Wiseman

Software Version: 1.0

Server Developer's Guide

Document Release Date: June 2007

Software Release Date: June 2007

Copyright Notices

Documentation Updates

Table 1 Document Changes

Contributor	Date	Contact	Change Log
Hewlett-Packard	06/2007	dev@wiseman.dev.java.net	Initial creation of the documentation. Primary Author: Joseph Ruzzi Contributing Authors: Simeon Pinder, Nancy Beers, Denis Rachal, and William Reichardt.

Contents

1 Introduction

This chapter provides overview information about the WS-Management specification and the Wiseman software.

The following sections are included in this chapter:

Intended Audience

Wiseman is primarily intended for:

- developers that want to manage resources using WS-Management-compliant Web services
- developers that want to consume WS-Management-compliant Web services
- system architects that want to create management solutions using WS-Management as implemented by Wiseman

Users are expected to have fundamental knowledge of: Java, XSD, XML, XPath, Web services, and JAXB.

WS-Management Overview

The Web Services for Management (WS-Management) specification provides a well-defined protocol for managing resources using Web services. The WS-Management specification draws on many specifications and provides best practices and usage requirements that are specific to managing resources and central to any systems management solution.

Solutions that adhere to the WS-Management specification provide management clients with the ability to:

- Discover the presence of management resources and navigate between them
- Get, put, create, and delete individual management resources, such as settings and dynamic values
- Enumerate the contents of containers and collections, such as large tables and logs
- Subscribe to events emitted by managed resources
- Execute specific management methods with strongly typed input and output parameters

The WS-Management specification is defined through the Distributed Management Task Force (DMTF) community with participation from industry-leading companies. The specification can be freely downloaded from *http://www.dmtf.org/standards/published_documents/DSP0226.pdf*

The following table describes each of the specifications that are utilized within the WS-Management specification.

Table 2

Specification	Description
WS-Addressing, 2004/08	This specification defines a construct to identify/reference a Web service endpoint for access and for addressing messages between endpoints, and it defines a construct of message information headers allowing uniform addressing of messages independent of underlying transport.
WS-Transfer, 2004/09	This specification defines operations using Web services infrastructure for sending and receiving XML representations of a resource and for creating and deleting a resource and its corresponding representation.
WS-Enumeration, 2004/09	This specification defines a simple SOAP-based protocol for the transfer of large data sets. Enumeration of the data by the consumer is facilitated through a session abstraction, allowing the consumer to request XML element information items from the data source over the span of one or more SOAP messages.
WS-Eventing, 2004/08	This specification defines a protocol for a Web service to register interest with another Web service in order to receive messages about events. The subscriber may manage the subscription by interacting with a Web service designated by the event source.

Wiseman Overview

Wiseman is an open source implementation of the WS-Management specification for the Java SE platform. Wiseman implements the WS-Management specification and its dependent specifications.

Wiseman provides design-time tools and also a WS runtime environment for hosting Web services created using Wiseman. Wiseman leverages many open source libraries, such as JAXWS, JAXB, SAAJ, and Velocity.

Wiseman Tools

Wiseman provides a set of tools that facilitate developing WS-Management-compliant Web services. They are provided to help developers focus on defining their resources without having to deal with low-level implementation details such as building WSDLs or serializing objects to and from XML.

These tools are not required to create WS-Management-compliant Web services but are instead provided to save time.

The tools include:

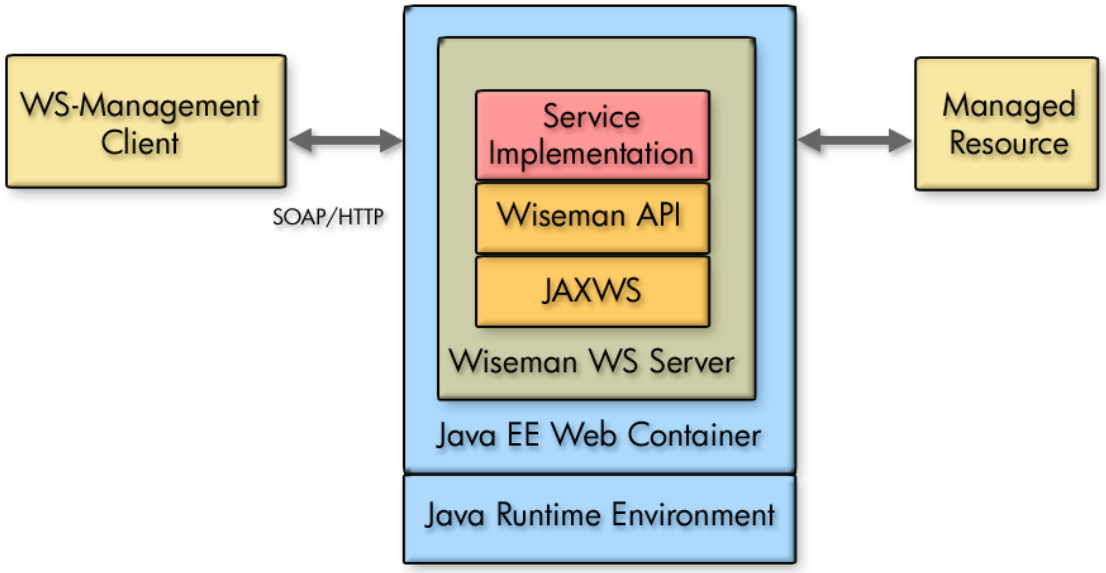
- A tool for generating WS-Management compliant WSDL based on a resource defined using an XML Schema.
- A tool for generating source Java classes from a WSDL

- A metadata viewer for discovering how to consume a WS-Management Web service
- A project template for quickly starting projects.
- Client utility classes to help consume WS-Management Web services
- Command-line utility for testing WS-Management Web services
- Sample applications that demonstrate many Wiseman features.

Conceptual Architecture

Wiseman provides a complete Web services runtime environment that is built using open standards. The runtime environment and service implementations are packaged together as a Web application and can be deployed to any Java EE Web container.

Figure 1 Conceptual Architecture



Wiseman does not use existing Web services environments because the WS-Management specification dictates that XML documents be dispatched (or mapped to) Java classes using WS-Addressing information and not off the first element in the SOAP document body as is common in the WS-I Basic Profile for Web services. In many cases, WS-Management documents may have no document body at all which is not permitted by WS-I. Because offerings such as Axis and JWSDP do not yet support addressing based dispatching of XML documents, these environments cannot be used.

Use Cases

Wiseman and WS-Management do not define or constrain what constitutes a resource. From a Wiseman perspective, as long as the resource provides an API interface that is implemented in Java, it can conceivably be exposed for manageability using Wiseman. Use cases for Wiseman are predominately Java-based. However, from a management client perspective, there are no domain or platform constraints, since the WS-Management protocol interoperates across platforms. This section provides several common use cases that demonstrate the use of Wiseman and WS-Management.

Network Management

Perhaps the most obvious Wiseman use cases involve network/systems management. Many management solutions in this space have already transitioned to distributed management solutions. Therefore, the use of Web services is a natural progression for distributed management. Some possible use cases include:

- Discovering network nodes in an enterprise
- Starting and Stopping nodes remotely
- Aggregating event messages
- Discovering configuration settings
- Pushing configuration updates
- Managing disparate platforms from a single console

Application Management

Any Java application can be managed using WS-Management Web services. One of the benefits of application management using WS-Management is that any WS-Management-based client, whether implemented in Java or not, can be used to manage a Java application. The most common management use cases include retrieving an application's performance statistics or changing an application's configuration.

Moreover, JSR 262 is working to define a Web services connector to access remote JMX MBean Servers. The Web services protocol used by JSR 262 is WS-Management. This would allow any type of client to interact with JMX MBeans using Web services. It is expected that the specification will become the standard for Java management using Web services.

Standalone Resource Management

Lastly, an enterprise contains many well-defined, standalone resources that can be exposed using Wiseman. For example, an LDAP repository can use WS-Management-based Web services to expose functionality for updating user information.

2 Getting Started

This chapter includes Wiseman installation directions as well as a quick demonstration to verify the installation and exercise some of the features that are included in the release. Complete these sections before starting any development work.

The following sections are included in this chapter:

Prerequisites

Wiseman depends on the following software:

- JDK 1.5.x or higher
- Apache Ant 1.6.5 or higher – Install Apache Ant and include its `\bin` directory in the `PATH` environment variable. Set an `ANT_HOME` environment variable to the Ant installation directory.
- Apache Tomact 5.5.17 or higher – Install Apache Tomcat. This documentation refers to the Tomcat home directory (e.g., `c:\jakarta-tomcat-5.5.17`) as `TOMCAT_HOME`. In addition, it is assumed that Tomcat is configured to listen at the default HTTP port (`8080`).

Installing Wiseman

Wiseman is released under the Apache License, Version 2.0. You must agree to the licenses as part of the installation.

To install Wiseman:

1. Download the Wiseman *binary distribution* to a directory on your computer. This directory is referred to as `INSTALL_DIR`.
2. From a command prompt, change directories to `INSTALL_DIR` and issue the following command:

   `java -jar WISEMAN_<version>.jar`

   The Apache License Agreement screen displays.
3. Scroll to read the Apache license agreement and click **ACCEPT** to accept the terms of the agreement.

4 From the Select install directory dialog box, select an installation directory. A directory called `wiseman` will be created in the installation directory. The `wiseman` directory will be referred to as `WISEMAN_HOME` throughout the documentation.

5 Click **INSTALL**. The Installing… dialog box displays and shows the status of the installation.

6 After the installation completes, click **OK** on the Install Completed dialog box.

Building the Samples

Wiseman includes two samples that demonstrate some of the features and capabilities of Wiseman. The samples are also used as a way of learning how to create and consume a resource's WS-Management Web services. The samples are located in the `WISEMAN_HOME\samples` directory. This section provides instructions for building the samples and also includes instructions for running each sample.

To build the samples:

1 If you require a proxy to connect to external Web sites, open the following files:

```
WISEMAN_HOME\samples\trafficlight_server\project.properties
WISEMAN_HOME\samples\trafficlight_client\project.properties
WISEMAN_HOME\samples\contacts\project.properties
WISEMAN_HOME\samples\contacts_client\project.properties
```

2 Modify the proxy properties appropriate for your environment.

3 Save and close the files.

4 From a command prompt, change directories to `WISEMAN_HOME\samples` and issue the following command:

```
ant
```

5 After the Ant script completes, copy the `WISEMAN_HOME\traffic.war` and `WISEMAN_HOME\users.war` file to the `TOMCAT_HOME\webapps` directory.

6 Using a text editor, open the `TOMCAT_HOME\conf\tomcat-users.xml` file and add the following credentials:

```
<role rolename="wsman"/>
<user username="wsman" password="secret" roles="wsman"/>
```

7 Save and close `tomcat-users.xml`.

8 Start Tomcat.

Accessing the WSDL

Wiseman provides a mechanism for accessing a resource's WSDL. This mechanism is also an easy way to verify that a deployment succeeded.

This procedure is only applicable for implementations that provide a WSDL. Implementations that utilize the Wiseman Metadata feature do not expose a WSDL. The contact sample is such an implementation. The Metadata feature is described later in this guide.

