WebServices - Axis

1. Axis Developer's Guide

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

1.1. Table of Contents

1.2. Introduction

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

1.3. 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.

1.4. 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"
```

1.5. Pluggable-Components

The Axis Architecture Guide explains the requirements for pluggable components.

1.5.1. 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.

1.5.2. 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.

1.5.2.1. 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).

1.5.2.2. Guidelines

1.5.2.3. 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...

1.5.2.4. 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).

1.5.2.5. 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).

1.5.3. 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.

1.6. 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.

1.7. 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.

1.8. 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.

1.9. 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.

1.9.1. 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).
2. 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:

   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.
3. 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:

   Messages.getMessage("sample00", "Rich Scheuerle", "Software Developer");

4. All keys in the properties file should use the syntax <string><2-digit-suffix>.
   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.

1.9.1.1. 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"));
```

1.9.2. 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
   `Messages.getMessage("myMsg00");`
   than
   `Messages.getResourceBundle().getString("myMsg00");`
2. The `getMessage` methods enable messages with variables.

1.9.2.1. 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.

1.9.3. 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.

1.10. 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.

1.10.1. 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.
2. Created a `SequenceTest.wsdl` file defining the webservice.
3. Ran the Wsdl2java emitter to create Java files:

   ```
   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.
4. 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:

   ```
   <!-- 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>
   ```
5. 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.

6. Done. Run `ant functional-tests` to verify. Check in your test.

1.11. 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

1.12. 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.

1.13. 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.

1.14. 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.

1.15. 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.

1.16. 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
```

1.17. Debugging

1.17.1. 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.

1.17.2. 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.

1.18. 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!