
    public java.lang.String getPassword();
    public void setPassword(java.lang.String);
    public GeneratorFactory getFactory();
    public void setFactory(GeneratorFactory);
    public org.apache.axis.wsdl.symbolTable.SymbolTable getSymbolTable();
    public javax.wsdl.Definition getCurrentDefinition();
    public java.lang.String getWSDLURI();
    public void run(String wsdl) throws java.lang.Exception;
    public void run(String context, org.w3c.dom.Document wsdlDoc) throws
java.io.IOException, javax.wsdl.WSDLException;
}
```

The basic behavior of this class is simple: you instantiate a Parser, then you run it.

```
Parser parser = new Parser();
parser.run("myfile.wsdl");
```

There are various options on the parser that have accessor methods:

- debug - default is false - dump the symbol table after the WSDL file has been parsed
- imports - default is true - should imported files be visited?
- verbose - default is false - list each file as it is being parsed
- timeout - default is 45 - the number of seconds to wait before halting the parse
- username - no default - needed for protected URI's
- password - no default - needed for protected URI's

Other miscellaneous methods on the parser:

- get/setFactory - get or set the GeneratorFactory on this parser - see below for details. The default generator factory is NoopFactory, which generates nothing.
- getSymbolTable - once a run method is called, the symbol table will be populated and can get queried.
- getCurrentDefinition - once a run method is called, the parser will contain a Definition object which represents the given wsdl file. Definition is a WSDL4J object.
- getWSDLURI - once the run method which takes a string is called, the parser will contain the string representing the location of the WSDL file. Note that the other run method - run(String context, Document wsdlDoc) - does not provide a location for the wsdl file. If this run method is used, getWSDLURI will be null.
- There are two run methods. The first, as shown above, takes a URI string which represents the location of the WSDL file. If you've already parsed the WSDL file into an XML Document, then you can use the second run method, which takes a context and the WSDL Document.

An extension of this class would ...
NOTE: continue this sentiment...

- WSDL2
- Parser is the programmatic interface into the WSDL parser. WSDL2 is the command line tool for the parser. It provides an extensible framework for calling the Parser from the command line. It is named WSDL2 because extensions of it will likely begin with WSDL2: **WSDL2**Java, **WSDL2**Lisp, **WSDL2**XXX.

```
public class WSDL2 {
   protected WSDL2();
   protected Parser createParser();
   protected Parser getParser();
   protected void addOptions(org.apache.axis.utils.CLOptionDescriptor[]);
   protected void parseOption(org.apache.axis.utils.CLOption);
   protected void validateOptions();
   protected void printUsage();
   protected void run(String[]);
   public static void main(String[]);
}
```

Like all good command line tools, it has a main method. Unlike some command line tools, however, its methods are not static. Static methods are not extensible. WSDL2's main method constructs an instance of itself and calls methods on that instance rather than calling static methods. These methods follow a behavior pattern. The main method is very simple:

```
public static void main(String[] args) {
 WSDL2 wsdl2 = new WSDL2();
 wsdl2.run(args);
}
```

The constructor calls createParser to construct a Parser or an extension of Parser.

run calls:

- parseOption to parse each command line option and call the appropriate Parser accessor. For example, when this method parses --verbose, it calls parser.setVerbose(true)
- validateOptions to make sure all the option values are consistent
- printUsage if the usage of the tool is in error
- parser.run(args);

If an extension has additional options, then it is expected to call addOptions before calling run. So extensions will call, as necessary, getParser, addOptions, run. Extensions will override, as necessary, createParser, parseOption, validateOptions, printUsage.

The generator framework consists of 2 files:

- Generator
- The Generator interface is very simple. It just defines a generate method.

  ```
  public interface Generator
  {
     public void generate() throws java.io.IOException;
  }
  ```

- GeneratorFactory

-
  ```
  public interface GeneratorFactory
  {
     public void generatorPass(javax.wsdl.Definition, SymbolTable);
     public Generator getGenerator(javax.wsdl.Message, SymbolTable);
     public Generator getGenerator(javax.wsdl.PortType, SymbolTable);
     public Generator getGenerator(javax.wsdl.Binding, SymbolTable);
     public Generator getGenerator(javax.wsdl.Service, SymbolTable);
     public Generator getGenerator(TypeEntry, SymbolTable);
     public Generator getGenerator(javax.wsdl.Definition, SymbolTable);
  ```

```
    public void setBaseTypeMapping(BaseTypeMapping);
    public BaseTypeMapping getBaseTypeMapping();
}
```

The GeneratorFactory interface defines a set of methods that the parser uses to get generators. There should be a generator for each of the WSDL constructs (message, portType, etc - note that these depend on the WSDL4J classes: javax.xml.Message, javax.xml.PortType, etc); a generator for schema types; and a generator for the WSDL Definition itself. This last generator is used to generate anything that doesn't fit into the previous categories.

In addition to the getGeneratorMethods, the GeneratorFactory defines a generatorPass method which provides the factory implementation a chance to walk through the symbol table to do any preprocessing before the actual generation begins.

Accessors for the base type mapping are also defined. These are used to translate QNames to base types in the given target mapping.

In addition to Parser, WSDL2, Generator, and GeneratorFactory, the org.apache.axis.wsdl.gen package also contains a couple of no-op classes: NoopGenerator and NoopFactory. NoopGenerator is a convenience class for extensions that do not need to generate artifacts for every WSDL construct. For example, WSDL2Java does not generate anything for messages, therefore its factory's getGenerator(Message, SymbolTable) method returns an instance of NoopGenerator. NoopFactory returns a NoopGenerator for all getGenerator methods. The default factory for Parser is the NoopFactory.

### Code Generator Back End

The meat of the WSDL2Java back end generators is in org.apache.axis.wsdl.toJava. Emitter extends Parser. org.apache.axis.wsdl.WSDL2Java extends WSDL2. JavaGeneratorFactory implements GeneratorFactory. And the various JavaXXXWriter classes implement the Generator interface.

NOTE: Need lots more description here...

### WSDL Framework Extension Examples

Everything above sounds rather complex. It is, but that doesn't mean your extension has to be.

#### Example 1 - Simple extension of WSDL2Java - additional artifact

The simplest extension of the framework is one which generates everything that WSDL2Java already generates, plus something new. Example 1 is such an extension. It's extra artifact is a file for each service that lists that service's ports. I don't know why you'd want to do this, but it makes for a good, simple example. See samples/integrationGuide/example1 for the complete implementation of this example.

- First you must create your writer that writes the new artifact. This new class extends org.apache.axis.wsdl.toJava.JavaWriter. JavaWriter dictates behavior to its extensions; it calls writeFileHeader and writeFileBody. Since we don't care about a file header for this example, writeFileHeader is a no-op method. writeFileBody does the real work of this writer.
- 
```
public class MyListPortsWriter extends JavaWriter {
    private Service service;
    public MyListPortsWriter(
      Emitter emitter,
      ServiceEntry sEntry,
      SymbolTable symbolTable) {
     super(emitter,
       new QName(
        sEntry.getQName().getNamespaceURI(),
        sEntry.getQName().getLocalPart() + "Lst"),
       "", "lst", "Generating service port list file", "service list");
     this.service = sEntry.getService();
    }
    protected void writeFileHeader() throws IOException {
    }
    protected void writeFileBody() throws IOException {
     Map portMap = service.getPorts();
     Iterator portIterator = portMap.values().iterator();

     while (portIterator.hasNext()) {
      Port p = (Port) portIterator.next();
      pw.println(p.getName());
     }
     pw.close();
    }
}
```

- Then you need a main program. This main program extends WSDL2Java so that it gets all the functionality of that tool. The main of this tool does 3 things:

- 
    - instantiates itself
    - adds MyListPortsWriter to the list of generators for a WSDL service
    - calls the run method.

  That's it!  The base tool does all the rest of the work.

```
public class MyWSDL2Java extends WSDL2Java {

    public static void main(String args[]) {
     MyWSDL2Java myWSDL2Java = new MyWSDL2Java();

     JavaGeneratorFactory factory =
       (JavaGeneratorFactory) myWSDL2Java.getParser().getFactory();
     factory.addGenerator(Service.class, MyListPortsWriter.class);

     myWSDL2Java.run(args);
    }
}
```

**Example 2 - Not quite as simple an extension of WSDL2Java - change an artifact**

In this example, we'll replace deploy.wsdd with mydeploy.useless.  For brevity, mydeploy.useless is rather useless.  Making it useful is an exercise left to the reader.  See samples/integrationGuide/example2 for the complete implementation of this example.

- First, here is the writer for the mydeploy.useless.  This new class extends org.apache.axis.wsdl.toJava.JavaWriter.  JavaWriter dictates behavior to its extensions; it calls writeFileHeader and writeFileBody.  Since we don't care about a file header for this example, writeFileHeader is a no-op method.  writeFileBody does the real work of this writer.  It simply writes a bit of a song, depending on user input.
- Note that we've also overridden the generate method.  The parser always calls generate, but since this is a server-side artifact, we don't want to generate it unless we are generating server-side artifacts (in other words, in terms of the command line options, we've specified the --serverSide option).

```
public class MyDeployWriter extends JavaWriter {
    public MyDeployWriter(Emitter emitter, Definition definition,
      SymbolTable symbolTable) {
     super(emitter,
       new QName(definition.getTargetNamespace(), "deploy"),
       "", "useless", "Generating deploy.useless", "deploy");
    }
    public void generate() throws IOException {
     if (emitter.isServerSide()) {
      super.generate();
     }
    }
    protected void writeFileHeader() throws IOException {
    }
    protected void writeFileBody() throws IOException {
     MyEmitter myEmitter = (MyEmitter) emitter;
     if (myEmitter.getSong() == MyEmitter.RUM) {
      pw.println("Yo!  Ho!  Ho!  And a bottle of rum.");
     }
     else if (myEmitter.getSong() == MyEmitter.WORK) {
      pw.println("Hi ho!  Hi ho!  It's off to work we go.");
     }
     else {
      pw.println("Feelings...  Nothing more than feelings...");
     }
     pw.close();
    }
}
```

- Since we're changing what WSDL2Java generates, rather than simply adding to it like the previous example did, calling addGenerator isn't good enough. In order to change what WSDL2Java generates, you have to create a generator factory and provide your own generators. Since we want to keep most of WSDL2Java's artifacts, we can simply extend WSDL2Java's factory - JavaGeneratorFactory - and override the addDefinitionGenerators method.
- 
```
public class MyGeneratorFactory extends JavaGeneratorFactory {
    protected void addDefinitionGenerators() {
     addGenerator(Definition.class, JavaDefinitionWriter.class); // WSDL2Java's
JavaDefinitionWriter
     addGenerator(Definition.class, MyDeployWriter.class); // our DeployWriter
     addGenerator(Definition.class, JavaUndeployWriter.class); // WSDL2Java's
JavaUndeployWriter
    }
}
```

- Now we must write the API's to our tool. Since we've added an option - song - we need both the programmatic API - an extension of Parser (actually Emitter in this case since we're extending WSDL2Java and Emitter is WSDL2Java's parser extension) - and the command line API.

- Here is our programmatic API. It adds song accessors to Emitter. It also, in the constructor, lets the factory know about the emitter and the emitter know about the factory.