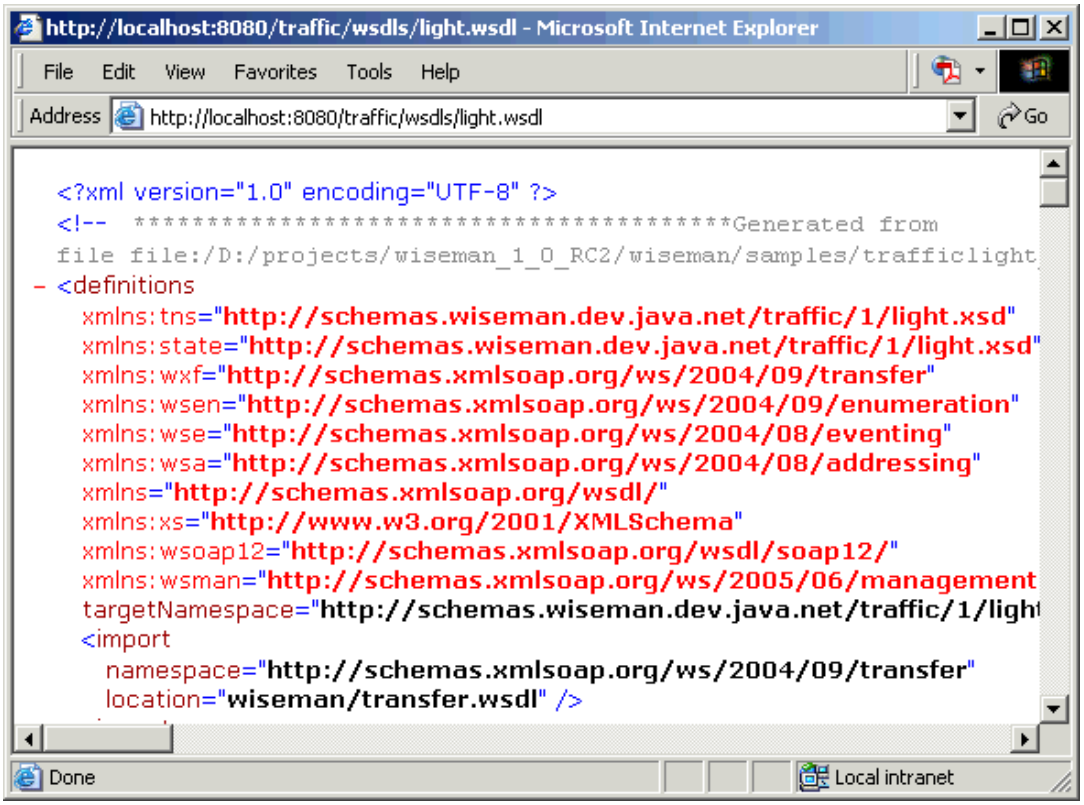
To access the traffic sample's WSDL:

1 Open a Web Browser.

2 Browse to the following address:

```
http://<host>:<port>/traffic/wsdls/light.wsdl
```

Replace <*host*>:<*port*> with the host name and port number for the Web container where the samples are deployed. The WSDL for the sample is returned as shown below:



Running the Traffic Sample

The Wiseman traffic sample demonstrates some of the features and implementation details of Wiseman. The sample represents a resource that manages a traffic light using a WS-Management-compliant Web service. The sample is located in the `WISEMAN_HOME\samples` directory.

The *Wiseman Tutorial* provides a step-by-step approach for exposing the management capabilities of the traffic light resource. In essence, the tutorial develops and deploys the sample traffic server that is used in this demonstration.

To run the traffic sample:

1 From a command prompt, change directories to `WISEMAN_HOME` and issue the following command to start the client:

```
java -jar traffic-client.jar
```

By default the Java logging level is set to `INFO`. Wiseman client SOAP messages are logged with level `FINE`. To see the SOAP messages printed on the console, create a `logging.properties` file with the following contents:

```
Handlers = java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level = FINE
com.sun.ws.management.level = FINE
```

Then, use the following command to start the client:

```
java -jar traffic-client.jar
-Djava.util.logging.config.file=.\logging.properties
```

Wiseman client SOAP messages with level `FINE` will then be logged to the console.

2. From the Traffic Light Client window, click **Create**. A new traffic light resource instance called `Light1` is created. A graphic representation of the traffic light displays in a separate window. This demonstrates a WS-Management Create Action.
3. From the Traffic Light Client window, click the Color drop-down list and select **green**.
4. Click **Put**. The `Light1` resource's green lamp illuminates. This demonstrates a WS-Management Put Action.
5. Create two additional traffic light resource instances.
6. Click **Enumerate**. Check the client's output to see the client iterate through the traffic instances. This demonstrates an Enumerate Action that uses a resource's factory iterator implementation.
7. Selecting any of the traffic light resource instances in the Traffic Light Client automatically results in a Get Action for that resource.

Running the Contacts Sample

The Wiseman contacts sample demonstrates using Wiseman's metadata feature to decorate a resource's WS-Management Web service. The Metadata is used to discover the Web service's capabilities and is also used by a client to easily consume the Web service. The contacts sample includes both a GUI client and a command-line client.

To run the contacts sample:

1. From a command prompt, change directories to `WISEMAN_HOME` and issue the following command to start the client:

   ```
   java -Dwsman.basicauthentication=true -Dwsman.user=wsman
   -Dwsman.password=secret -jar gui-users-client.jar
   ```

2. From the client, click **GET**. This demonstrates a WS-Management Get Action and uses the XPath expression in the ResourceState Filter field to get the age of the Bill.Gates resource. The response displays in the Message Response window.
3. Close the client.
4. From the same command prompt, run the command-line client by issuing the following command:

   ```
   java -Dwsman.basicauthentication=true -Dwsman.user=wsman
   -Dwsman.password=secret -jar users-client.jar
   ```

Setting Up a Development Environment

This section provides instructions for setting up a development environment when working with Wiseman. General instructions are provided as well as instructions specific for the Eclipse IDE.

General Setup Instructions

The following source paths and classpath are required to develop with Wiseman, including working with the Wiseman samples.

Source Path

- `WISEMAN_HOME/trafficlight_client/src`
- `WISEMAN_HOME/trafficlight_server/src`
- `WISEMAN_HOME/trafficlight_server/model/src`
- `WISEMAN_HOME/contacts/src`
- `WISEMAN_HOME/contacts_client/src`

Classpath

- `JAVA_HOME`
- `WISEMAN_HOME/lib/jaxws/activation.jar`
- `WISEMAN_HOME/lib/ant.jar`
- `WISEMAN_HOME/lib/junit.jar`
- `WISEMAN_HOME/lib/servlet-api.jar`
- `WISEMAN_HOME/lib/velocity-1.4.jar`
- `WISEMAN_HOME/lib/velocity-dep-1.4.jar`
- `WISEMAN_HOME/lib/wiseman_src.jar`
- `WISEMAN_HOME/lib/wiseman_test.jar`
- `WISEMAN_HOME/lib/wiseman.jar`
- `WISEMAN_HOME/lib/wiseman-client.jar`
- `WISEMAN_HOME/lib/wiseman-client-src.jar`
- `WISEMAN_HOME/lib/wiseman-tools.jar`
- `WISEMAN_HOME/lib/wsdl4j.jar`
- `WISEMAN_HOME/lib/jaxws/FastInfoset.jar`
- `WISEMAN_HOME/lib/jaxws/http.jar`
- `WISEMAN_HOME/lib/jaxws/jaxb-api.jar`
- `WISEMAN_HOME/lib/jaxws/jaxb-impl.jar`
- `WISEMAN_HOME/lib/jaxws/jaxb-xjc.jar`
- `WISEMAN_HOME/lib/jaxws/jaxws-api.jar`
- `WISEMAN_HOME/lib/jaxws/jaxws-rt.jar`
- `WISEMAN_HOME/lib/jaxws/jaxws-tools.jar`
- `WISEMAN_HOME/lib/jaxws/jsr173_api.jar`
- `WISEMAN_HOME/lib/jaxws/jsr181-api.jar`
- `WISEMAN_HOME/lib/jaxws/jsr250-api.jar`
- `WISEMAN_HOME/lib/jaxws/resolver.jar`

- `WISEMAN_HOME/lib/jaxws/saaj-api.jar`
- `WISEMAN_HOME/lib/jaxws/saaj-impl.jar`
- `WISEMAN_HOME/lib/jaxws/sjsxp.jar`

Eclipse Setup Instructions

Wiseman includes an Eclipse project file that can be reused. In addition, a .classpath file is provided that loads all the necessary source and library dependencies.

To setup a development environment:

1. Start Eclipse
2. Create a new Java project.
3. From the Window menu, select **Preferences->Java-Build Path->Classpath Variables**.
4. Add the following variable:

   `WISEMAN_HOME=`*<directory of Wiseman distribution>*
5. Click **OK**.
6. From the File menu, select **Import->General->Existing Projects into Workspace**.
7. Click **Next**.
8. Click **Browse** and select the `WISEMAN_HOME/samples` directory.
9. Click **Finish**. The sample source directories and all required libraries are imported into the project.

Unit Testing

Realistically, you will not be writing a client application as you develop your WS Management services. A unit test is used to call a service as part of the development process. To simplify the process of JUnit testing a resource, a class which extends JUnit's `TestCase` is provided called `WsManBaseTestSupport`. This class can be found in the `WISEMAN_HOME/lib/wiseman-test.jar`. By extending this test case, its utility methods can be used to call a resource and test it.

Wiseman also includes a command line utility that can be used to test WS-Management-based services. For more information, see the *Wiseman Client Developer's Guide* located in the `WISEMAN_HOME/docs`.

To encourage you to try developing unit tests for yourself, a sample unit test for the Traffic sample is provided. See `com.sun.traffic.light.test.TrafficLightTestCase`. This test case extends `WsManBaseTestSupport` and uses utility operations to perform both JAXB and XPath based testing of server responses. Users that are new to unit testing and JUnit, can go to *http://www.junit.org*.

3 Generating Source for a Resource

This chapter provides instructions for using Wiseman tools to generate the source files that are used to expose a resource as a WS-Management-compliant Web service. The chapter also describes each of the generated files.

The following sections are included in this chapter:

Overview

Wiseman includes two generation tools that are implemented as Ant tasks:

- **Xsd2Wsdl** – The Xsd2Wsdl tool is used to generate a WSDL from an XML schema file. The WSDL is a resource's service description and defines, among other things, the operations that are defined in the WS-management specification. The generated WSDL also contains template sections that can be used to implement any custom operations that are required by a resource.
- **Wsdl2Wsman** – The Wsdl2Wsman tool generates a set of source Java files based on the generated WSDL. The source files are used to implement the WSDL's operations specific to a resource's backend model and also include classes used to facilitate marshalling (creating an XML document from a Java object) and unmarshalling (creating a Java object from an XML document) for a resource.

The generation tools utilize various libraries such as Apache Velocity, JAXWS and JAXB. The auto generation process saves the developer from having to complete these tasks manually and can save time as well as possible errors.

Defining a Resource

A resource is defined in this documentation as an object with attributes and operations that represents a component to be managed and exposed as a WS-Management compliant Web service. Resources are modeled using a collection of primitive types, which are collected into complex structures. These structures are described in an XML schema (XSD) document. This section provides a brief description of how to create an XSD file; however, XSD is beyond the scope of this documentation.

