WebServices - Axis

1. Axis System Integration Guide

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

1.1. Table of Contents

- Introduction
- Pluggable APIs
  - Components
  - Logging/Tracing
  - Configuration
  - Handlers
  - Internationalization
  - Performance Monitoring
  - Encoding
  - WSDL Parser and Code Generator Framework
- Client SSL

1.2. 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.

1.3. 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.

1.3.1. Components

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

1.3.1.1. 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).

1.3.1.2. 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.

1.3.1.3. 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.

1.3.1.4. 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

1.3.2. 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.

1.3.2.1. 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.

1.3.2.2. 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.

1.3.2.3. 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.

1.3.2.4. 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

1.3.3. 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.

1.3.4. Handlers

See the Architecture Guide for current information on Handlers.

1.3.5. 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.

1.3.5.1. 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.

1.3.5.2. 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).

1.3.6. Performance Monitoring

Axis does not yet include specific Performance Monitoring Plugs.

1.3.7. Encoding

Axis does not yet include an Encoding Plug.

1.3.8. 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)

1.3.8.1. 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);

1.3.8.2. 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: WSDL2Java, WSDL2Lisp, WSDL2XXX.

```
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.

1.3.8.3. 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...

1.3.8.4. WSDL Framework Extension Examples

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

1.3.8.5. 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);
  }
```

```
}
```

1.3.8.6. 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() {
    // WSDL2Java's JavaDefinitionWriter
    addGenerator(Definition.class, JavaDefinitionWriter.class);

    // our DeployWriter
    addGenerator(Definition.class, MyDeployWriter.class);

    // WSDL2Java's JavaUndeployWriter
    addGenerator(Definition.class, JavaUndeployWriter.class);
  }
}
```

- 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.

1.4. 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.