```
public class MyEmitter extends Emitter {
    public static final int RUM  = 0;
    public static final int WORK = 1;
    private int song = -1;

    public MyEmitter() {
     MyGeneratorFactory factory = new MyGeneratorFactory();
     setFactory(factory);
     factory.setEmitter(this);
    }
    public int getSong() {
     return song;
    }
    public void setSong(int song) {
     this.song = song;
    }
}
```

And here is our command line API. It's a bit more complex that our previous example's main program, but it does 2 extra things:

1. accept a new command line option: --song rum|work (this is the biggest chunk of the new work).
2. create a new subclass of Parser

```
public class WSDL2Useless extends WSDL2Java {
    protected static final int SONG_OPT = 'g';
    protected static final CLOptionDescriptor[] options = new CLOptionDescriptor[]{
    new CLOptionDescriptor("song",
      CLOptionDescriptor.ARGUMENT_REQUIRED,
      SONG_OPT,
      "Choose a song for deploy.useless:  work or rum")
    };

    public WSDL2Useless() {
     addOptions(options);
    }
    protected Parser createParser() {
     return new MyEmitter();
    }
    protected void parseOption(CLOption option) {
     if (option.getId() == SONG_OPT) {
      String arg = option.getArgument();
      if (arg.equals("rum")) {
      ((MyEmitter) parser).setSong(MyEmitter.RUM);
      }
      else if (arg.equals("work")) {
      ((MyEmitter) parser).setSong(MyEmitter.WORK);
      }
     }
     else {
     super.parseOption(option);
     }
    }
    public static void main(String args[]) {
     WSDL2Useless useless = new WSDL2Useless();