A resource should be able to be described by an XSD document. The following example, taken from the Wiseman Traffic Light Sample, is an example of a simple schema document that models a traffic light. If you are new to schema, use the example as a template for modeling your own resource.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
   targetNamespace="http://schemas.wiseman.dev.java.net/traffic/1/light.xsd"
   elementFormDefault="qualified"
   blockDefault="#all"
   xmlns:tl="http://schemas.wiseman.dev.java.net/traffic/1/light.xsd"
   xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:complexType name="TrafficLightType">
      <xs:sequence>
         <xs:element name="name" type="xs:string"/>
         <xs:element name="color" type="xs:string"/>
         <xs:element name="x" type="xs:int"/>
         <xs:element name="y" type="xs:int"/>
      </xs:sequence>
   </xs:complexType>
   <xs:element name="trafficlight" type="tl:TrafficLightType"/>
</xs:schema>
```

This XSD describes a complex type called `TrafficLightType`. The `TrafficLightType` complex type defines a template for what this resource (represented as XML) requires in order to be implemented. In this case, the XSD describes a resource that manages a traffic light. It has four attributes or properties. These are `name`, `color`, `x` and `y`. The `x` and `y` attributes are the map coordinates of the light. An example traffic light resource based on this XSD might be described as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<tl:TrafficLightType
   xmlns:tl=" http://schemas.wiseman.dev.java.net/traffic/1/light.xsd">
   <tl:name>Light1</tl:name>
   <tl:color>red</tl:color >
   <tl:x>100</tl:x>
   <tl:y>210</tl:y>
</tl:TrafficLightType >
```

The example uses the namespace (`xmlns:tl="http://schemas.wiseman.dev.java.net/traffic/1/light.xsd"`) that is defined in the XSD file. Typically, two namespaces are in a schema document. The Schema Specification namespace, (`xmlns:xs="http://www.w3.org/2001/XMLSchema"`), which contains the elements used to describe a complex type and the namespace that is used to define the parts of your complex structure. Notice that the schema also declares a target namespace (`targetNamespace="http://schemas.wiseman.dev.java.net/traffic/1/light.xsd"`). This is the namespace that all elements without an explicit namespace will use. In this case, the target namespace and the resource's namespace match.

Using the Template Ant Script

Wiseman includes a project template that includes an Ant script that can be reused to generate the source files for any resource that is defined using XSD. The Ant script contains many targets that are discussed throughout this book; however, only the `generate` target is used in this procedure.

To generate source files using the template Ant script:

1. Create a working directory. This directory will be referred to as `WORK_DIR`.
2. Copy the `WISEMAN_HOME\samples\templates\wsdl_war_template.jar` template archive to `WORK_DIR`.

3 From a command prompt, change directories to WORK_DIR and issue the following command to extract the template.

jar –xf wsdl_war_template.jar

The project template files and directory structure are extracted. The wsdl_war_template.jar template archive can be deleted from WORK_DIR.

4 Copy a resource's schema to both the xsd and schemas directories.

Within the xsd directory, the schema must be located in a directory structure that matches the resource's namespace URI. For example, the sample traffic light schema defines the namespace URI http://schemas.wiseman.dev.java.net/traffic/1/ and therefore would be located in a directory as follows:

WORK_DIR/xsd/schemas.wiseman.dev.java.net/traffic/1/light.xsd

WORK_DIR/schemas/light.xsd

5 From WORK_DIR, open project.properties and modify the properties to reflect your environment and resource. In particular, change to the following properties are required:

- wiseman.root: the location of WISEMAN_HOME.
- war.context.path: the context path to use when deploying the WAR.
- user.wsdl.file: the name of the WSDL file to be generated
- user.xsd.file: the name of the schema file that was copied to WORK_DIR/xsd.
- war.filename: the name to be given to the generated WAR
- resource.uri: the resource's URI.

6 Save and close project.properties.

7 Open WORK_DIR/etc/binding.properties and modify the com.sun.ws.management.xml.custom.packagenames property to include custom package names for initializing a JAXBContext.

8 Save and close binding.properties.

9 From a command prompt, change directories to WORK_DIR and issue the following command:

ant generate

A WS-Management-compliant WSDL is created and copied to the WORK_DIR\wsdls directory. The Java source files are generated and copied to the WORK_DIR\gen-src directory.

These directories may be different if the properties file was modified to use different output directories.

Using the Ant Tasks

This section describes the Xsd2Wsdl and Wsdl2Wsman tools that are implemented as Ant targets. These targets can be used to create an Ant script that generates the necessary Wiseman source files. Both tools are used to generate the necessary source files. These tools are used in the Wiseman template Ant script.

Using the Xsd2Wsdl Tool

Wiseman includes a tool that generates a WS-Management-compliant WSDL file from an XSD file. The tool is an Ant task that is implemented in the `com.sun.ws.management.tools.Xsd2WsdlTask` class which is located in the `WISEMAN_HOME\lib\wiseman-tools.jar` file.

Task Definition

The task is defined as follows:

```
<taskdef name="xsd2wsdl"
   classname="com.sun.ws.management.tools.Xsd2WsdlTask"
   classpath="path/to/WISEMAN_HOME/lib/*.jar" />
```

Attributes

The Xsd2Wsdl tool takes the following attributes:

Table 1 Xsd2Wsdl Tool Attributes

Attribute	Description	Required
classpath	The classpath to be passed to Velocity. The classpath should reference all the jars in the `WISEMAN_HOME/lib` directory.	Yes
outputDir	Enter a directory where the generated WSDL is placed.	Yes
xsdFile	Enter the full path to the XSD file to be processed.	Yes
serviceAddress	Enter the service address to be used in the WSDL file. The address includes the host and port.	Yes
wsdlFileName	Enter a name to be given to the generated WSDL file.	No. Defaults to *<schema_filename>*.wsdl
resourceURI	Enter the resource URI to be used in the WSDL port definition.	No
wsdlTemplate	Enter the name of a custom WSDL template to be used during the generation process.	No
resourceName	Enter the resource name to be used in the WSDL.	No, unless the XSD contains multiple types
resourceType	Enter the resource type to be used in the WSDL.	No, unless the XSD contains multiple types
targetNamespace	Enter the target namespace to be used in the WSDL.	No, unless the XSD contains multiple types
singleHandler	Enter `true` or `false`. Set this attribute to `true` to indicate that a single handler be used.	No. Default is `true`

Example

```
<taskdef name="xsd2wsdl"
         classname="com.sun.ws.management.tools.Xsd2WsdlTask">
   <classpath>
      <fileset dir="${lib.dir}">
         <include name="*.jar"/>
      </fileset>
   </classpath>
</taskdef>

<xsd2wsdl xsdFile="${basedir}/my_resource.xsd"
          resourceURI="urn:myco.net/my/resource"
          outputDir="${basedir}/wsdls"
          serviceAddress="http://myco.net:8080/my_resource"/>
```

The `lib.dir` property must point to the `WISEMAN_HOME\lib` directory. This example generates a WSDL for the `my_resource.xsd` schema and writes the WSDL to the `\wsdls` directory. The Web service's port binding uses `urn:myco.net/my/resource` as the resource URI and the service address is `http://myco.net:8080/my_resource`.

Using the Wsdl2Wsman Tool

Wiseman includes a tool that generates Java classes from a WSDL file. The tool is an Ant task that is implemented in the `com.sun.ws.management.tools.Wsdl2WsmanTask` class which is located in the `WISEMAN_HOME\lib\wiseman-tools.jar` file.

Task Definition

The task is defined as follows:

```
<taskdef name="wsdl2wsman"
   classname="com.sun.ws.management.tools.Wsdl2WsmanTask"
   classpath=" path/to/WISEMAN_HOME/lib/**/*.jar " />
```

Attributes

The Wsdl2Wsman tool task takes the following attributes:

Table 2 Wsdl2Wsman Tool Attributes

Attribute	Description	Required
classpath	The classpath to be passed to wsdl2WsmanGenerator. The classpath should reference all the jars in the `WISEMAN_HOME/lib` directory.	Yes
generateJaxb	Enter `true` or `false`. Set this attribute to `true` to indicate that JAXB be used to generate Java bindings based on the WSDL.	No. Default is `true`
outputDir	Enter a directory where the generated source files are placed.	Yes
wsdlFile	Enter the full path to a WS-Management-compliant WSDL file.	Yes

Attribute	Description	Required
processAsSchema	If true, generate types by extracting schema sections from WSDL. If false, the WSDL file is directly processed.	No. Default is `false`

Example

```
<taskdef name="wsdl2wsman"
         classname="com.sun.ws.management.tools.Wsdl2WsmanTask" >
   <classpath>
      <fileset dir="${lib.dir}">
         <include name="*.jar"/>
      </fileset>
   </classpath>
</taskdef>
<wsdl2wsman generateJaxb="true"
            outputDir="${basedir}/gen-src"
            wsdlFile="${basedir}/my_wsdl.wsdl"
            processAsSchema="false" >
</wsdl2wsman>
```

The `lib.dir` property must point to the `WISEMAN_HOME\lib` directory. This example generates Java source files based on the `my_wsdl.wsdl` and outputs the source files to a directory called `gen-src`.

Explaining the Generated Files

The Wiseman generation tools create a WSDL file and a set of source Java files. These generated files are described below.

WSDL

The generated WSDL contains a single portType that organizes all of the operations that are defined in the WS-Management and accompanying specifications. The WSDL also includes a custom operation (including an operation binding) that can be used as a template to include as many custom operations that the resource requires. The operations are briefly described below. See the *WS-Management Specification* for complete information about the intended use of the operations. The instructions for implementing these operations are discussed later in this guide.

Table 3 WSDL Operations

Operation	Specification	Description
Get	WS-Transfer	The Get operation is used to retrieve an XML representation of a resource which corresponds to real-world objects.
Put	WS-Transfer	The Put operation is used to update a resource in its entirety and must conform to the resource's schema.

Operation	Specification	Description
Delete	WS-Transfer	The Delete operation removes (by either literal or logical deletion) a resource instance.
Create	WS-Transfer	The Create operation is used to construct a resource instance.
EnumerateOp	WS-Enumeration	The EnumerateOp operation begins a query for all instances of a resource provided the resource supports a mechanism for querying the instances.
PullOp	WS-Enumeration	The Pull operation iterates over the result set retrieved from an enumeration operation.
RenewOp	WS-Enumeration	The RenewOp operation renews an enumerator which may have timed out. The WS-Management specification does not recommend implementing this operation.
GetStatusOp	WS-Enumeration	The GetStatusOp operation gets the status of the enumerator. The WS-Management specification does not recommend implementing this operation.
ReleaseOp	WS-Enumeration	The releaseOp operation cancels an enumeration.
EnumerationEndOp	WS-Enumeration	The EnumerationEndOp operation notifies that an enumerator has terminated. The WS-Management specification does not recommend implementing this operation. This operation is currently commented out of the generated WSDL because of client code generation issues.
SubscribeOp	WS-Eventing	The SubscribeOp operation subscribes to a resource's events.
SubscriptionEnd	WS-Eventing	The SubscriptionEnd operation delivers a message to indicate that a subscription has terminated. The WS-Management specification makes this an optional operation.
RenewOp	WS-Eventing	The RenewOp operation renews a subscription prior to its expiration.
GetStatusOp	WS-Eventing	The GetStatusOp operation requests the status of a subscription. The WS-Management specification does not recommend implementing this operation.
UnsubscribeOp	WS-Eventing	The UnsubscribeOp operation cancels a subscription.

Source Java Files

The generated source Java files provide the necessary facilities to expose a resource as a WS-Management compliant Web service. The files implement the operations that are described in the generated WSDL. By default, the operations respond with a "Not Implemented" fault

for their actions until the operations are actually connected to a resource implementation. In addition, JAXB files are created to facilitate marshalling\unmarshalling the XML representation of a resource to Java.

Table 4 Generated Source Java Files

Java File	Description
<*resource*>_Handler	A handler that delegates to <*Resource*>Handler.
<*Resource*>Handler	A handler that implements WS-Management operations. The resource must be able to provide the actual implementation of the methods in this class.
<*resource*>IteratorFactory	Used to implement enumeration.
ObjectFactory	JAXB generated file. This class provides the ability to represent XML content as Java objects.
package-info	JAXB generated file. This class maps a schema to a Java package.
<*schema_defined_type*>	JAXB generated file. This class is a Java representation of the elements defined in the schema.
binding.properties	Properties file for JAXB.

4 Implementing Resource Access

This chapter provides instructions for implementing the WS-Transfer operations and also includes instructions for fragment-level access. These operations are typically implemented in the generated handler class and are implemented specific to a resource's model.

The following sections are included in this chapter:

Overview

The WS-Transfer specification defines four operations that are used to interact with a resource instance. When using the Wiseman tools, these operations are automatically defined in the resource's management WSDL and corresponding methods are automatically added to a resource's generated handler class. A developer uses the generated methods in the handler class to connect the operations to a resource's actual model.

The operations are:

- `Create` – The Create operation is used to construct a resource instance.
- `Get` – The Get operation is used to retrieve an XML representation of a resource which corresponds to real-world objects.
- `Put` – The Put operation is used to update a resource in its entirety and must conform to the resource's schema.
- `Delete` – The Delete operation removes (by either literal or logical deletion) a resource instance.

The transfer operations work on a single instance of a resource. If a resource can have multiple instances, the enumeration operations must be used. Enumeration is discussed in Chapter 5 "Managing Groups of Resources.

Wiseman also supports an extension to the transfer operations called fragment-level transfer. Fragment-level transfer allows a user to complete transfer operations on a subset of the resource model instead of always having to work with a resource's complete model. Each of the transfer operations supports using fragments.

Using the Handler Class

The Wiseman Wsdl2Wsman generation tool creates a handler class for a resource. The handler class is called *<resource>*`Handler.java` and is located in the `/net/..` directory of the generated source directory. You must modify this class specific to your resource.

The handler class contains a method for each of the WS-Transfer operations (`Create`, `Get`, `Put`, and `Delete`). The methods are used to connect the operations with a resource's actual back-end implementation. For example, the `create()` method would be implemented to construct an instance of a resource. How a resource actually fulfills the requirements of these methods is not defined or restricted by Wiseman. If a resource already contains facilities to satisfy these methods, then the facilities can be reused. However, if no facilities presently exist, they must be created.

The transfer operations need not be implemented if its functionality is not required. The methods in the generated handler class initially delegate to methods in the `TransferSupport` class.

```
public TransferSupport() {
super();
}

public void create(HandlerContext context,Management request, Management response) {
throw new ActionNotSupportedFault();
}

public void delete(HandlerContext context,Management request, Management response) {
throw new ActionNotSupportedFault();
}

public void get(HandlerContext context,Management request, Management response) {
throw new ActionNotSupportedFault();
}

public void put(HandlerContext context,Management request, Management response) {
throw new ActionNotSupportedFault();

        }
```

As generated these methods will compile, deploy and at runtime return a not implemented fault for every request.

The handler class also provides some useful tools to facilitate getting started. It has a JDK logger to help avoid using `System.out.print` in the handler.

```
private final Logger log;
```

Keep in mind that exceptions/faults are for clients to see and log information is for developers to read. Try to keep developer details in the log and keep faults simple and helpful to your clients.

Implementing a Create Action

A generated handler contains a `Create()` method that is used to create a resource instance. The method must be able to create a resource instance using the values specified in the request. The resource is associated with an endpoint reference.

The following example, from the Traffic Light sample, demonstrates an implementation of the `Create()` method.

```
public static void create(Management request, Management response, Logger log) {
   try {
          // Get the resource passed in the body
          TransferExtensions xferRequest = new TransferExtensions(request);
          TransferExtensions xferResponse = new TransferExtensions(response);
          Object element = xferRequest.getResource(QNAME);

          TrafficLight light;

          if (element != null) {
             JAXBElement<TrafficLightType> tlElement = getResource(request);
             TrafficLightType tlType = tlElement.getValue();

             // Create and store a reference to this object in our mini-model
             light = TrafficLightModel.getModel().create(tlType.getName());

             // Transfer values
             light.setColor(tlType.getColor());
             light.setX(tlType.getX());
             light.setY(tlType.getY());

          } else {

             // Create and store a reference to this object in our mini-model
             light = TrafficLightModel.getModel().create(null);
             log.log(
                Level.INFO, "The body of your request is empty but it is optional.");
          }

          // Define a selector (in this case name)
          HashMap<String, String> selectors = new HashMap<String, String>();
          selectors.put("name", light.getName());

          EndpointReferenceType epr = xferResponse.createEndpointReference(
             request.getTo(), request.getResourceURI(), selectors);
          xferResponse.setCreateResponse(epr);

   } catch (SOAPException e) {
      throw new InternalErrorFault(e);
   } catch (JAXBException e) {
      throw new InternalErrorFault(e);
   }
}
```

Implementing a Get Action

A generated handler contains a `Get()` method that is used to get a resource instance. The method must be able to query a resource for a given instance and return that instance in the service response.

The following steps are used to implement a Get Action:

1. Extract the Selectors from the SOAP header. The Selectors indicate which resource instance is being asked to provide its state.
2. Query your model and find the resource instance.
3. Create the JAXB Type for your resources XML document from its Factory which was pre-generated in your handler.

The following example, from the Traffic Light sample, demonstrates an implementation of the `Get()` method.

```
public static void get(Management request, Management response, Logger log) {

   try {

   // Use name selector to find the right light in the model
   String name = getNameSelector(request);
   TrafficLight light = TrafficLightModel.getModel().find(name);

   if (light == null) {
      log.info("An attempt was made to get a resource that did not exist called " +
         name);
      throw new InvalidSelectorsFault(
         InvalidSelectorsFault.Detail.INSUFFICIENT_SELECTORS);
   }

   TransferExtensions xferRequest = new TransferExtensions(request);
   TransferExtensions xferResponse = new TransferExtensions(response);
   xferResponse.setFragmentGetResponse(
      xferRequest.getFragmentHeader(), createLight(light));

   } catch JAXBException e) {
      throw new InternalErrorFault(e);
   } catch (SOAPException e) {
      throw new InternalErrorFault(e);
   }
}
```

Implementing a Put Action

A generated handler contains a `Put()` method that is used to update a resource instance. The method retrieves changes from the request and updates a resource instance in its entirety.

The following example, from the Traffic Light sample, demonstrates an implementation of the `Put()` method.

```
public static void put(Management request, Management response, Logger log) {

   // Use name selector to find the right light
   String name = getNameSelector(request);
   TrafficLight light = TrafficLightModel.getModel().find(name);
```

```
    if (light == null) {
       throw new InvalidSelectorsFault(
          InvalidSelectorsFault.Detail.INSUFFICIENT_SELECTORS);
    }

    // Get the resource passed in the body
    Object obj = getResource(request);

    if ((obj instanceof JAXBElement) == false) {
       throw new InternalErrorFault("Wrong resource type \n");
    }

    JAXBElement elem = (JAXBElement) obj;
    JAXBElement<TrafficLightType> tlElement = (JAXBElement<TrafficLightType>) obj;
    TrafficLightType tlType = tlElement.getValue();

    // Transfer values
    light.setName(tlType.getName());
    light.setColor(tlType.getColor());
    light.setX(tlType.getX());
    light.setY(tlType.getY());

    try {
       TransferExtensions xferResponse = new TransferExtensions(response);
       xferResponse.setPutResponse();
    } catch (SOAPException e) {
       throw new InternalErrorFault(e);
    }
 }
```

Implementing a Delete Action

A generated handler contains a `Delete()` method that is used to delete a resource instance. The method must be able to query a resource for a given instance and then delete the resource instance.

The following example, from the Traffic Light sample, demonstrates an implementation of the `Delete()` method.

```
 public static void delete(Management request, Management response, Logger log) {

    // Use name selector to find the right light
    String name = getNameSelector(request);
    TrafficLight light = TrafficLightModel.getModel().find(name);

    if (light == null) {
       log.log(Level.WARNING, "An attempt was made to delete a resource that
          did not exist called " + name);
       throw new InvalidSelectorsFault(
          InvalidSelectorsFault.Detail.INSUFFICIENT_SELECTORS);
    }

    // Remove it from the list and then remove the actual GUI instance.
    TrafficLightModel.getModel().destroy(name);
    try {
       TransferExtensions xferResponse = new TransferExtensions(response);
       xferResponse.setDeleteResponse();
    } catch (SOAPException e) {
```

```
            throw new InternalErrorFault(e);
        }
    }
```

Using Fragment Resource Access

The WS-Transfer operations are used to work on a complete instance of a resource. This means that a developer must always work with the complete model of a resource even if only a portion of the model is of interest. For example, a management client may only wish to get the available space on a disk resource, but may not be interested in the manufacturer, speed, or total amount of space on the disk.

A separate EPR could be specified for portions of the model; however, for many resources this would be impractical and would lead to an excessive amount of EPRs to maintain. As an alternative, the WS-Management specification extends the WS-Transfer operations in order to support fragment-level (property) access of resources. The underlying mechanism used to implement fragment-level access is typically XPath. However, any selection dialect could potentially be used. XPath is beyond the scope of this documentation.

Wiseman supports fragment-level access and provides two methods that can be used to implement fragment-level access when performing any of the WS-Transfer operations. The methods are part of the `TransferSupport` class and are used to retrieve and parse the fragment transfer header from the transfer request message.

- `SOAPElement locateFragmentHeader(SOAPElement[] allHeaders)` – returns the fragment transfer element (if any) from the list of SOAP headers
- `String extractFragmentMessage(SOAPElement element)` – returns the XML fragment from the specified element

Fragment-level access is also supported when using enumeration. See the “Managing Groups of Resource” chapter for complete details.

Implementing a Fragment-Level Create Action

At this time, a fragment-level create operation is functionally identical to a non-fragment create operation. The point of a fragment operation is to modify a few subcomponents/properties of what would otherwise be an atomic element in the strictly Transfer sense. To be able to modify only a few of the properties of the resource, the XSD must be designed so that some of the properties are optional. This means that any `Transfer.Create` operation, where not every one of the optional properties is specified, is a valid `Transfer.Create` and a valid fragment-level create action. This nullifies the use of fragment header. As described, the two operations are indistinguishable and there is no longer any need to discuss such functionality as it has already been described above in the `Transfer.Create` section. A fragment-level create request does not make sense in accordance with the WS-Management specification.