     useless.run(args);
    }
}
```

Let's go through this one method at a time.

- constructor - this constructor adds the new option --song rum|work. (the abbreviated version of this option is "-g", rather an odd abbreviation, but "-s" is the abbreviation for --serverSide and "-S" is the abbreviation for --skeletonDeploy. Bummer. I just picked some other letter.)
- createParser - we've got to provide a means by which the parent class can get our Parser extension.
- parseOption - this method processes our new option. If the given option isn't ours, just let super.parseOption do its work.
- main - this main is actually simpler than the first example's main. The first main had to add our generator to the list of generators. In this example, the factory already did that, so all that this main must do is instantiate itself and run itself.

## Client SSL

The default pluggable secure socket factory module (see Pluggable APIs) uses JSSE security. Review the JSSE documentation for details on installing, registering, and configuring JSSE for your runtime environment.

## 1.4.6. Axis Architecture Guide

### 1.4.6.1. Axis Architecture Guide

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

#### Introduction

This guide records some of the rationale of the architecture and design of Axis.

#### Architectural Overview

Axis consists of several subsystems working together, as we shall see later. In this section we'll give you an overview of how the core of Axis works.

##### Handlers and the Message Path in Axis

Put simply, Axis is all about processing Messages. When the central Axis processing logic runs, a series of **Handlers** are each invoked in order. The particular order is determined by two factors - deployment configuration and whether the engine is a client or a server. The object which is passed to each Handler invocation is a **MessageContext**. A MessageContext is a structure which contains several important parts: 1) a "request" message, 2) a "response" message, and 3) a bag of properties. More on this in a bit.

There are two basic ways in which Axis is invoked:

1. As a **server**, a **Transport Listener** will create a MessageContext and invoke the Axis processing framework.
2. As a **client**, application code (usually aided by the client programming model of Axis) will generate a MessageContext and invoke the Axis processing framework.

In either case, the Axis framework's job is simply to pass the resulting MessageContext through the configured set of Handlers, each of which has an opportunity to do whatever it is designed to do with the MessageContext.

##### Message Path on the Server

The server side message path is shown in the following diagram. The small cylinders represent Handlers and the larger, enclosing cylinders represent **Chains** (ordered collections of Handlers which will be described shortly).
A message arrives (in some protocol-specific manner) at a Transport Listener. In this case,

let's assume the Listener is a HTTP servlet. It's the Listener's job to package the protocol-specific data into a **Message** object (org.apache.axis.Message), and put the Message into a **MessageContext**. The MessageContext is also loaded with various **properties** by the Listener - in this example the property "http.SOAPAction" would be set to the value of the SOAPAction HTTP header. The Transport Listener also sets the **transportName** String on the MessageContext , in this case to "http". Once the MessageContext is ready to go, the Listener hands it to the AxisEngine.

The AxisEngine's first job is to look up the transport by name. The transport is an object which contains a **request** Chain, a **response** Chain, or perhaps both. A **Chain** is a Handler consisting of a sequence of Handlers which are invoked in turn -- more on Chains later. If a transport request Chain exists, it will be invoked, passing the MessageContext into the invoke() method. This will result in calling all the Handlers specified in the request Chain configuration.

After the transport request Handler, the engine locates a global request Chain, if configured, and then invokes any Handlers specified therein.

At some point during the processing up until now, some Handler has hopefully set the **serviceHandler** field of the MessageContext (this is usually done in the HTTP transport by the "URLMapper" Handler, which maps a URL like "http://localhost/axis/services/AdminService" to the "AdminService" service). This field determines the Handler we'll invoke to execute service-specific functionality, such as making an RPC call on a back-end object. Services in Axis are typically instances of the "SOAPService" class (org.apache.axis.handlers.soap.SOAPService), which may contain **request** and **response** Chains (similar to what we saw at the transport and global levels), and must contain a **provider**, which is simply a Handler responsible for implementing the actual back end logic of the service.

For RPC-style requests, the provider is the org.apache.axis.providers.java.RPCProvider class. This is just another Handler that, when invoked, attempts to call a backend Java object whose class is determined by the "className" parameter specified at deployment time. It uses the SOAP RPC convention for determining the method to call, and makes sure the types of the incoming XML-encoded arguments match the types of the required parameters of the resulting method.

### The Message Path on the Client

The Message Path on the client side is similar to that on the server side, except the order of scoping is reversed, as shown below.
The **service** Handler, if any, is called first - on the client side, there is no "provider" since the service is being provided by a remote node, but there is still the possibility of request and

response Chains. The service request and response Chains perform any service-specific processing of the request message on its way out of the system, and also of the response message on its way back to the caller.

After the service request Chain, the global request Chain, if any, is invoked, followed by the transport. The **Transport Sender**, a special Handler whose job it is to actually perform whatever protocol-specific operations are necessary to get the message to and from the target SOAP server, is invoked to send the message. The response (if any) is placed into the responseMessage field of the MessageContext, and the MessageContext then propagates through the response Chains - first the transport, then the global, and finally the service.

## Subsystems

Axis comprises several subsystems working together with the aim of separating responsibilities cleanly and making Axis modular. Subsystems which are properly layered enable parts of a system to be used without having to use the whole of it (or hack the code).

The following diagram shows the layering of subsystems. The lower layers are independent of the higher layers. The 'stacked' boxes represent mutually independent, although not necessary mutually exclusive, alternatives. For example, the HTTP, SMTP, and JMS transports are independent of each other but may be used together.

In fact, the Axis source code is not as cleanly separated into subsystems as the above diagram might imply. Some subsystems are spread over several packages and some packages overlap more than one subsystem. Proposals to improve the code structure and make it conform more accurately to the notional Axis subsystems will be considered when we get a chance.

## Message Flow Subsystem

### Handlers and Chains

Handlers are invoked in sequence to process messages. At some point in the sequence a Handler may send a request and receive a response or else process a request and produce a response. Such a Handler is known as the *pivot point* of the sequence. As described above, Handlers are either transport-specific, service-specific, or global. The Handlers of each of these three different kinds are combined together into Chains. So the overall sequence of Handlers comprises three Chains: transport, global, and service. The following diagram shows two sequences of handlers: the client-side sequence on the left and the server-side sequence on the right.

A web service does not necessarily send a response message to each request message, although many do. However, response Handlers are still useful in the message path even when there isn't a response message, e.g. to stop timers, clean up resources, etc.

A Chain is a composite Handler, i.e. it aggregates a collection of Handlers as well as implementing the Handler interface as shown in the following UML diagram:

A Chain also has similarities to the Chain of Responsibility design pattern in which a request flows along a sequence of Handlers until it is processed. Although an Axis Chain may process a request in stages over a succession of Handlers, it has the same advantages as Chain of Responsibility: flexibility and the ease with which new function can be added.

Back to message processing -- a message is processed by passing through the appropriate Chains. A message context is used to pass the message and associated environment through the sequence of Handlers. The model is that Axis Chains are constructed offline by having Handlers added to them one at a time. Then they are turned online and message contexts start to flow through the Chains. Multiple message contexts may flow through a single Chain concurrently. Handlers are never added to a Chain once it goes online. If a Handler needs to be added or removed, the Chain must be 'cloned', the modifications made to the clone, and then the clone made online and the old Chain retired when it is no longer in use. Message contexts that were using the old Chain continue to use it until they are finished. This means that Chains do not need to cope with the addition and removal of Handlers while the Chains are processing message contexts -- an important simplification.

The deployment registry has factories for Handlers and Chains. Handlers and Chains can be defined to have 'per-access', 'per-request', or 'singleton' scope although the registry currently only distinguishes between these by constructing non-singleton scope objects when requested and constructing singleton scope objects once and holding on to them for use on subsequent creation requests.

### Targeted Chains

A **Targeted Chain** is a special kind of chain which may have any or all of: a request Handler, a pivot Handler, and a response Handler. The following class diagram shows how Targeted Chains relate to Chains. Note that a Targeted Chain is an aggregation of Handlers by virtue of extending the Chain interface which is an aggregation of Handlers.

A service is a special kind of Targeted Chain in which the pivot Handler is known as a "provider".

### Fault Processing

Now let's consider what happens when a fault occurs. The Handlers prior to the Handler that raised the fault are driven, in reverse order, for onFault (previously misnamed 'undo'). The scope of this backwards scan is interesting: all Handlers previously invoked for the current Message Context are driven.

*Need to explain how "FaultableHandlers" and "WSDD Fault Flows" fit in.*

### Message Contexts

The current structure of a MessageContext is shown below. Each message context may be associated with a request Message and/or a response Message. Each Message has a SOAPPart and an Attachments object, both of which implement the Part interface.
The typing of Message Contexts needs to be carefully considered in relation to the Axis architecture. Since a Message Context appears on the Handler interface, it should not be tied to or biassed in favour of  SOAP. The current implementation is marginally biassed towards SOAP in that the setServiceHandler method narrows the specified Handler to a SOAPService.

### Engine

Axis has an abstract AxisEngine class with two concrete subclasses: AxisClient drives the client side handler chains and AxisServer drives the server side handler chains. The relationships between these classes is fairly simple:

### Engine Configuration

The EngineConfiguration interface is the means of configuring the Handler factories and global options of an engine instance. An instance of a concrete implementation of EngineConfiguration must be passed to the engine when it is created and the engine must be notified if the EngineConfiguration contents are modified. The engine keeps a reference to the EngineConfiguration and then uses it to obtain Handler factories and global options.

The EngineConfiguration interface belongs to the Message Flow subsystem which means that the Message Flow subsystem does not depend on the Administration subsystem.

## Administration Subsystem

The Administration subsystem provides a way of configuring Axis engines. The configuration information an engine needs is a collection of factories for runtime artefacts such as Chains and SOAPServices and a set of global configuration options for the engine.

The Message Flow subsystem's EngineConfiguration interface is implemented by the Administration subsystem. FileProvider enables an engine to be configured statically from a file containing a deployment descriptor which is understood by the WSDDDeployment class. SimpleProvider, on the other hand, enables an engine to be configured dynamically.

### WSDD-Based Administration

WSDD is an XML grammer for deployment descriptors which are used to statically configure Axis engines. Each Handler needs configuration in terms of the concrete class name of a factory for the Handler, a set of options for the handler, and a lifecycle scope value which determines the scope of sharing of instances of the Handler.

The structure of the WSDD grammar is mirrored by a class hierarchy of factories for runtime artefacts. The following diagram shows the classes and the types of runtime artefacts they produce (a dotted arrow means "instantiates").

## Message Model Subsystem

### SOAP Message Model

The XML syntax of a SOAP message is fairly simple. A SOAP message consists of an *envelope* containing:

- an optional *header* containing zero or more *header entries* (sometimes ambiguously referred to as *headers*),
- a *body* containing zero or more *body entries*, and
- zero or more additional, non-standard elements.

The only body entry defined by SOAP is a *SOAP fault* which is used for reporting errors.

Some of the XML elements of a SOAP message define namespaces, each in terms of a URI and a local name, and encoding styles, a standard one of which is defined by SOAP.

Header entries may be tagged with the following optional SOAP attributes:

- *actor* which specifies the intended recipient of the header entry in terms of a URI, and
- *mustUnderstand* which specifies whether or not the intended recipient of the header entry is required to process the header entry.

So the SOAP message model looks like this:

### Message Elements

The classes which represent SOAP messages form a class hierarchy based on the MessageElement class which takes care of namespaces and encodings. The SOAPHeaderElement class looks after the actor and mustUnderstand attributes.

During deserialization, a parse tree is constructed consisting of instances of the above classes in parent-child relationships as shown below.

### Deserialization

The class mainly responsible for XML parsing, i.e. deserialization, is DeserializationContext ('DC'). DC manages the construction of the parse tree and maintains a stack of SAX handlers, a reference to the MessageElement that is currently being deserialized, a stack of namespace mappings, a mapping from IDs to elements, a set of type mappings for deserialization (see Encoding Subsystem) and a SAX event recorder.

Elements that we scan over, or ones for which we don't have a particular deserializer, are recorded - in other words, the SAX events are placed into a queue which may be 'played back' at a later time to any SAX ContentHandler.

Once a SOAPEnvelope has been built, either through a parse or manual construction by the user, it may be output using a SerializationContext (also see Encoding Subsystem). MessageElements all have an output() method which lets them write out their contents.

The SAX handlers form a class hierarchy:

and stack up as shown in the following diagram:

Initially, the SAX handler stack just contains an instance of EnvelopeHandler which represents the fact that parsing of the SOAP envelope has not yet started. The EnvelopeHandler is constructed with a reference to an EnvelopeBuilder, which is the SAX handler responsible for parsing the SOAP envelope.

During parsing, DC receives the events from the SAX parser and notifies either the SAX handler on the top of its handler stack, the SAX event recorder, or both.

On the start of an element, DC calls the SAX handler on the top of its handler stack for onStartChild. This method returns a SAX handler to be used to parse the child, which DC pushes on its SAX handler stack and calls for startElement. startElement, amongst other things, typically creates a new MessageElement of the appropriate class and calls DC for pushNewElement. The latter action creates the parent-child relationships of the parse tree.

On the end of an element, DC pops the top SAX handler from its handler stack and calls it for endElement. It then drives SAX handler which is now on the top of the handler stack for onEndChild. Finally, it sets the MessageElement that is currently being deserialized to the parent of the current one.

Elements which are not defined by SOAP are treated using a SOAPHandler as a SAX event handler and a MessageElement as a node in the parse tree.

### Encoding Subsystem

Encoding is most easily understood from the bottom up. The basic requirement is to transform between values of programming language datatypes and their XML representations. In Axis, this means encoding (or 'serializing') Java objects and primitives into XML and decoding (or 'deserializing') XML into Java objects and primitives. The basic classes that implement these steps are *serializers* and *deserializers*.

Particular serializers and deserializers are written to support a specific XML processing mechanism such as DOM or SAX. So *serializer factories* and *deserializer factories* are introduced to construct serializers and deserializers for a XML processing mechanism which is specified as a parameter.

As is apparent from the above class diagrams, each pair of Java type and XML data type which needs encoding and decoding requires specific serializers and deserializers (actually one of each per XML processing mechanism). So we need to maintain a mapping from a pair of Java type and XML data type, identified by a QName, to a serializer factory and a deserializer factory. Such a mapping is known as a *type mapping*. The type mapping class hierarchy is shown below. Notice how the default type mapping instantiates the various serializer and deserialiser factories.

There is one final level of indirection. How do we know which type mapping to use for a particular message? This is determined by the encoding which is specified in the message. A *type mapping registry* maintains a map from encoding name (URI) to type mapping. Note that the XML data type QNames are defined by the encoding.

So, in summary, to encode a Java object or primitive data value to a XML datatype or to decode the latter to the former, we need to know:

- the Java type we are dealing with,
- the QName of the XML data type we want to encode it as,
- the XML processing mechanism we are using, and
- the encoding name.

### WSDL Tools Subsystem

The WSDL Tools subsystem contains WSDL2Java and Java2WSDL. The Axis runtime does not depend on these tools -- they are just there to make life easier for the user.

#### WSDL2Java

This tool takes a description of a web service written in WSDL and emits Java artefacts used to access the web service.

There are three layers inside the tool:

- framework: SymbolTable, Emitter, WriterFactory
- WSDL2Java plugin to the framework: WSDL2Java (the main), JavaWriterFactory, and all the WSDL-relative writers: JavaPortTypeWriter, JavaBindingWriter, etc.
- The actual WSDL2Java emitters, one for each file generated: JavaInterfaceWriter, JavaStubWriter, etc.

#### Java2WSDL

tbd.

### Interaction Diagrams

#### Client Side Processing

The client side Axis processing constructs a Call object with associated Service, MessageContext, and request Message as shown below before invoking the AxisClient engine.

An instance of  Service and its related AxisClient instance are created before the Call object. The Call object is then created by invoking the Service.createCall *factory method*. Call.setOperation creates a Transport instance, if a suitable one is not already associated with the Call instance. Then Call.invoke creates a MessageContext and associated request Message, drives AxisClient.invoke, and processes the resultant MessageContext. This significant method calls in this sequence are shown in the following interaction diagram.

### Pluggable-Component Discovery

While most pluggable components infrastructures (jaxp/xerces, commons-logging, etc) provide discovery features, it is foreseen that there are situations where these may evolve over time. For example, as leading-edge technologies are reworked and adopted as standards, discovery mechanisms are likely to change.

Therefore, component discovery must be relegated to a **single** point of control within AXIS, typically an AXIS-specific factory method. These factory methods should conform to current standards, when available. As technologies evolve and/or are standardized, the factory methods should be kept up-to-date with appropriate discovery mechanisms.

### Open Issues

1. The relationship between the Axis subsystems needs to be documented and somewhat

cleaned up as there is leakage of responsibilities between some of the subsystems. For example, there is some SOAP and HTTP bias in the basic MessageContext type and associated classes.

2. What classes are included in the "encoding" subsystem? Are the encoding and message model subsystems independent of the other subsystems which depend on "message flow"?
3. (Possibly related to the previous issue) How should we distribute the classes in the above diagram between the Axis subsystems taking into account SOAP-specific and HTTP-specific features?
4. The Axis Engine currently knows about three layers of handlers: transport, global, and service. However, architecturally, this is rather odd. What "law" of web services ensures that there will always and only ever be *three* layers? It would be more natural to use Targeted Chains with their more primitive notion of request, pivot, and response Handlers. We would then implemented the Axis Engine as a Targeted Chain whose pivot Handler is itself a Targeted Chain with global request and response Handlers and a service pivot Handler (which is itself a Targeted Chain as we have just described). Such an Axis Engine architecture is shown in the diagram below.
6. WSDDService.faultFlows is initialised to an empty Vector and there is no way of adding a fault flow to it. Is this dead code or is something else missing?
7. If a fault occurs after the pivot Handler, should the backwards scan notify Handlers which were invoked prior to the pivot Handler? The current implementation does notify such Handlers. However, this is not consistent with the processing of faults raised in a downstream system and stored in the message context by the pivot Handler. These faults are passed through any response Handlers, but do not cause onFault to be driven in the local engine.
8.

We need to consider what's going on here. If you take a sequence of Handlers and then introduce a distribution boundary into the sequence, what effect should that have on the semantics of the sequence in terms of its effects on message contexts? The following diagram shows a client-side Handler sequence invoking a server-side Handler sequence. We need to consider how the semantics of this combined sequence compares with the sequence formed by omitting the transport-related Handlers.

### 1.4.7. Axis Reference Guide

#### 1.4.7.1. Axis Reference Guide

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

## Tools Reference

### WSDL2Java Reference