Implementing a Fragment-Level Get Action

Fragment-Level get actions are similar to regular get actions except a specified XPath expression is executed against the representation that would normally have been retrieved. The result is a fragment instead of the complete resource representation.

The following example demonstrates a fragment-level get implementation for the Traffic Light sample. It would be implemented in the handler's `Get()` method. Code is required to search for and apply the data from the fragment transfer header. The following example assumes that the client is using XPath to define the fragment.

```
try {

   // Use name selector to find the right light in the model
   String name = getNameSelector(request);
   TrafficLight light = TrafficLightModel.getModel().find(name);

   if (light == null) {
      log.info(
         "An attempt was made to get a resource that did not exist called " + name);
      throw new InvalidSelectorsFault(
         InvalidSelectorsFault.Detail.INSUFFICIENT_SELECTORS);
   }

   TransferExtensions xferRequest = new TransferExtensions(request);
   TransferExtensions xferResponse = new TransferExtensions(response);

   //################################################

   //locate fragment header
   Document xmlObj = Management.newDocument();
   SOAPElement headerElement = null;
   Element element = null;
   try {
      headerElement = ManagementUtility.locateHeader(request.getHeaders(),
         TransferExtensions.FRAGMENT_TRANSFER);

      //if header sent then pull out body for fragment parsing
      if(headerElement!=null){
         String content = headerElement.getTextContent();

         //locate content for return
         String NS = "http://schemas.wiseman.dev.java.net/traffic/1/light.xsd";
         if((content!=null)&&(content.indexOf("color")>-1)){
            element= xmlObj.createElementNS(NS, "t1:color");
            element.setTextContent(light.getColor());
         }
         else if((content!=null)&&(content.indexOf("name")>-1)){
            element= xmlObj.createElementNS(NS, "t1:name");
            element.setTextContent(light.getName());
         }
         else if((content!=null)&&(content.indexOf("x")>-1)){
            element= xmlObj.createElementNS(NS, "t1:x");
            element.setTextContent(light.getX()+"");
         }
         else if((content!=null)&&(content.indexOf("y")>-1)){
            element= xmlObj.createElementNS(NS, "t1:y");
            element.setTextContent(light.getY()+"");
         }
      }
   } catch (SOAPException e1) {
      e1.printStackTrace();
   }

   final DocumentFragment fragment = xmlObj.createDocumentFragment();
   fragment.appendChild(element);
   final Object xmlFragment =
      BaseSupport.createXmlFragment(((DocumentFragment)fragment).getChildNodes());
```

```
   if(headerElement!=null){
      xferResponse.setFragmentGetResponse(xferRequest.getFragmentHeader(), xmlFragment);
   }else{
      xferResponse.setFragmentGetResponse(
         xferRequest.getFragmentHeader(), createLight(light));
   }

   //###########################################
   //  xferResponse.setFragmentGetResponse(
   //     xferRequest.getFragmentHeader(), createLight(light));

} catch (JAXBException e) {
throw new InternalErrorFault(e);
} catch (SOAPException e) {
throw new InternalErrorFault(e);
}
```

Implementing a Fragment-Level Put Action

Fragment-Level put actions are similar to regular put actions except that they only transfer the fragment being updated. While the fragment may be considered part of an instance from the observer's perspective, the referenced fragment is treated as the complete instance during the execution of the operation.

The following example demonstrates a fragment-level put implementation for the Traffic Light sample. It would be implemented in the handler's `Put()` method. The server must identify which fields have been specified in the XML fragment, and then update those fields in the data item specified by the selector set. The following example implementation assumes that the client is using XPath to define the fragment.

```
// Use name selector to find the right light
String name = getNameSelector(request);
TrafficLight light = TrafficLightModel.getModel().find(name);

if (light == null) {
   throw new InvalidSelectorsFault(InvalidSelectorsFault.Detail.INSUFFICIENT_SELECTORS);
}

//###########################################
//locate fragment header

SOAPElement headerElement = null;
try {
   headerElement = ManagementUtility.locateHeader(request.getHeaders(),
      TransferExtensions.FRAGMENT_TRANSFER);

   //if header sent then pull out body for fragment parsing
   if(headerElement!=null){
      String content = headerElement.getTextContent();

      //locate child
      Node element= request.getBody().getFirstChild().getFirstChild();
         if((content!=null)&&(content.indexOf("color")>-1)){
            light.setColor(element.getTextContent());
         }
         else if((content!=null)&&(content.indexOf("name")>-1)){
            light.setName(element.getTextContent());
         }
         else if((content!=null)&&(content.indexOf("x")>-1)){
            int value = Integer.parseInt(element.getTextContent());
```

```
            light.setX(value);
         }
         else if((content!=null)&&(content.indexOf("y")>-1)){
            int value = Integer.parseInt(element.getTextContent());
            light.setY(value);
         }
      }
   } catch (SOAPException e1) {
      e1.printStackTrace();
   }
   Object obj = null;
   if(headerElement!=null){
      TrafficLightType jaxbEl = trafficLightFactory.createTrafficLightType();
      jaxbEl.setColor(light.getColor());
      jaxbEl.setName(light.getName());
      jaxbEl.setX(light.getX());
      jaxbEl.setY(light.getY());
      obj=trafficLightFactory.createTrafficlight(jaxbEl);
   }else{
      obj=getResource(request);
   }

   //##############################################
   // Object obj = getResource(request);

   if ((obj instanceof JAXBElement) == false) {
      throw new InternalErrorFault("Wrong resource type \n");
   }

   JAXBElement elem = (JAXBElement) obj;
   JAXBElement<TrafficLightType> tlElement = (JAXBElement<TrafficLightType>) obj;
   TrafficLightType tlType = tlElement.getValue();

   // Transfer values
   light.setName(tlType.getName());
   light.setColor(tlType.getColor());
   light.setX(tlType.getX());
   light.setY(tlType.getY());

   try {
      TransferExtensions xferResponse = new TransferExtensions(response);
      xferResponse.setPutResponse();
   } catch (SOAPException e) {
      throw new InternalErrorFault(e);
   }
```

Implementing a Fragment-Level Delete Action

A fragment-level delete action involves locating the fields specified in the fragment and removing them from the data item indicated by the selector set in the request header. Only optional fields may be removed from a data item. The purpose of a `Transfer.Delete` operation is to delete the entire resource specified by the selectors passed in. It is not possible to delete portions of a Resource as this functionality is functionally identical and more correctly defined by the fragment put operation where the resource representation is updated. In other words, the notion of fragment-level delete does not make sense in regard to how fragment operations are specified by the WS-Management specification.

5 Managing Groups of Resources

This chapter provides instructions for enumerating over many instances of a resource. This functionality is provided by the WS-Enumeration operations. However, enumeration must be implemented specific to a resource's model.

The following sections are included in this chapter:

Overview

The WS-Transfer operations are used on single instances of a resource. WS-Enumeration is used when a resource has multiple instances. The WS-Enumeration specification defines six operations, but only three operations are required and typically implemented. The operations are:

- Enumerate – The Enumerate operation establish the enumeration context and begins a query for all instances of a resource provided the resource supports a mechanism for querying the instances.
- Pull – The Pull operation iterates over the result set retrieved from an enumeration operation.
- Release – The Release operation cancels an enumeration and any associated resources.

When using the Wiseman tools, these operations are automatically defined in the resource's management WSDL and corresponding methods (EnumerateOp, PullOp, and ReleaseOp) are automatically added to a resource's generated handler class. A developer uses the generated methods in the handler class to connect the operations to a resource's actual enumeration model.

An application developer that wishes to expose a resource that has more than one instance should support the Enumerate, Pull, and Release operations. Wiseman has provided several support classes to assist the developer in this effort. These classes provide the following functionality to the developer:

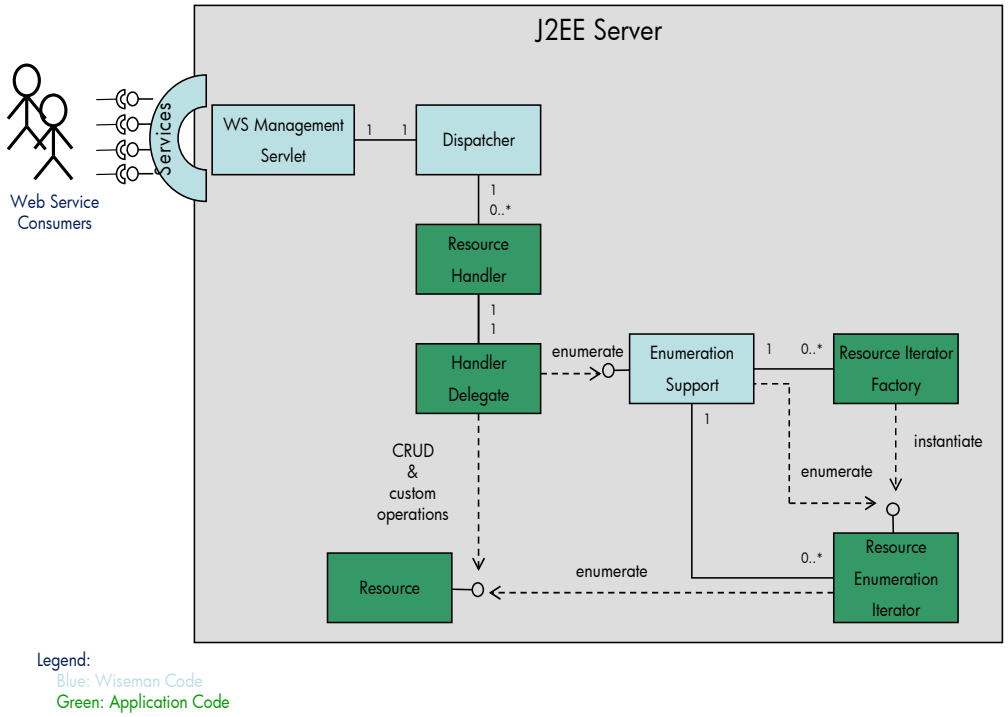
- Maintaining an enumeration context between Web service calls such as Enumerate, Pull and Release
- Monitoring for expiration of enumeration context and proper cleanup of resources
- Managing of OperationTimeout for a particular Web Service call

- Filtering of the response objects
- Processing of Enumerate, Pull, and Release requests
- Building proper Enumerate, Pull, and Release responses

Much of the above functionality is provided by the `com.sun.ws.management.server.EnumerationSupport` class. This is the main Wiseman support class used by application developers to provide Enumeration support for their resource instances.

The following diagram gives a depiction of the basic architecture on the Wiseman server side. Items in blue are Wiseman code. Items in green are Application developer code. In the center, `EnumerationSupport` is depicted.

Figure 2 EnumerationSupport



The Resource Handler class calls `EnumerationSupport` to initiate and process an Enumeration, Pull or Release call. `EnumerationSupport` then queries the Resource Enumeration Iterator to collect the resource instances and build a response for the client.

Enumerate Overview

The following list provides an overview of the run-time events that are associated with an Enumeration operation.

- When an enumerate request is received, the <*resource*>`Handler` class passes the request to `EnumerationSupport.enumerate()` for further processing. Notice that the <*resource*>`Handler` class is normally subclassed to the Wiseman `EnumerationHandler` class, or the `ResourceHandler` class. These classes, by default, will automatically call `EnumerationSupport.enumerate()` for an enumeration request, so the application developer only needs to subclass the <*resource*>`Handler` class to the Wiseman `EnumerationHandler` or `ResourceHandler`. The superclass will call `EnumerationSupport` for the <*resource*>`Handler` class.
- `EnumerationSupport.enumerate()` processes the Enumerate request. To do so it will need to instantiate a resource iterator to provide the resource instances. The resource iterator must be written by the application developer. To instantiate the resource iterator, `EnumerationSupport.enumerate()`uses the <*resource*>`IteratorFactory` class which is automatically generated and must be implemented by the application developer. For each resource URI there is one <*resource*>`IteratorFactory` class registered with `EnumerationSupport`. Once `EnumerationSupport.enumerate()` has validated the request and created the Iterator to provide the resource instances, it will create a context ID and context to maintain the context between WS calls. `EnumerationSupport.enumerate()` then builds a response message with the context ID and returns the response message to the <*resource*>`Handler` class.
- The <*resource*>`Handler` class can add any additional information to the message and return it back to the Resource Handler who in turn returns the response to the Wiseman Dispatcher. Eventually the response message is returned to the Web Service Consumer.
- Once the consumer has received the response with the enumeration context, it may begin issuing Pull requests in order to obtain the resource instances.

Pull Overview

The following list provides an overview of the run-time events that are associated with a Pull operation.

- When a pull request is received, the <*resource*>`Handler` class passes the request on to `EnumerationSupport.pull()`for further processing. Notice that the <*resource*>`Handler` class is normally subclassed to the Wiseman `EnumerationHandler` class, or the `ResourceHandler` class. These classes, by default, will automatically call `EnumerationSupport.pull()` for a pull request, so the application developer only needs to subclass the <*resource*>`Handler` class to the Wiseman `EnumerationHandler` or `ResourceHandler`. The superclass will call `EnumerationSupport` for the <*resource*>`Handler` class.
- `EnumerationSupport.pull()` checks if the request has a valid enumeration context ID and locates the context. If it is not valid, the proper fault is returned to the Web Service consumer, otherwise the request is processed. `EnumerationSupport.pull()`locates the Iterator instance for this context and calls `hasNext()` and `next()` to obtain the resource instances from the Iterator. As it does so, it will apply any filter specified by the WS Consumer and collect the number of instances requested by the WS Consumer. Once the requested number of resource instances are collected from the Iterator, `EnumerationSupport.pull()` will build a response message to return to the <*resource*>`Handler` class.
- The <*resource*>`Handler` class can add any additional information to the message and return it back to the Resource Handler who in turn returns the response to the Wiseman Dispatcher. Eventually the response message is returned to the Web Service Consumer.

- Once the consumer has received the response with the resource instances, it may issue more Pull requests until all resources have been obtained from the Service. This is indicated by an end of sequence flag in the final Pull response.
- When the Iterator returns `false` from `hasNext()` this indicates end of sequence has been reached, `EnumerationSupport.pull()` will set the end of sequence flag in the response and release the context. After this response the context is no longer valid.

Release Overview

The following list provides an overview of the run-time events that are associated with a Release operation.

- When a release request is received, the <*resource*>`Handler` class passes the request on to `EnumerationSupport.release()` for further processing. Notice that the <*resource*>`Handler` class is normally subclassed to the Wiseman `EnumerationHandler` class, or the `ResourceHandler` class. These classes, by default, will automatically call `EnumerationSupport.release()` for a release request, so the application developer only needs to subclass the <*resource*>`Handler` class to the Wiseman `EnumerationHandler` or `ResourceHandler`. The superclass will call `EnumerationSupport` for the <*resource*>`Handler` class.
- `EnumerationSupport.release()` checks if the request has a valid enumeration context ID and locates the context. If it is not valid, the proper fault is returned to the Web Service consumer, otherwise the request is processed. `EnumerationSupport.release()`locates the Iterator instance and calls `release()` on the Iterator in order to indicate to the Iterator that it may release any resources it has. `EnumerationSupport.release()` then releases the context and builds the release response message for the WS Consumer. The response message is then returned to the <*resource*>`Handler` class.
- The <*resource*>`Handler` class can add any additional information to the message and return it back to the Resource Handler who in turn returns the response to the Wiseman Dispatcher. Eventually the response message is returned to the Web Service Consumer.

Implementing the Factory Iterator

There is one <*resource*>`IteratorFactory` class for each resource URI supported. This class must be written by the application developer and must implement the `com.sun.ws.management.ser.IteratorFactory` interface.

```
public interface IteratorFactory {
   public EnumerationIterator newIterator(final HandlerContext context,
                                          final Enumeration request,
                                          final DocumentBuilder db,
                                          final boolean includeItem,
                                          final boolean includeEPR)
   throws UnsupportedFeatureFault, FaultException;
}
```

This interface has just one method, `newIterator()`. The Wiseman Wsdl2Wsman tool automatically creates a <*resource*>`IteratorFactory` class and template Iterator class. The <*resource*>`IteratorFactory` class is registered with `EnumerationSupport` by calling `EnumerationSupport.registerIteratorFactory()`. The generated <*resource*>`Handler` class

automatically contains a call to EnumerationSupport.registerIteratorFactory() in the constructor.

The following example is from the Traffic Light sample.

```
public LightHandler() {
   super();

   // Initialize our member variables
   log = Logger.getLogger(LightHandler.class.getName());
   try {

      // Register the IteratorFactory with EnumerationSupport
      EnumerationSupport.registerIteratorFactory(
         "urn:resources.wiseman.dev.java.net/traffic/1/light",
         new LightIteratorFactory("urn:resources.wiseman.dev.java.net/traffic/1/light"));
   } catch (Exception e) {
      throw new InternalErrorFault(e);
   }
}
```

Implementing Enumeration Iteration

For each WS Consumer Enumerate request, a Resource Enumeration Iterator instance is created. The instance is created by `EnumerationSupport.enumerate()` calling the newIterator() method in the *<resource>*`IteratorFactory` class. This class is written by the application developer and must implement the `com.sun.ws.management.ser.EnumerationIterator` interface.

```
public interface EnumerationIterator {

   int estimateTotalItems();

   boolean isFiltered();

   boolean hasNext();

   EnumerationItem next();

   void release();

}
```

There are five methods the application developer must implement. The generated *<resource>*`IteratorFactory` class automatically contains these methods. The developer needs to add specific code to access their resources. Code that needs to be added is clearly marked with `TODO` comments.

The following example is from the Traffic Light sample. The Traffic Light sample uses a separate implementation class (`LightIteratorImpl`) to implement these methods. The *<resource>*`IteratorFactory` methods call the implementation class's methods

```
public class LightIteratorImpl {

   private static final String WSMAN_TRAFFIC_RESOURCE =
      "urn:resources.wiseman.dev.java.net/traffic/1/light";

   private static final Logger log = Logger.getLogger(LightIteratorImpl.class.getName());
```

```
    private int length;
    private final String address;
    private final boolean includeEPR;

    private Iterator<TrafficLight> lights;

    public LightIteratorImpl(String address, boolean includeEPR) {
       this.address = address;
       this.includeEPR = includeEPR;

       Collection<TrafficLight> coll = TrafficLightModel.getModel().getList().values();
       this.length = coll.size();
       this.lights = coll.iterator();
    }

    @SuppressWarnings("unchecked")
    public EnumerationItem next() {

       // Get the next light
       TrafficLight light = lights.next();
       Map<String, String> selectors = new HashMap<String, String>();
       selectors.put("name", light.getName());
       try {
          final EndpointReferenceType epr;

          if (includeEPR == true) {
             epr = TransferSupport.createEpr(address, WSMAN_TRAFFIC_RESOURCE, selectors);

          } else {
             epr = null;
          }

          // Create the EnumerationItem and return it
          JAXBElement<TrafficLightType> lightElement =
             LightHandlerImpl.createLight(light);
          return new EnumerationItem(lightElement, epr);
       } catch (JAXBException e) {
          throw new InternalErrorFault(e.getMessage());
       }
    }

    public int estimateTotalItems() {
       return length;
    }

    public boolean hasNext() {
       return lights.hasNext();
    }

    public void release() {
       length = 0;
       lights = new ArrayList<TrafficLight>().iterator();
    }
}
```

Implementing Enumeration Item

The `EnumerationItem` object is used to hold a single resource instance and an endpoint reference to that resource instance. It is the return value from the `next()` method in the Enumeration Iterator (previously discussed). For each call to `next()` the application developer creates an `EnumerationItem` that holds a resource item and its EPR.

WS Consumers can request either the resource alone, the resource and its EPR or the EPR alone. This is specified in the `Enumerate` call and is passed to the `newIterator()` method in the <*resource*>`IteratorFactory` class. In general, it is best to always set both in the `EnumerationItem`, and allow `EnumerationSupport` to choose which are returned to the WS Consumer. If `EnumerationSupport` filters the request, it always needs the resource itself in order to apply the filter, even in the case where the WS Consumer only wants the EPR.

The following example is from the Traffic Light sample and shows how to build the `EnumerationItem` in the `next()` method.

```
public EnumerationItem next() {

      // Get the next light
      TrafficLight light = lights.next();
      Map<String, String> selectors = new HashMap<String, String>();
      selectors.put("name", light.getName());
      try {
         final EndpointReferenceType epr;

         if (includeEPR == true) {
            epr = TransferSupport.createEpr(address, WSMAN_TRAFFIC_RESOURCE, selectors);

         } else {
            epr = null;
         }

         // Create the EnumerationItem and return it
         JAXBElement<TrafficLightType> lightElement =
            LightHandlerImpl.createLight(light);
         return new EnumerationItem(lightElement, epr);
      } catch (JAXBException e) {
         throw new InternalErrorFault(e.getMessage());
      }
   }
```

The resource item passed in the `EnumerationItem` must be an element that can be marshaled by the `XmlBinding` class. A `JAXBContext` is used to do the marshalling so this can either be a `org.w3c.dom.Document`, `org.w3c.dom.Element`, or a `JAXBElement` recognized by the `XmlBinding` object. See below for more information on `XmlBinding`.

Using EPR and Selectors

Endpoint References (EPRs) are part of the WS Addressing definition. WS-Management specification uses the August 2004 version of the WS-Addressing standard. The definition in the standard states:

“A Web service endpoint is a (referenceable) entity, processor, or resource where Web service messages can be targeted. Endpoint references convey the information needed to

identify/reference a Web service endpoint, and may be used in several different ways: endpoint references are suitable for conveying the information needed to access a Web service endpoint, but are also used to provide addresses for individual messages sent to and from Web services."

EPRs are used in WS-Enumeration to address a single resource as well as a set of resources. An EPR with no selector set will typically address all resources instances at a Web service endpoint. For example, If the Traffic Light sample did not use a selector, then a request would return all traffic light instances. An EPR with the selector `Name="name"` and value of `Light1` will address the single light resource with the name `Light1`.

Similarly an EPR with the selector `Name="color"` and value of `red` would address the subset of lights that are `red`.

To support this type of subset addressing, examine the selector set in the Enumerate request message from the Web service consumer and appropriately filter the response objects. The Traffic Light sample does not support this feature.