```
Usage:  java org.apache.axis.wsdl.WSDL2Java [options] WSDL-URI
Options:
    -h, --help
      print this message and exit
    -v, --verbose
      print informational messages
    -n, --noImports
      only generate code for the immediate WSDL document
    -O, --timeout <argument>
      timeout in seconds (default is 45, specify -1 to disable)
    -D, --Debug
      print debug information
   -W, --noWrapped
      turn off support for "wrapped" document/literal
    -s, --server-side
      emit server-side bindings for web service
    -S, --skeletonDeploy <argument>
      deploy skeleton (true) or implementation (false) in deploy.wsdd.
      Default is false.  Assumes --server-side.
    -N, --NStoPkg <argument>=<value>
      mapping of namespace to package
    -f, --fileNStoPkg <argument>
      file of NStoPkg mappings (default NStoPkg.properties)
    -p, --package <argument>
      override all namespace to package mappings, use this package
     name instead
    -o, --output <argument>
      output directory for emitted files
    -d, --deployScope <argument>
      add scope to deploy.xml: "Application", "Request", "Session"
    -t, --testCase
      emit junit testcase class for web service
    -a, --all
```

```
      generate code for all elements, even unreferenced ones
    -T, --typeMappingVersion
      indicate 1.1 or 1.2. The default is 1.1 (SOAP 1.1 JAX-RPC compliant.
      1.2 indicates SOAP 1.1 encoded.)
    -F, --factory <argument>
      name of a custom class that implements GeneratorFactory interface
      (for extending Java generation functions)
    -i, --nsInclude <namespace>
      namescape to specifically include in the generated code (defaults to
      all namespaces unless specifically excluded with the -x option)
    -x, --nsExclude <namespace>
      namespace to specifically exclude from the generated code (defaults to
      none excluded until first namespace included with -i option)
    -p, --property <name>=<value>
      name and value of a property for use by the custom GeneratorFactory
    -H, --helperGen
      emits separate Helper classes for meta data
    -U, --user <argument>
      username to access the WSDL-URI
    -P, --password <argument>
      password to access the WSDL-URI
    -c, --implementationClassName <argument>
      use this as the implementation class
```

**-h, --help**

Print the usage statement and exit

**-v, --verbose**

See what the tool is generating as it is generating it.

**-n, --noImports**

Only generate code for the WSDL document that appears on the command line. The default behaviour is to generate files for all WSDL documents, the immediate one and all imported ones.

**-O, --timeout**

Timeout in seconds. The default is 45. Use -1 to disable the timeout.

**-D, --Debug**

Print debug information, which currently is WSDL2Java's symbol table. Note that this is only printed after the symbol table is complete, ie., after the WSDL is parsed successfully.

**-W, --noWrapped**

This turns off the special treatment of what is called "wrapped" document/literal style operations. By default, WSDL2Java will recognize the following conditions:

- If an input message has is a single part.
- The part is an element.
- The element has the same name as the operation
- The element's complex type has no attributes

When it sees this, WSDL2Java will 'unwrap' the top level element, and treat each of the components of the element as arguments to the operation. This type of WSDL is the default for Microsoft .NET web services, which wrap up RPC style arguments in this top level schema element.

**-s, --server-side**

Emit the server-side bindings for the web service:

- a skeleton class named <bindingName>Skeleton. This may or may not be emitted (see -S, --skeletonDeploy).
- an implementation template class named <bindingName>Impl. Note that, if this class already exists, then it is not emitted.
- deploy.wsdd
- undeploy.wsdd

**-S, --skeletonDeploy <argument>**

Deploy either the skeleton (true) or the implementation (false) in deploy.wsdd. In other words, for "true" the service clause in the deploy.wsdd file will look something like:

```
<service name="AddressBook" provider="java:RPC"> <parameter name="className" value="samples.addr.AddressBookSOAPBindingSkeleton"/> ... </service>
```

and for "false" it would look like:

```
<service name="AddressBook" provider="java:RPC"> <parameter name="className" value="samples.addr.AddressBookSOAPBindingImpl"/> ... </service>
```

The default for this option is false. When you use this option, the --server-side option is assumed, so you don't have to explicitly specify --server-side as well.

**-N, --NStoPkg <argument>=<value>**

By default, package names are generated from the namespace strings in the WSDL document in a magical manner (typically, if the namespace is of the form "http://x.y.com" or "urn:x.y.com" the corresponding package will be "com.y.x"). If this magic is not what you want, you can provide your own mapping using the --NStoPkg argument, which can be repeated as often as necessary, once for each unique namespace mapping. For example, if there is a namespace in the WSDL document called "urn:AddressFetcher2", and you want files generated from the objects within this namespace to reside in the package samples.addr, you would provide the following option to WSDL2Java:

```
--NStoPkg urn:AddressFetcher2=samples.addr
```

(Note that if you use the short option tag, "-N", then there must not be a space between "-N" and the namespace.)

**-f, --fileNStoPkg <argument>**

If there are a number of namespaces in the WSDL document, listing a mapping for them all could become tedious. To help keep the command line terse, WSDL2Java will also look for mappings in a properties file. By default, this file is named "NStoPkg.properties" and it must reside in the default package (ie., no package). But you can explicitly provide your own file using the --fileNStoPkg option.

The entries in this file are of the same form as the arguments to the --NStoPkg command line option. For example, instead of providing the command line option as above, we could provide the same information in NStoPkg.properties:

```
urn\:AddressFetcher2=samples.addr
```

(Note that the colon must be escaped in the properties file.)

If an entry for a given mapping exists both on the command line and in the properties file, the command line entry takes precedence.

**-p, --package <argument>**

This is a shorthand option to map all namespaces in a WSDL document to the same Java package name. This can be useful, but dangerous. You must make sure that you understand the effects of doing this. For instance there may be multiple types with the same name in different namespaces. It is an error to use the --NStoPkg switch and --package at the same time.

**-o, --output <argument>**

The root directory for all emitted files.

**-d, --deployScope <argument>**

Add scope to deploy.wsdd: "Application", "Request", or "Session". If this option does not appear, no scope tag appears in deploy.wsdd, which the Axis runtime defaults to "Request".

**-t, --testCase**

Generate a client-side JUnit test case. This test case can stand on its own, but it doesn't really do anything except pass default values (null for objects, 0 or false for primitive types). Like the generated implementation file, the generated test case file could be considered a template that you may fill in.

**-a, --all**

Generate code for all elements, even unreferenced ones. By default, WSDL2Java only generates code for those elements in the WSDL file that are referenced.

A note about what it means to be referenced. We cannot simply say: start with the services, generate all bindings referenced by the service, generate all portTypes referenced by the referenced bindings, etc. What if we're generating code from a WSDL file that only contains portTypes, messages, and types? If WSDL2Java used service as an anchor, and there's no service in the file, then nothing will be generated. So the anchor is the lowest element that exists in the WSDL file in the order:

1. types
2. portTypes
3. bindings
4. services

For example, if a WSDL file only contained types, then all the listed types would be generated. But if a WSDL file contained types and a portType, then that portType will be generated and only those types that are referenced by that portType.

Note that the anchor is searched for in the WSDL file appearing on the command line, **not** in imported WSDL files. This allows one WSDL file to import constructs defined in another WSDL file without the nuisance of having all the imported WSDL file's constructs generated.

### -T, --typeMappingVersion <argument>

Indicate 1.1 or 1.2. The default is 1.2 (SOAP 1.2 JAX-RPC compliant).

### -F, --factory <argument>

Used to extend the functionality of the WSDL2Java emitter. The argument is the name of a class which extends JavaWriterFactory.

### -H, --helperGen

Emits separate Helper classes for meta data.

### -U, --user <argument>

This username is used in resolving the WSDL-URI provided as the input to WSDL2Java. If the URI contains a username, this will override the command line switch. An example of a URL with a username and password is: `http://user:password@hostname:port/path/to/service?WSDL`

### -P, --password <argument>

This password is used in resolving the WSDL-URI provided as the input to WSDL2Java. If the URI contains a password, this will override the command line switch.

### -c, --implementationClassName <argument>

Set the name of the implementation class.Especially useful when exporting an existing class as a web service using java2wsdl followed by wsdl2java. If you are using the skeleton deploy option you must make sure, after generation, that your implementation class implements the port type name interface generated by wsdl2java. You should also make sure that all your exported methods throws java.lang.RemoteException.

### Java2WSDL Reference

Here is the help message generated from the current tool:

Java2WSDL emitter Usage: java org.apache.axis.wsdl.Java2WSDL [options] class-of-portType Options: -h, --help print this message and exit -I, --input <argument> input WSDL filename -o, --output <argument> output WSDL filename -l, --location <argument> service location url -P, --portTypeName <argument> portType name (obtained from class-of-portType if not specified) -b, --bindingName <argument> binding name (--servicePortName value + "SOAPBinding" if not ~~specified) -S, --serviceElementName <argument> service element name (defaults to~~

--servicePortName value + "Service") -s, --servicePortName <argument> service port name (obtained from --location if not specified) -n, --namespace <argument> target namespace -p, --PkgtoNS <argument>=<value> package=namespace, name value pairs -m, --methods <argument> space or comma separated list of methods to export -a, --all look for allowed methods in inherited class -w, --outputWsdlMode <argument> output WSDL mode: All, Interface, Implementation -L, --locationImport <argument> location of interface wsdl -N, --namespaceImpl <argument> target namespace for implementation wsdl -O, --outputImpl <argument> output Implementation WSDL filename, setting this causes --outputWsdlMode to be ignored -i, --implClass <argument> optional class that contains implementation of methods in class-of-portType. The debug information in the class is used to obtain the method parameter names, which are used to set the WSDL part names. -x, --exclude <argument> space or comma separated list of methods not to export -c, --stopClasses <argument> space or comma separated list of class names which will stop inheritance search if --all switch is given -T, --typeMappingVersion <argument> indicate 1.1 or 1.2. The default is 1.1 (SOAP 1.1 JAX-RPC compliant 1.2 indicates SOAP 1.1 encoded.) -A, --soapAction <argument> value of the operations soapAction field. Values are DEFAULT, OPERATION or NONE. OPERATION forces soapAction to the name of the operation. DEFAULT causes the soapAction to be set according to the operations meta data (usually ""). NONE forces the soapAction to "". The default is DEFAULT. -y, --style <argument> The style of binding in the WSDL, either DOCUMENT, RPC, or WRAPPED. -u, --use <argument> The use of items in the binding, either LITERAL or ENCODED -e, --extraClasses <argument> A space or comma separated list of class names to be added to the type section. -C, --importSchema A file or URL to an XML Schema that should be physically imported into the generated WSDL -X, --classpath additional classpath elements Details: portType element name= <--portTypeName value> OR <class-of-portType name> binding element name= <--bindingName value> OR <--servicePortName value>Soap Binding service element name= <--serviceElementName value> OR <--portTypeName value> Service port element name= <--servicePortName value> address location = <--location value>

**-h , --help**
Prints the help message.

**-I, --input <WSDL file>**
Optional parameter that indicates the name of the input wsdl file. The output wsdl file will contain everything from the input wsdl file plus the new constructs. If a new construct is already present in the input wsdl file, it is not added. This option is useful for constructing a wsdl file with multiple ports, bindings, or portTypes.

**-o, --output <WSDL file>**
Indicates the name of the output WSDL file. If not specified, a suitable default WSDL file is written into the current directory.

**-l, --location <location>**
Indicates the url of the location of the service. The name after the last slash or backslash is the name of the service port (unless overridden by the -s option). The service port address location attribute is assigned the specified value.

**-P, --portTypeName <name>**
Indicates the name to use for the portType element. If not specified, the class-of-portType name is used.

**-b, --bindingName <name>**
Indicates the name to use for the binding element. If not specified, the value of the --servicePortName + "SoapBinding" is used.

**-S, --serviceElementName <name>**
Indicates the name of the service element. If not specified, the service element is the <portTypeName>Service.

**-s, --servicePortName <name>**
Indicates the name of the service port. If not specified, the service port name is derived from the --location value.

**-n, --namespace <target namespace>**
Indicates the name of the target namespace of the WSDL.

**-p, --PkgToNS <package> <namespace>**
Indicates the mapping of a package to a namespace. If a package is encountered that does not have a namespace, the Java2WSDL emitter will generate a suitable namespace name. This option may be specified multiple times.

**-m, --methods <arguments>**
If this option is specified, only the indicated methods in your interface class will be exported into the WSDL file. The methods list must be comma separated. If not specified, all methods declared in the interface class will be exported into the WSDL file.

**-a, --all**
If this option is specified, the Java2WSDL parser will look into extended classes to determine the list of methods to export into the WSDL file.

**-w, --outputWSDLMode <mode>**
Indicates the kind of WSDL to generate. Accepted values are:

- All --- (default) Generates wsdl containing both interface and implementation WSDL constructs.
- Interface --- Generates a WSDL containing the interface constructs (no service element).
- Implementation -- Generates a WSDL containing the implementation. The interface WSDL is imported via the -L option.

**-L, --locationImport <url>**
Used to indicate the location of the interface WSDL when generating an implementation WSDL.

**-N, --namespaceImpl <namespace>**
Namespace of the implementation WSDL.

**-O, --outputImpl <WSDL file>**
Use this option to indicate the name of the output implementation WSDL file. If specified, Java2WSDL will produce interface and implementation WSDL files. If this option is used, the -w option is ignored.

**-i, --implClass <impl-class>**
Sometimes extra information is available in the implementation class file. Use this option to specify the implementation class.

**-x, --exclude <list>**
List of methods to not exclude from the wsdl file.

**-c, --stopClasses <list>**
List of classes which stop the Java2WSDL inheritance search.

**-T, --typeMappingVersion <version>**
Choose the default type mapping registry to use. Either 1.1 or 1.2.

**-A, --soapAction <argument>**
The value of the operations soapAction field. Values are DEFAULT, OPERATION or NONE. OPERATION forces soapAction to the name of the operation. DEFAULT causes the soapAction to be set according to the operation's meta data (usually ""). NONE forces the soapAction to "". The default is DEFAULT.

**-y, --style <argument>**
The style of the WSDL document: RPC, DOCUMENT or WRAPPED. The default is RPC. If RPC is specified, an rpc wsdl is generated. If DOCUMENT is specified, a document wsdl is generated. If WRAPPED is specified, a document/literal wsdl is generated using the wrapped approach. Wrapped style forces the use attribute to be literal.

**-u, --use <argument>**
The use of the WSDL document: LITERAL or ENCODED. If LITERAL is specified, the

XML Schema defines the representation of the XML for the request. If ENCODED is specified, SOAP encoding is specified in the generated WSDL.

**-e, --extraClasses <argument>**
Specify a space or comma seperated list of class names which should be included in the **types** section of the WSDL document. This is useful in the case where your service interface references a base class and you would like your WSDL to contain XML Schema type defintions for these other classes. The --extraClasses option can be specified duplicate times. Each specification results in the additional classes being added to the list.

**-C, --importSchema**
A file or URL to an XML Schema that should be physically imported into the generated WSDL

**-X, --classpath**
Additional classpath elements

## Deployment (WSDD) Reference

Note : all the elements referred to in this section are in the WSDD namespace, namely "http://xml.apache.org/axis/wsdd/".

**<deployment>**
The root element of the deployment document which tells the Axis engine that this is a deployment. A deployment document may represent EITHER a complete engine configuration OR a set of components to deploy into an active engine.
**<GlobalConfiguration>**
This element is used to control the engine-wide configuration of Axis. It may contain several subelements:

- <**parameter**> : This is used to set options on the Axis engine - see the Global Axis Configuration section below for more details. Any number of <**parameter**> elements may appear.
- <**role**> : This is used to set a SOAP actor/role URI which the engine will recognize. This allows SOAP headers targeted at that role to be successfully processed by the engine. Any number of <**role**> elements may appear.
- <**requestFlow**> : This is used to configure global request Handlers, which will be invoked before the actual service on every request. You may put any number of <**handler**> or <**chain**> elements (see below) inside the <**requestFlow**>, but there may only be one <**requestFlow**>.
- <**responseFlow**> : This is used to configure global response Handlers, which will be invoked after the actual service on every request. You may put any number of

**<handler>** or **<chain>** elements (see below) inside the **<responseFlow>**, but there may only be one **<responseFlow>**.

**<undeployment>**

The root element of the deployment document which tells Axis that this is an undeployment.

**<handler [name="*name*"] type="*type*"/>**

Belongs at the top level inside a **<deployment>** or **<undeployment>**, or inside a **<chain>**, **<requestFlow>**, or **<responseFlow>**. Defines a Handler, and indicates the type of the handler. "Type" is either the name of another previously defined Handler, or a QName of the form "**java:class.name**". The optional "name" attribute allows you to refer to this Handler definition in other parts of the deployment. May contain an arbitrary number of **<parameter name="***name***" value="***value***">** elements, each of which will supply an option to the deployed Handler.

**<service name="*name*" provider="*provider*" >**

Deploys/undeploys an Axis Service. This is the most complex WSDD tag, so we're going to spend a little time on it.

**Options** may be specified as follows : **`<parameter name="`**`name`**`"`** **`value="`**`value`**`"/>`**, and common ones include:

- **className** : the backend implementation class
- **allowedMethods** : Each provider can determine which methods are allowed to be exposed as web services.
  To summaries for Axis supplied providers:

  Java RPC Provider (provider="java:RPC") by default all public methods specified by the class in the className option, including any inherited methods are available as web services.
  For more details regarding the Java Provider please see **WHERE???**.

  Java MsgProvder (provider="java:MSG")

  In order to further restrict the above methods, the **allowedMethods** option may be used to specify in a space delimited list the names of only those methods which are allowed as web services. It is also possible to specify for this option the value `"*"` which is functionally equivalent to not specify the option at all. Also, it is worth mentioning that the **operation** element is used to further define the methods being offered, but it does not affect which methods are made available.

  *Note, while this is true for Axis supplied providers, it is implementation dependent on each individual provider. Please review your providers documentation on how or if it supports this option.*

**Note, Exposing any web service has security implications.**As a best practices guide it is highly recommend when offering a web service in un secure environment to restrict allowed methods to only those required for the service being offered. And, for those that are made available, to **fully** understand their function and how they may access and expose your systems's resources.

- **allowedRoles** : comma-separated list of roles allowed to access this service (Note that these are security roles, as opposed to SOAP roles. Security roles control access, SOAP roles control which SOAP headers are processed.)
- **extraClasses** : Specify a space or comma seperated list of class names which should be included in the **types** section of the WSDL document. This is useful in the case where your service interface references a base class and you would like your WSDL to contain XML Schema type defintions for these other classes.

If you wish to define handlers which should be invoked either before or after the service's provider, you may do so with the **<requestFlow>** and the **<responseFlow>** subelements. Either of those elements may be specified inside the **<service>** element, and their semantics are identical to the **<chain>** element described below - in other words, they may contain **<handler>** and **<chain**> elements which will be invoked in the order they are specified.

To control the roles that should be recognized by your service Handlers, you can specify any number of <**role**> elements inside the service declaration.

Example:
<service name="test"> <parameter name="className" value="test.Implementation"/> <parameter name="allowedMethods" value="*"/> <namespace>http://testservice/</namespace> <role>http://testservice/MyRole</role> <requestFlow> <!-- Run these before processing the request --> <handler type="java:MyHandlerClass"/> <handler type="somethingIDefinedPreviously"/> </requestFlow> </service> **Metadata** may be specified about particular operations in your service by using the <operation> tag inside a service. This enables you to map the java parameter names of a method to particular XML names, to specify the parameter modes for your parameters, and to map particular XML names to particular operations.
<operation name="method">
</operation>

**<chain name="*name*"> <subelement/>... </chain>**
Defines a chain. Each *handler* (i.e. deployed handler name) in the list will be invoked() in turn when the chain is invoked. This enables you to build up "modules" of commonly used functionality. The subelements inside chains may be <**handler**>s or <**chain**>s. <handler>s inside a <chain> may either be defined in terms of their Java class:

<chain name="myChain"> <handler type="java:org.apache.axis.handlers.LogHandler"/> </chain> or may refer to previously defined <handlers>, with the "type" of the handler referring to the name of the other handler definition:
<handler name="logger" type="java:org.apache.axis.handlers.LogHandler"/>
<chain name="myChain"/>
<handler type="logger"/>
</chain>
**<transport name="*name*">**
Defines a transport on the server side. Server transports are invoked when an incoming request arrives. A server transport may define **<requestFlow>** and/or **<responseFlow>** elements to specify handlers/chains which should be invoked during the request (i.e. incoming message) or response (i.e. outgoing message) portion of processing (this function works just like the **<service>** element above). Typically handlers in the transport request/response flows implement transport-specific functionality, such as parsing protocol headers, etc.
For any kind of transport (though usually this relates to HTTP transports), users may allow Axis servlets to perform arbitrary actions (by means of a "plug-in") when specific query strings are passed to the servlet (see the section Axis Servlet Query String Plug-ins in the Axis Developer's Guide for more information on what this means and how to create a plug-in). When the name of a query string handler class is known, users can enable it by adding an appropriate **<parameter>** element in the Axis server configuration's **<transport>** element. An example configuration might look like the following:

```
<transport name="http">
<parameter name="useDefaultQueryStrings" value="false" />
<parameter name="qs.name" value="class.name" />
</transport>
```

In this example, the query string that the Axis servlet should respond to is *?name* and the class that it should invoke when this query string is encountered is named `class.name`. The `name` attribute of the **<parameter>** element must start with the string "qs." to indicate that this **<parameter>** element defines a query string handler. The `value` attribute must point to the name of a class implementing the `org.apache.axis.transport.http.QSHandler` interface. By default, Axis provides for three Axis servlet query string handlers (*?list*, *?method*, and *?wsdl*). See the Axis server configuration file for their definitions. If the user wishes not to use these default query string handlers (as in the example), a **<parameter>** element with a `name` attribute equal to "useDefaultQueryStrings" should have its `value` attribute set to `false`. By

default it is set to `true` and the element is not necessary if the user wishes to have this default behavior.
**<transport name="*name*" pivot="*handler type*">**
Defines a transport on the client side, which is invoked when sending a SOAP message. The "pivot" attribute specifies a Handler to be used as the actual sender for this transport (for example, the HTTPSender). Request and response flows may be specified as in server-side transports to do processing on the request (i.e. outgoing message) or response (i.e. incoming message).
**<typeMapping qname="*ns:localName*" classname="*classname*" serializer="*classname*" deserializer="*classname*"/>**
Each typeMapping maps an XML qualified name to/from a Java class, using a specified Serializer and Deserializer.
**<beanMapping qname="*ns:localName*" classname="*classname*">**
A simplified type mapping, which uses pre-defined serializers/deserializers to encode/decode JavaBeans. The class named by "classname" must follow the JavaBean standard pattern of get/set accessors.

**<documentation>**
Can be used inside a **<service>**, an **<operation>** or an operation **<parameter>**. The content of the element is arbitrary text which will be put in the generated wsdl inside a wsdl:document element.
Example:
```
<operation name="echoString" >
  <documentation>This operation echoes a
string</documentation>
  <parameter name="param">
     <documentation>a string</documentation>
  </parameter>
</operation>
```

### Global Axis Configuration

The server is configured (by default) by values in the server-config.wsdd file, though a dedicated Axis user can write their own configuration handler, and so store configuration data in an LDAP server, database, remote web service, etc. Consult the source on details as to how to do that. You can also add options to the web.xml file and have them picked up automatically. We don't encourage that as it is nice to keep configuration stuff in one place.

In the server-config file, there is a global configuration section, which supports parameter name/value pairs as nested elements. Here are the options that we currently document, though there may be more (consult the source, as usual).