Using Filtering

Filter handling for enumerations is implemented in the `EnumerationHandler` class. The `EnumerationSupport` class is encapsulated by the `EnumerationHandler` class. The static method `EnumerationSupport.createFilter()` is used to create a filter object from the enumeration request object. This method parses the request and creates a Filter object instance if a filter is specified in the request's header. This method is used by the `EnumerationSupport.enumerate()` method to locate and apply any user specified filters.

Filtering, by default supports XPath filters and any other filter dialect registered with `com.sun.ws.management.BaseSupport`. Wiseman automatically registers an XPath 1.0 filter. Other filters may be registered by individual applications. For reasons such as performance or custom filter support, the Enumeration Iterator may wish to pre-filter the resource instances and disable filtering by `EnumerationSupport`. The application developer can disable `EnumerationSupport` filtering by always returning `true` to calls to the Iterator's `isFiltered()` method.

Fragment Processing

WS-Management allows for the XPath filter to select only fragments of a resource. See "Using Fragment Resource Access" in Chapter 4. `EnumerationSupport` filtering supports fragment processing. An application that uses `EnumerationSupport` to do fragment processing, must return `false` for the Iterator's `isFiltered()`method. It may still pre-filter the resources and allow `EnumerationSupport` to do the fragment processing on the pre-filtered resources.

Implementing Timeout Processing

There are two types of timeouts that concern a Web Service consumer: Expires and Operation Timeout. They are both discussed below.

Expires

A consumer may request an Expires timeout in the Enumerate request. This feature, defined in WS-Enumeration, sets a timeout for the entire enumeration operation from enumerate request to release. If the timeout occurs before the consumer has completed the enumeration, the server may invalidate the enumeration context and release any resources.

The `EnumerationSupport` class provides support for the Expires timeout. A background thread monitors all enumeration contexts and will release the context if the expiration occurs. When the context is released, `EnumerationSupport` calls the Iterator's `release()` method to indicate to the Iterator that it may release any resources it is holding.

Operation Timeout

For each enumeration operation, the consumer may set an `OperationTimeout` in the request. `Operationtimeout` is defined by WS-Management. The request must not take any longer to complete than the specified `OperationTimeout`.

Operation timeout is also valid for WS-Transfer operations. Wiseman supports this timeout feature on the server side and terminates any request that exceeds this requested value. If `OperationTimeout` is not specified by the consumer, a default value of 5 minutes is set by the Wiseman server. This value may be changed by setting the `OperationTimeoutDefault` property in the `wiseman.properties` file included in the WAR. The value is entered as milliseconds.

`EnumerationSupport.pull()` keeps track of the `OperationTimeout` as it loops over the `next()` call. If the time expires before the requested number of MaxElements are collected, the `EnumerationSupport` class returns the number of elements collected up to this point.

The `next()` call should not block for any extended length of time. If for any reason the next item is not readily available, the Iterator should return `null` to indicate that the next item is not available at this time. This is different from the `hasNext()` method which indicates that there are no more items at all to be expected from this Iterator. `EnumerationSupport.pull()`calls `wait(<OperationTimeout>)` on the Iterator and blocks waiting for the next item, if no items have yet been collected, otherwise it returns any items it has already collected and ends the current pull request.

When the next item becomes available, the Iterator should indicate that it is now available by calling `notifyAll()` on the Iterator. This must be done by the application developer in a separate thread, and is usually used in Eventing Pull operations. The `notifyAll()`awakes any pending `wait(<OperationTimeout>)` call on the Iterator and `EnumerationSupport` continues processing the pull request.

If the OperationTimeout occurs before the next item becomes available, and no items have yet been collected, `EnumerationSupport` returns a `wsman:TimedOut` fault to the consumer.

Implementing Notifications during a Pull Operation

Notifications can be implemented at the beginning and end of a pull sequence. The EnumerationPullIterator interface is used to implement this feature.

```
public interface EnumerationPullIterator extends EnumerationIterator {
```

```
    public void startPull(final HandlerContext context, final Enumeration request);

    public void endPull(final Enumeration response);
}
```

This interface provides two additional methods that are called to provide additional information to the Iterator.

If an application developer requires notification of creation and deletion of an enumeration context, it should implement the `ContextListener` interface and pass this object in the `EnumerationSupport.enumerate()` call. This will notify the listener when the enumeration context is bound and unbound.

Best Practices

The following list provides some best practices when implementing enumeration:

- Always return both the item and the EPR regardless of whether the consumer wants one or both. This will allow the `EnumerationSupport` class to do any filtering.
- If XPath filtering is being used and the Iterator filters the requests, the Iterator should still set `isFiltered()` to false, as EnumerationSupport will then provide XmlFragment support for the filtered items.
- The Iterator's `next()` method should not block for any extended length of time, or it may cause the OperationTimeout to be exceeded. See "Implementing Timeout Processing" above.
- Iterator Factories should be registered with the `EnumerationSupport` class. Eventing Iterator Factories used in Eventing Pull should be registered with the `EventingSupport` class. Both are called by `EnumerationSupport.pull()`.
- Release any Iterator resources when the Iterator `release()` method is called.
- Set the address in any EPRs created to the actual address of the service, and not the anonymous address. This saves work on the client side. The address can be obtained from the EPR in the request or the `HandlerContext` specified in the `newIterator()` call.
- EPRs created in Enumeration are used to address individual resource for Read, Put, or Delete operations. Therefore, they must address the Web service endpoint for these operations.
- It is good practice to use one Resource URI for all operations, e.g. Enumeration, Create, Get, Put and Delete, as well as custom operations.

6 Providing Metadata

This chapter provides instructions for using metadata to discover what management operations are available for a managed resource and how to access the operations. Metadata usage is not defined or required by the WS-Management specification and is provided by Wiseman as a convenience.

The following sections are included in this chapter:

Overview

Metadata can have many different meanings depending on the context in which the metadata is defined. In Wiseman, metadata is used to describe how to access a resource's transfer, enumeration and eventing functionality. More specifically, the metadata describes the EPRs, Addressing.To, ResourceURIs, SelectorSets, Options, and usage guidelines needed to access WS-Management-compliant resources.

There is a clear need for WS-Management developers to be able to include metadata. However, the WS-Management specification 1.0(version A) does not explicitly describe how to address this need. Moreover, the specification alludes to using Management Catalog, but work in this area is still in the pre-DMTF namespace and unfinished.

Wiseman provides a proprietary metadata implementation that is based on the WS-MetadataExchange specification. The implementation is an interim solution and is not, in any way, related to the WS-Management specification. Developers can use this solution for their metadata needs until the WS-Management specification defines a solution in this area.

Wiseman implements metadata exposure for Wiseman-enabled servers/clients by exposing metadata content using `Transfer.Get` calls to a metadata endpoint/address. This functionality must be enabled and described using the appropriate properties in the `metadata.properties` file. Annotations are used to decorate a resource's handler class. All annotated handler class names need to be added to the `metadata-handlers.properties` file for successful exposure. When enabled, decorated handler classes will have their metadata exposed on the Metadata resource.

Adding Annotations

Annotations are added to a resource's handler class. There are two annotations that can be defined:

`com.sun.ws.management.metadata.annotations.*`

- `WsManagementDefaultAddressingModelAnnotation`
- `WsManagementEnumerationAnnotation`

`WsManagementDefaultAddressingModelAnnotation` abstracts the general details for a WS-Management instance that supports the `DefaultAddressingModel` type defined in the WS-Management specification. This annotation is primarily meant for resources that implement WS-Transfer operations.

`WsManagementEnumerationAnnotation` includes a `WsManagementDefaultAddressingModelAnnotation` annotation while adding a few more fields that should be useful for developers attempting to describe Enumeration. This annotation is meant for resources that implement WS-Enumeration operations.

The following examples demonstrate how to use these annotations. See the Java docs for each type for more detailed information.

```
…
@WsManagementDefaultAddressingModelAnnotation(
   getDefaultAddressDefinition=
@WsManagementAddressDetailsAnnotation(
   wsaTo="http://localhost:8080/wsman/",
   wsmanResourceURI="wsman:employee"),
   resourceMetaDataUID = "uuid:TRANS-84763554ao2q")
…
```

```
…
@WsManagementEnumerationAnnotation(
   getDefaultAddressModelDefinition=
@WsManagementDefaultAddressingModelAnnotation(
   getDefaultAddressDefinition=
@WsManagementAddressDetailsAnnotation(
   wsaTo="http://localhost:8080/wsman/",
   wsmanResourceURI="wsman:enumer"),
   resourceMetaDataUID = "uuid:9476309284763554ao2q"),
   resourceEnumerationAccessRecipe = "Use Get with lastname='srcParam ' "',
   resourceFilterUsageDescription = "4 Best results filter on hireDate")
…
```

For each resource handler class that is annotated, add an entry to the `metadata-handlers.properties` file. For example:

```
annotated.handler.1=com.sun.ws.management.server.handler.wsman.auth.user_Handler
annotated.handler.2=com.sun.ws.management.server.handler.wsman.auth.create_Handler
annotated.handler.3=com.sun.ws.management.server.handler.wsman.auth.user_Handler
```

Consuming Annotations

To consume annotations:

1. Add the Metadata location in the `metadata.properties` file.

   All Wiseman implementations should implement and facilitate client calls for Identify. Wiseman returns a few additional Identify elements that describe how to access the metadata resource. These properties are configurable in the `metadata.properties` file using the following properties (note escaped characters ':' should be handled by default):

   ```
   Metadata.refresh.interval=900000
   ```

```
metadata.refresh.enabled=true
metadata.index.to=http\://localhost\:8080/wsman/
metadata.index.enabled=true
metadata.index.resourceURI=wsman\:metadata
metadata.index.description=The meta-data endpoint/index provides information to access
resources on this server. Submit Transfer GET request to meta-data-to and
meta-data-resourceuri
```

The following example shows a typical identify response.

```
<id:IdentifyResponse>
 <id:ProductVendor>The Wiseman Project - https://wiseman.dev.java.net</id:ProductVendor>
 <id:ProductVersion>0.6</id:ProductVersion>
 <id:ProtocolVersion>http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd</id:ProtocolVersion>
 <id:SpecVersion>1.0.0a</id:SpecVersion>
 <id:BuildId>200702222058</id:BuildId>
 <wsmeta:MetaDataDescription xmlns:wsmeta="(ns not shown for brevity)">The meta-data endpoint/index
       provides information to access resources on this server. Submit Transfer GET request to
       meta-data-to and meta-data-resourceuri
 </wsmeta:MetaDataDescription>
 <wsmeta:MetaDataResourceURI xmlns:wsmeta="(ns not shown for brevity)">wsman:metadata</wsmeta:MetaDataResourceURI>
 <wsmeta:MetaDataEnabled xmlns:wsmeta="(ns not shown for brevity)">true</wsmeta:MetaDataEnabled>
 <wsmeta:MetaDataTo xmlns:wsmeta="(ns not shown for brevity)">http://localhost:8080/wsman/</wsmeta:MetaDataTo>
</id:IdentifyResponse>
```

2 Consume the initial metadata address.

Use the following pattern to submit the request:

```
//Request identify info to get MetaData root information
final Identify identify = new Identify();
identify.setIdentify();

//Send identify request
final Addressing response =  HttpClient.sendRequest(
   identify.getMessage(), DESTINATION);
```

Use the following pattern to extract Identify information.

```
SOAPElement el = IdentifyUtility.locateElement(
   id, AnnotationProcessor.META_DATA_RESOURCE_URI);

//retrieve the MetaData ResourceURI
String resUri=el.getTextContent();
```

3 Build/send the metadata request message.

```
//Build the GET request to be submitted for the metadata
Management m = TransferUtility.createMessage(
   metTo, resUri, Transfer.GET_ACTION_URI, null, null, 30000, null);

//Parse the getResponse for the MetaData
final Addressing getResponse = HttpClient.sendRequest(m);
```

Use the `ManagementUtility` class to convert the `MetadataExchange` request to familiar Management instances.

```
//retrieve all the metadata descriptions
Management[] metaDataList = ManagementUtility.extractEmbeddedMetaDataElements(
   getResponse);
```

4 Consume the messages. The metadata elements returned are Management instances so one can simply change the relevant remaining fields and begin requests to those servers.