```
<globalConfiguration> <parameter name="adminPassword" value="admin"/>
<parameter name="attachments.Directory" value="c:\temp\attachments"/>
<parameter name="sendMultiRefs" value="true"/> <parameter
name="sendXsiTypes" value="true"/> <parameter
name="attachments.implementation"
value="org.apache.axis.attachments.AttachmentsImpl"/> <parameter
name="sendXMLDeclaration" value="true"/> <parameter
name="enable2DArrayEncoding" value="true"/> </globalConfiguration>
```

## Individual Service Configuration

*TODO*

Here are some of the per-service configuration options are available; these can be set in the wsdd file used to deploy a service, from where they will be picked up.

More may exist.

| | |
|---|---|
| **style** | whether to use RPC:enc or doc/lit encoding |
| **SingleSOAPVersion** | When set to either "1.1" or "1.2", this configures a service to only accept the specified SOAP version. Attempts to connect to the service using another version will result in a fault. |
| **wsdlFile** | The path to a WSDL File; can be an absolute path or a resource that axis.jar can load. Useful to export your custom WSDL file. When specify a path to a resource, place a forward slash to start at the beginning of the classpath (e.g "/org/someone/res/mywsdl.wsdl"). How does Axis know whether to return a file or resource? It looks for a file first, if that is missing a resource is returned. |

## Axis Logging Configuration

Axis uses the Jakarta Projects's commons-logging API, as implemented in commons-logging.jar to implement logging throughout the code. Normally this library routes the logging to the Log4j library, provided that an implementation of log4j is on the classpath of the server or client. The commons-logging API can also bind to Avalon, System.out or the Java1.4 logger. The JavaDocs for the library explain the process for selecting a logger, which can be done via a system property or a properties file in the classpath.

Log4J can be configured using the file log4j.properties in the classpath; later versions also support an XML configuration. Axis includes a preconfigured log4j.properties file in axis.jar. While this is adequate for basic use, any complex project will want to modify their own version of the file. Here is what to do

1. Open up axis.jar in a zipfile viewer and remove log4j.properties from the jar
2. Or, when building your own copy of axis.jar, set the Ant property exclude.log4j.configuration to keep the properties file out the JAR.
3. Create your own log4J.properties file, and include it in WEB-INF/classes (server-side), in your main application JAR file client side.
4. Edit this log4J properties file to your hearts content. Server side, setting up rolling logs with fancy html output is convenient, though once you start clustering the back end servers that ceases to be as usuable. Log4J power tools, such as 'chainsaw', are the secret here.

### Log Categories

Axis classes that log information create their own per-class log, each of which may output information at different levels. For example, the main entry point servlet has a log called org.apache.axis.transport.http.AxisServlet, the AxisEngine is org.apache.axis.AxisEngine, and so on. There are also special logs for special categories.

| | |
|---|---|
| org.apache.axis.TIME | A log that records the time to execute incoming messages, splitting up into preamble, invoke, post and send times. These are only logged at debug level. |
| org.apache.axis.EXCEPTIONS | Exceptions that are sent back over the wire. AxisFaults, which are normally created in 'healthy' operation, are logged at debug level. Other Exceptions are logged at the Info level, as they are more indicative of server side trouble. |
| org.apache.axis.enterprise | "Enterprise" level stuff, which generally means stuff that an enterprise product might want to track, but in a simple environment (like the Axis build) would be nothing more than a nuisance. |

## Pre-Configured Axis Components Reference

### On the server:

**SimpleSessionHandler**
uses SOAP headers to do simple session management
**LogHandler**
The LogHandler will simply log a message to a logger when it gets invoked.
**SoapMonitorHandler**
Provides the hook into the message pipeline sending the SOAP request and response messages to the SoapMonitor utility.
**DebugHandler**
Example handler that demonstrates dynamically setting the debug level based on a the value of a soap header element.
**ErrorHandler**
Example handler that throws an AxisFault to stop request/response flow processing.
**EchoHandler**
The EchoHandler copies the request message into the response message.
**HTTPAuth**
The HTTPAuthHandler takes HTTP-specific authentication information (right now, just Basic authentication) and turns it into generic MessageContext properties for username and password
**SimpleAuthenticationHandler**
The SimpleAuthentication handler passes a MessageContext to a SecurityProvider (see org.apache.axis.security) to authenticate the user using whatever information the SecurityProvider wants (right now, just the username and password).
**SimpleAuthorizationHandler**
This handler, typically deployed alongside the SimpleAuthenticationHandler (a chain called "authChecks" is predefined for just this combination), checks to make sure that the currently authenticated user satisfies one of the allowed roles for the target service. Throws a Fault if access is denied.
**MD5AttachHandler**
Undocumented, uncalled, untested handler that generates an MD5 hash of attachment information and adds the value as an attribute in the soap body.
**URLMapper**
The URLMapper, an HTTP-specific handler, usually goes on HTTP transport chains (it is deployed by default). It serves to do service dispatch based on URL - for instance, this is the Handler which allows URLs like http://localhost:8080/axis/services/MyService?wsdl to work.
**RPCProvider**
The RPCProvider is the pivot point for all RPC services. It accepts the following

options:
**className** = the class of the backend object to invoke
**methodName** = a space-separated list of methods which are exported as web services. The special value "*" matches all public methods in the class.
**MsgProvider**
The MsgProvider is the pivot point for all messaging services. It accepts the following options:
**className** = the class of the backend object to invoke
**methodName** = a space-separated list of methods which are exported as web services. The special value "*" matches all public methods in the class.
**JWSHandler**
Performs drop-in deployment magic.
**JAXRPCHandler**
Wrapper around JAX-RPC compliant handlers that exposes an Axis handler interface to the engine.
**LocalResponder**
The LocalResponder is a Handler whose job in life is to serialize the response message coming back from a local invocation into a String. It is by default on the server's local transport response chain, and it ensures that serializing the message into String form happens in the context of the server's type mappings.

**On the client:**

**SimpleSessionHandler**
uses SOAP headers to do simple session management
**JAXRPCHandler**
Wrapper around JAX-RPC compliant handlers that exposes an Axis handler interface to the engine.
**HTTPSender**
A Handler which sends the request message to a remote server via HTTP, and collects the response message.
**LocalSender**
A Handler which sends the request message to a "local" AxisServer, which will process it and return a response message. This is extremely useful for testing, and is by default mapped to the "local:" transport. So, for instance, you can test the AdminClient by doing something like this:
% java org.apache.axis.client.AdminClient -llocal:// list

### 1.4.8. Axis: further reading

### 1.4.8.1. Recommended Reading

Here are things you can read to understand and use Axis better. Remember, you also have access to all the source if you really want to find out how things work (or why they don't).

**Axis installation, use and internals**

1. Tutorial for building J2EE Applications using JBOSS and ECLIPSE
   A good tutorial on open source Enterprise Java Dev, whose chapter nine covers Axis.
2. Web Services with JAX-RPC and Apache Axis.
   by Pankaj Kumar. Starting with a 10000 ft. view of Web Services, prior technologies, current and emerging standards, it quickly gets into the nitty-gritties of using JAX-RPC and Apache Axis for writing and executing programs. Has a nice coverage of different invocation styles - generated stubs, dynamic proxy and dynamic invocation interface. A good place to start if you are new to Web Services and Axis.
   The author also maintains a Web Services Resource Page.
3. Apache Axis SOAP for Java
   Dennis Sosnoski covers Axis. This is another good introductory guide.
4. Enabling SOAPMonitor in Axis 1.0.
   Dennis Sosnoski on how to turn the SOAP monitor on and off, and use it to log your application.
5. Axis in JRun
   Macromedia authored coverage of using Axis from inside JRun.
6. Ask the magic eight ball
   Example of using an Axis service with various caller platforms/languages.
7. Configure Axis Web Services
   Kevin Jones talks a bit about configuring axis, showing how to return handwritten WSDL from the ?wsdl query.
8. Different WSDL Styles in Axis
   Kevin Jones looks at the document and wrapped styles of WSDL2Java bindings.

**Specifications**

1. SOAP Version 1.1
   Remember that SOAP1.1 is not an official W3C standard.
2. SOAP Version 1.2 Part 0: Primer
   This and the follow-on sections cover what the W3C think SOAP is and how it should be used.
3. Web Services Description Language (WSDL) 1.1
4. RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1
   This is HTTP. You really do need to understand the basics of how this works, to work out

why your web service doesn't :)

5. SOAP with Attachments API for Java (SAAJ)
   SAAJ enables developers to produce and consume messages conforming to the SOAP 1.1 specification and SOAP with Attachments note.
6. Java API for XML-Based RPC (JAX-RPC)
   The public API for Web Services in Java. JAX-RPC enables Java technology developers to develop SOAP based interoperable and portable web services. JAX-RPC provides the core API for developing and deploying web services on the Java platform.
7. XML Schema Part 0: Primer
   The W3C XML Schema, (WXS) is one of the two sets of datatype SOAP supports, the other being the SOAP Section 5 datatypes that predate WXS. Complicated as it is, it is useful to have a vague understanding of this specification.
8. Java API for XML Messaging (JAXM)
   JAXM enables applications to send and receive document oriented XML messages using a pure Java API. JAXM implements Simple Object Access Protocol (SOAP) 1.1 with Attachments messaging so that developers can focus on building, sending, receiving, and decomposing messages for their applications instead of programming low level XML communications routines.

**Explanations, articles and presentations**

1. A Gentle Introduction to SOAP
   Sam Ruby tries not to scare people.
2. A Busy Developer's Guide to WSDL 1.1
   Quick intro to WSDL by the eponymous Sam Ruby.
3. Axis - an open source web service toolkit for Java
   by Mark Volkmann, Partner, Object Computing, Inc. A very good introduction to SOAP and Axis. Highly Recommended.
4. When Web Services Go Bad
   Steve Loughran tries to scare people. A painful demonstration how deployment and system management are trouble spots in a production service, followed by an espousal of a deployment-centric development process. Remember, it doesn't have to be that bad.
5. JavaOne 2002, Web Services Today and Tomorrow
   (Java Developer connection login required)
6. The Java Web Services Tutorial: Java API for XML-based RPC
   This is part of Sun's guide to their Java Web Services Developer Pack. The examples are all based on their JWSDP, but as Axis also implements JAX-RPC, they may all port to Axis.
7. Using Web Services Effectively.
   Blissfully ignoring issues such as versioning, robustness and security and all the other details a production Web Service needs, instead pushing EJB as the only way to process

requests, this is Sun's guide to using web services in Java. It also assumes Java is at both ends, so manages to skirt round the interop problem.

8. Making Web Services that Work
   A practical but suspiciously code free paper on how to get web services into production. As well as coverage of topics such as interop, versioning, security, this (57 page) paper looks at the deployment problem, advocating a fully automated deployment process in which configuration problems are treated as defects for which automated test cases and regresssion testing are appropriate. Happyaxis.jsp is the canonical example of this. The author, Steve Loughran also looks a bit at what the component model of a federated web service world would really be like.

**Interoperability**

1. To infinity and beyond - the quest for SOAP interoperability
   Sam Ruby explains why Interop matters so much.
2. The Wondrous Curse of Interoperability
   Steve Loughran on interop challenges (especially between .NET and Axis), and how to test for them.

**Advanced topics**

1. Requirements for and Evaluation of RMI Protocols for Scientific Computing
2. Architectural Styles and the Design of Network-based Software Architectures
   The theoretical basis of the REST architecture
3. Investigating the Limits of SOAP Performance for Scientific Computing
4. Architectural Principles of the World Wide Web
   The W3C architects say how things should be done.

**Books**

1. *Beginning Java Web Services*
   Meeraj Kunnumpurath et al, Wrox Press, September 2002.
   An introductory book, with the early chapters focusing on Axis.
   The sample chapter shows how to install Axis with Tomcat 4.0: we do not believe that their approach is the best. It is easier to drop jaxrpc.jar and saaj.jar into the CATALINA_HOME/common/lib dir than it is to add all axis jars to the classpath by hand. The book is based on Axis Beta-3.
2. *Java development with Ant*
   by Erik Hatcher and Steve Loughran, Manning Press, July 2002.
   A book on Ant development which covers Web Service development with Axis, along with other topics relevant to Java developers using Ant. The Web Service chapter, chapter 15, is free to download, and was the birthplace of happyaxis.jar.
   The book is based on Axis Beta-2; the web site contains updated documentation where

appropriate.
3. *AXIS: Next Generation Java SOAP*
by Romin Irani and S Jeelani Bashna, Wrox Press, May 2002.
The first nothing-but-Axis book.
It is based on Beta-1. This is a reasonable book, despite is apparent thinness and relative age. If it has a major weakness it believes everything works as intended, which regular Axis users will know is not quite true yet. Maybe they didn't want to fault missing features and other gotchas, assuming they would be fixed by the time the product shipped, but the effective result is that you can get into minor trouble working from this book, trying to use bits that aren't there, or just don't work (yet).
4. *Building Web Services with Java: Making Sense of XML, SOAP, WSDL and UDDI*
Steve Graham et al, December 2001.
Covering very early versions of Axis along with other aspects of Web Service technologies. One of the authors, Glen Daniels, is an Axis committer and active contributor, so the quality of the Axis coverage is high. Good explanations of SOAP, UDDI, and the like.

Authors, publishers: we welcome additions to this section of any books which have some explicit coverage of Axis. Free paper/pdf copies and other forms of bribery accepted.

**External Sites covering Web Services**

1. IBM developerWorks Web Services corner
There are lots of interesting articles on Web Services here, many of which are Axis related. There is also a listing of "all current open standards and specifications that define the Web services family of protocols", though Soap with Attachments is mysteriously absent.

### 1.4.9. Axis requirements & status

#### 1.4.9.1. Requirements

There is a non-requirements section below.
Release cycles are explained below.

#### 1.4.9.2. Non-requirements (won't be supported)

*We find the SOAP spec. to be unclear on these issues so we decided not to support them.*

1. RPC calls in SOAP headers
2. Multiple RPC calls in a single SOAP message

#### 1.4.9.3. Releases and test cycles

We're planning on releasing alpha1 (a1), alpha2 (a2), beta, and 3.0.
alpha is a preview.
subsequent alphas are to show the growing set of features and docs and test cases and all that.
Beta is functionally complete.

## 1.5. Axis (C++)

### 1.5.1. Axis C++ 1.5 Final is Available!

#### 1.5.1.1. Axis C++ 1.5 Final

Download Axis C++

#### 1.5.1.2. Key features of Axis C++ 1.5 Final

**New client side transport**
This is called Axis3 transport.This is a cleaner and extensible Transport compared to Axis2Transport.
WSDL tool fixes to handle nil types.
Supports Broader XSD types.
Supports Broader XSD Any types
Comprehensive test framework which includes client & server side.

#### 1.5.1.3. The Bug Fixes

AXISCPP-605
AXISCPP-602
AXISCPP-600
AXISCPP-596
AXISCPP-594
AXISCPP-593
AXISCPP-592
AXISCPP-591
AXISCPP-590
AXISCPP-589
AXISCPP-587
AXISCPP-586
AXISCPP-585

AXISCPP-584
AXISCPP-583
AXISCPP-582
AXISCPP-581
AXISCPP-579
AXISCPP-576
AXISCPP-575
AXISCPP-569
AXISCPP-568
AXISCPP-567
AXISCPP-566
AXISCPP-564
AXISCPP-562
AXISCPP-561
AXISCPP-560
AXISCPP-556
AXISCPP-555
AXISCPP-553
AXISCPP-550
AXISCPP-549
AXISCPP-545
AXISCPP-534
AXISCPP-532
AXISCPP-531
AXISCPP-530
AXISCPP-528
AXISCPP-525
AXISCPP-524
AXISCPP-523
AXISCPP-521
AXISCPP-520
AXISCPP-518
AXISCPP-517
AXISCPP-516
AXISCPP-514
AXISCPP-513
AXISCPP-511
AXISCPP-510
AXISCPP-509
AXISCPP-508

AXISCPP-507
AXISCPP-504
AXISCPP-503
AXISCPP-502
AXISCPP-501
AXISCPP-500
AXISCPP-499
AXISCPP-498
AXISCPP-497
AXISCPP-496
AXISCPP-495
AXISCPP-494
AXISCPP-493
AXISCPP-492
AXISCPP-491
AXISCPP-490
AXISCPP-489
AXISCPP-488
AXISCPP-487
AXISCPP-486
AXISCPP-485
AXISCPP-484
AXISCPP-483
AXISCPP-482
AXISCPP-481
AXISCPP-480
AXISCPP-479
AXISCPP-478
AXISCPP-477
AXISCPP-475
AXISCPP-474
AXISCPP-472
AXISCPP-470
AXISCPP-469
AXISCPP-468
AXISCPP-466
AXISCPP-465
AXISCPP-464
AXISCPP-463
AXISCPP-462

AXISCPP-459
AXISCPP-457
AXISCPP-456
AXISCPP-454
AXISCPP-453
AXISCPP-451
AXISCPP-450
AXISCPP-447
AXISCPP-446
AXISCPP-444
AXISCPP-443
AXISCPP-442
AXISCPP-441
AXISCPP-437
AXISCPP-436
AXISCPP-433
AXISCPP-431
AXISCPP-430
AXISCPP-428
AXISCPP-425
AXISCPP-420
AXISCPP-419
AXISCPP-418
AXISCPP-417
AXISCPP-415
AXISCPP-414
AXISCPP-413
AXISCPP-410
AXISCPP-409
AXISCPP-408
AXISCPP-407
AXISCPP-403
AXISCPP-400
AXISCPP-398
AXISCPP-392
AXISCPP-390
AXISCPP-389
AXISCPP-385
AXISCPP-383
AXISCPP-376

AXISCPP-375
AXISCPP-371
AXISCPP-364
AXISCPP-355
AXISCPP-348
AXISCPP-346
AXISCPP-344
AXISCPP-341
AXISCPP-340
AXISCPP-335
AXISCPP-331
AXISCPP-310
AXISCPP-306
AXISCPP-305
AXISCPP-303
AXISCPP-300
AXISCPP-293
AXISCPP-288
AXISCPP-270
AXISCPP-268
AXISCPP-242
AXISCPP-216
AXISCPP-207
AXISCPP-164

#### 1.5.1.4. Known Issues

GNU make based build system is not working.
Out of the two parsers Expat and Xerces, only Xerces is supported.
C support is not complete.
There are no vc projects for samples
Pending bugs in Jira.

We hope you will enjoy using Axis C++.
Numerous efforts are currently underway to improve Axis C++ as a whole. Please have a look at the TODO page to learn about 1.5 plans for Axis C++.
We value your feed back very much.

Please report any bugs in Jira and feel free to let us know your thoughts and/or problems in axis-c-user@ws.apache.org
We welcome you to contribute to Axis C++ and please join the discussions in

axis-c-dev@ws.apache.org

### 1.5.2. Axis C++ Documentation

#### 1.5.2.1. Documentation

Apache Axis C++ 1.5 Final

**Installation Guides**

- Linux Installation Instructions
- Windows Installation Instructions

**User Guides**

- Linux User Guide
- Windows User Guide

**Developer Guides**

- Windows Developer's Guide
- ANT Build Guide

**Reference Material**

- Handler Tutorial
- Architecture Guide
- WSDL2Ws Tool
- Memory Management Guide

### 1.5.3. Axis C++ download page

#### 1.5.3.1. Download Axis C++

Direct Link

(Direct link to a Axis C++ distribution folder)

Mirror Sites

(Click on the address of a mirror. Once you are presented with the contents of the dist folder, click on the "ws" folder.
Then click on "axis-c" to find the distribution)

### 1.5.4. The Axis C++ team

### 1.5.4.1. The Axis C++ team

**Active Contributors**

Susantha Kumara <susantha@virtusa.com, susantha@opensource.lk>,

Damitha Kumarage <damitha@beyondm.net, damitha@opensource.lk>,

Roshan Weerasuriya <roshan@jkcsworld.com, roshan@opensource.lk>,

Sanjaya Singharage <sanjayas@jkcsworld.com,sanjayas@opensource.lk>,

John Hawkins <HAWKINSJ@uk.ibm.com>,

Samisa Abeysinghe <samisa_abeysinghe@yahoo.com>,

Fred Preston <PRESTONF@uk.ibm.com>,

Mark Whitlock <mark_whitlock@uk.ibm.com>,

Andrew Perry <PERRYAN@uk.ibm.com>,

Adrian Dick <adrian.dick@uk.ibm.com>,

Sanjiva Weerawarana <sanjiva@opensource.lk>,

Farhaan Mohideen <farhaan@opensource.lk>,

Nithyakala Thangarajah <nithya@opensource.lk>,

Rangika Mendis <rangika@opensource.lk>,

Sharanka Perera <sharanka@opensource.lk>,

M.F.Rinzad Ahamed <rinzad@opensource.lk>,

**Additional Contributors**

Chaminda Divitotawela <cdivitotawela@virtusa.com, chadiv@opensource.lk>,

Nuwan Gurusinghe <nuwan@beyondm.net, nuwan@opensource.lk>,

Chamindra de Silva <chamindra@virtusa.com>,

Kanchana Welagedara <kanchana@opensource.lk>,

Srinath Perera <hemapani@cse.mrt.ac.lk, hemapani@opensource.lk>, Thushantha Ravipriya De Alwis <thushantha@beyondm.net, ravi@opensource.lk>,

Dimuthu Leelarathne <muthulee@cse.mrt.ac.lk, muthulee@opensource.lk>,

Jeyakumaran.C <jkumaran@opensource.lk>,

Vairamuthu Thayapavan <vtpavan@opensource.lk>,

Satheesh Thurairajah

Piranavam ThiruChelvan <chelvan@opensource.lk>,

Dharmarajeswaran Dharmeehan <dhar@opensource.lk>,

Selvarajah Selvendra <selva@opensource.lk>,

Lilantha Darshana <Lilantha@virtusa.com>,

Nadika Ranasinghe <nranasinghe@virtusa.com, nadika@opensource.lk>,

## 1.6. Downloads

### 1.6.1. WebServices - Axis

#### 1.6.1.1. WebServices - Axis - Releases

| Name | Date | Description |
|---|---|---|
| 1.2RC2 | November 17, 2004 | Release Candidate #2 for version 1.2. |
| 1.2RC1 | September 30, 2004 | Release Candidate #1 for version 1.2. |
| 1.2beta3 | August 17, 2004 | Third beta release for version 1.2. |
| 1.2beta2 | July 14, 2004 | Second beta release for version 1.2. |
| 1.2beta1 | April 1, 2004 | First beta release for version 1.2. |
| 1.2alpha | December 1, 2003 | Alpha Version 1.2. |
| 1.1 (from mirror) | June 16, 2003 | Final Version 1.1. |
| 1.1rc2 | March 5, 2003 | Release Candidate #2 for |

| | | |
|---|---|---|
| | | version 1.1. |
| 1.1rc1 | February 9, 2003 | Release Candidate #1 for version 1.1. |
| 1.1beta | December 3, 2002 | Beta for 1.1 release |
| 1.0 | October 7, 2002 | Release 1.0. |
| 1.0rc2 | September 30, 2002 | Release Candidate #2 for version 1.0. |
| 1.0rc1 | September 6, 2002 | Release Candidate #1 for version 1.0. |
| Beta 3 | July 9, 2002 | Third beta release (changes since beta 2). |
| Beta 2 | April 29, 2002 | Second beta release (changes since beta 1) |
| Beta 1 | March 15, 2002 | First beta release. |
| Alpha 3 | December 14, 2001 | Third Alpha - add JAX RPC, WSDD, more WSDL functionallity, etc. |
| Alpha 2 | September 21, 2001 | Second Alpha - add WSDL functionality, many bug fixes |
| Alpha 1 | August 15, 2001 | First Alpha release |

For nightly builds, see the Interim Drops page.

### 1.6.2. WebServices - Axis

#### 1.6.2.1. WebServices - Axis - Interim

Nightly builds are done of the current source in the CVS repository. The source and binaries from these builds are available at:
http://cvs.apache.org/dist/axis/nightly

Nightly Snapshots of the current CVS source tree are available at:
http://cvs.apache.org/snapshots/ws-axis/

## 1.7. Translation

## 1.8. Related Projects

## 1.9. Misc

### 1.9.1. WebServices - Axis

#### 1.9.1.1. WebServices - Axis - Who We Are

The Axis Project operates on a meritocracy: the more you do, the more responsibility you will obtain. This page lists all of the people who have gone the extra mile and are Committers. If you would like to get involved, the first step is to join the mailing lists.

We ask that you please do not send us emails privately asking for support. We are non-paid volunteers who help out with the project and we do not necessarily have the time or energy to help people on an individual basis. Instead, we have setup mailing lists which often contain hundreds of individuals who will help answer detailed requests for help. The benefit of using mailing lists over private communication is that it is a shared resource where others can also learn from common mistakes and as a community we all grow together.

**Active Committers (Java)**

- Andras Avar <andras.avar@nokia.com>
- David Chappell <chappell@sonicsoftware.com>
- Glen Daniels <gdaniels@apache.org>
- Doug Davis <dug@apache.org>
- Eric Friedman <ericf@apache.org>
- Chris Haddad <haddadc@apache.org>
- Tom Jordahl <tomj@macromedia.com>
- Dominik Kacprzak <dominik@apache.org>
- Rick Kellogg <rmkellogg@comcast.net >
- Toshiyuki Kimura (Toshi) <kimuratsy@nttdata.co.jp>
- Steve Loughran
- Jaime Meritt <jmeritt@sonicsoftware.com>
- Yuhichi Nakamura <nakamury@apache.org>
- Thomas Sandholm <sandholm@mcs.anl.gov>
- Igor Sedukhin <igors@apache.org>
- Davanum Srinivas <dims@yahoo.com>
- Sanjiva Weerawarana <sanjiva@watson.ibm.com>
- Changshin Lee (a.k.a. Ias) <iasandcb@tmax.co.kr>
- Srinath Perera <hemapani@opensource.lk>

- Venkat Reddy <venkat@apache.org>
- Jarek Gawor <gawor@apache.org>
- Jongjin Choi <jjchoe@tmax.co.kr>

**Active Committers (C++)**

- Chaminda Divitotawela <chadiv@opensource.lk>
- Nuwan Gurusinghe <nuwan@opensouce.lk>
- Susantha Kumara <susantha@opensource.lk>
- Damitha Kumarage <damitha@opensource.lk>
- Nadika Ranasinghe <nadika@opensource.lk>
- Sanjaya Sinharage <sanjayasing@opensource.lk>
- Roshan Weerasuriya <roshan@opensource.lk>
- Sanjiva Weerawarana <sanjiva@watson.ibm.com>

**Committers Emeriti (committers that have been inactive for 3 months or more)**

- Vahe Amirbekyan <avahe@apache.org>
- Russell Butek <butek@us.ibm.com>
- Wouter Cloetens <wouter@mind.be>
- Matt Duftler <duftler@apache.org>
- Steve Graham <sggraham@us.ibm.com>
- Rob Jellinghaus <robj@helium.com>
- Jacek Kopecky <jacek@idoox.com>
- Ravi Kumar <rkumar@borland.com>
- Berin Loritsch <bloritsch@apache.org>
- George Matkovits <matkovitsg@apache.org>
- Kevin Mitchell <kmitchell@apache.org>
- Vidyanand Murunikkara <vidyanand@infravio.com>
- Bill Nagy <wnagy@us.ibm.com>
- Christopher Nelson <cnelson@synchrony.net>
- Ryo Neyama <neyama@apache.org>
- Glyn Normington <glyn@apache.org> [Apache home page]
- Rick Rineholt <rineholt@us.ibm.com >
- Sam Ruby <rubys@us.ibm.com>
- Rich Scheuerle <scheu@us.ibm.com>
- Matt Seibert <mseibert@us.ibm.com>
- Richard Sitze <rsitze@apache.org>
- James Snell <jasnell@us.ibm.com>

### 1.9.2. WebServices - Axis

#### 1.9.2.1. WebServices - Axis - Contact Us

If you have questions or comments about this site, please send email to: axis-dev@ws.apache.org.

If you have questions or comments about the software or documentations on this site, please subscribe to the axis-user mailing list:

Mailing lists

The Axis project is an effort of the Apache Software Foundation. The address for general ASF correspondence and licensing questions is:

apache@apache.org

You can find more contact information for the Apache Software Foundation on the contact page of the main Apache site.

### 1.9.3. WebServices - Axis

#### 1.9.3.1. WebServices - Axis - Legal Stuff

All material on this website is Copyright © 1999-2003, The Apache Software Foundation.

Sun, Sun Microsystems, Solaris, Java, JavaServer Web Development Kit, and JavaServer Pages are trademarks or registered trademarks of Sun Microsystems, Inc. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. Windows, WindowsNT, and Win32 are registered trademarks of Microsoft Corp. All other product names mentioned herein and throughout the entire web site are trademarks of their respective owners.

### 1.9.4. WebServices - Axis

#### 1.9.4.1. WebServices - Axis - Misc Notes

Misc notes and docs that might be of interest...

- Current list of requirements
- Notes from the 1st Face-2-Face
- Notes from the 2nd Face-2-Face
- Notes from the Interop meeting with Microsoft
- Glen's note about SOAPVerse
- Toshi's note about Caching Mechanism

- SOAPMonitor User's Guide [for nightly build]
- Axis site in Japanese [translation]