The above steps are performed for you automatically by the Wiseman API call available from the `com.sun.ws.management.metadata.annotations.AnnotationProcessor` class.:

```
public static Management findAnnotatedResourceByUID(String metaUidForAnnotatedResource,
   String wisemanServer);
```

Using the Metadata Viewer

The `com.sun.ws.management.metadata.viewer.MetadataViewer` class has a `main()` method and can be run standalone. A batch file (`mViewer.bat`) and shell script (`mViewer.sh`), located in the `WISEMAN_HOME\tools` directory, can be used to start the Metadata Viewer.

The tool makes an HTTP connection to a Wiseman enabled Web server and requires the following system properties to be defined as command line parameter or set within the GUI tool before clicking the **Load** button:

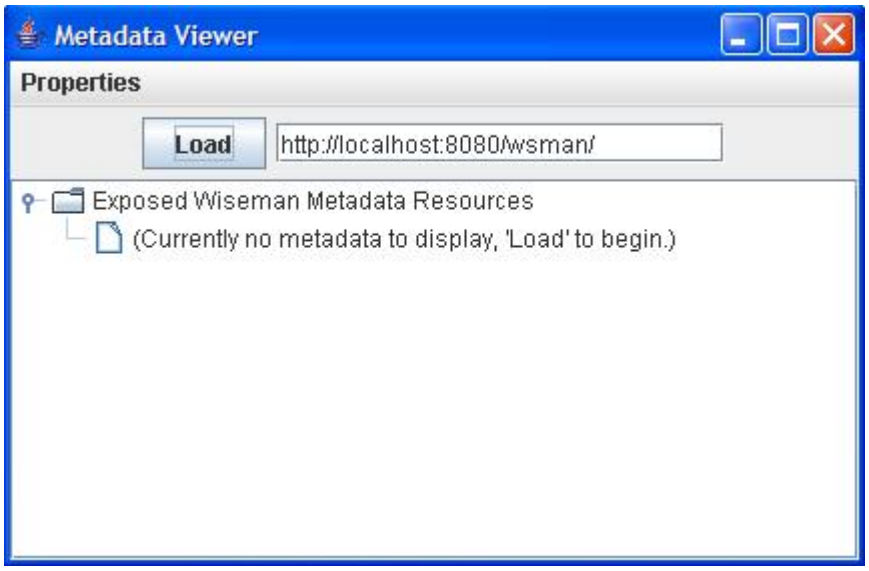
```
-Dwsman.basicauthentication=true
```

```
-Dwsman.user=wsman
```

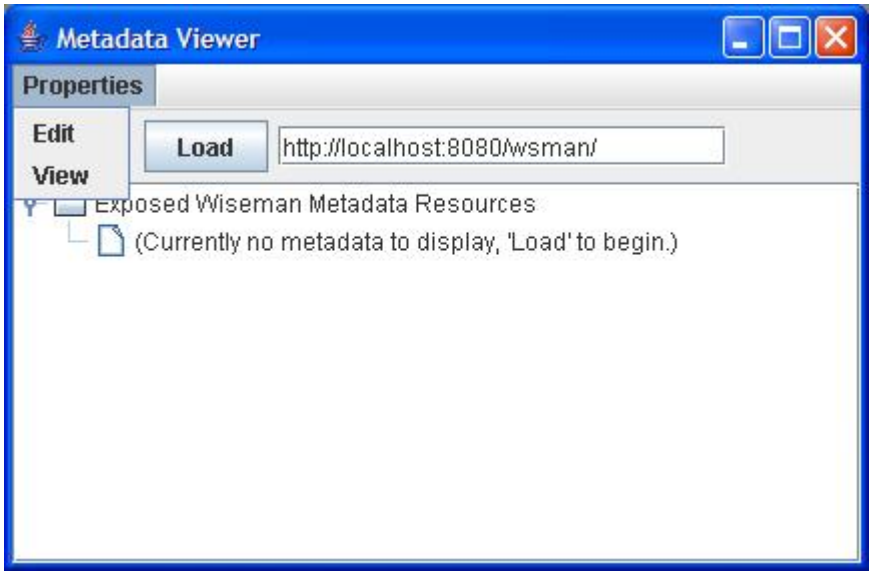
```
-Dwsman.password=secret
```

The following screen shots provide a brief overview of the tool.

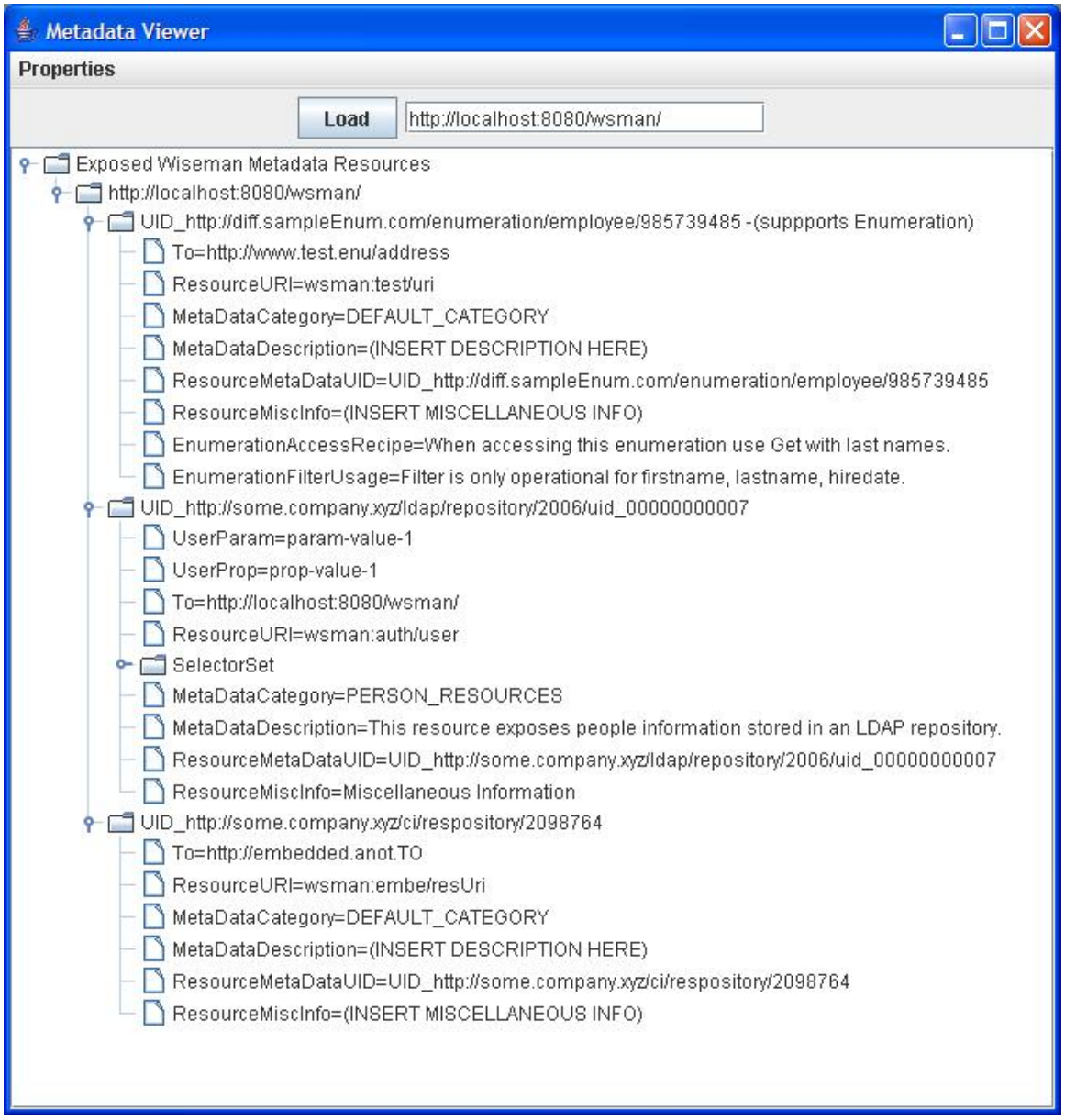
Unpopulated View



Properties editor view to set necessary parameters if not passed in as –D values to VM



Loaded metadata viewer with test annotation data



Best Practices

The following list provides some best practices when providing metadata:

- Be sure to define a unique `ResourceMetaDataUID` for each annotation that you define. This is useful for programmatically locating metadata information instances.
- Be succinct filling out annotation details.
- Fill out all of the annotation details with as much unique information as possible.

7 Implementing Eventing

This chapter provides an introduction to WS-Eventing and provides two approaches for implementing eventing. Eventing is considered an advanced implementation topic of Wiseman. Readers should be familiar with the content presented in previous chapters before using this chapter to build an eventing implementation.

The following sections are included in this chapter:

Overview

The WS-Eventing specification defines a set of operations that are used to support eventing in Web services. When using the Wiseman tools, these operations are automatically defined in the resource's management WSDL.

The operations are:

- `Subscribe` – The Subscribe operation subscribes to a resource's events.
- `Renew` – The Renew operation renews a subscription prior to its expiration.
- `Unsubscribe` – The Unsubscribe operation cancels a subscription.
- `GetStatus` – The GetStatus operation requests the status of a subscription. The WS-Management specification does not recommend implementing this operation; however, it is implemented as part of a Wiseman-generated WSDL, but not supported by Wiseman's eventing support class.

The WS-Eventing specification defines a SubscriptionEnd operation that delivers a message to indicate that a subscription has terminated. The WS-Management specification makes this an optional operation. This operation is not implemented in a Wiseman-generated WSDL.

The Wiseman tools and framework provide all the JAXB types for all of the Eventing elements that are required to build an implementation consistent with the WS-Eventing specification. An eventing implementation can be simplistic or advanced depending on a resource model's specific eventing requirements.

When adding Eventing functionality to a WS-Management resource using Wiseman there are many implementation options that are available, since the goal of the WS-Eventing specification is to provide extensibility for more sophisticated and/or currently unanticipated subscription scenarios.

This chapter discusses two approaches for creating an eventing implementation: Using the `EventSupport` class and a Metadata approach. Both approaches utilize the Wiseman API. A brief conceptual architecture is first provided for those new to WS-Eventing.

Conceptual Architecture of WS-Eventing

There are four WS-Eventing entities that are associated with a basic eventing implementation. These entities are defined below and shown in Figure 3.

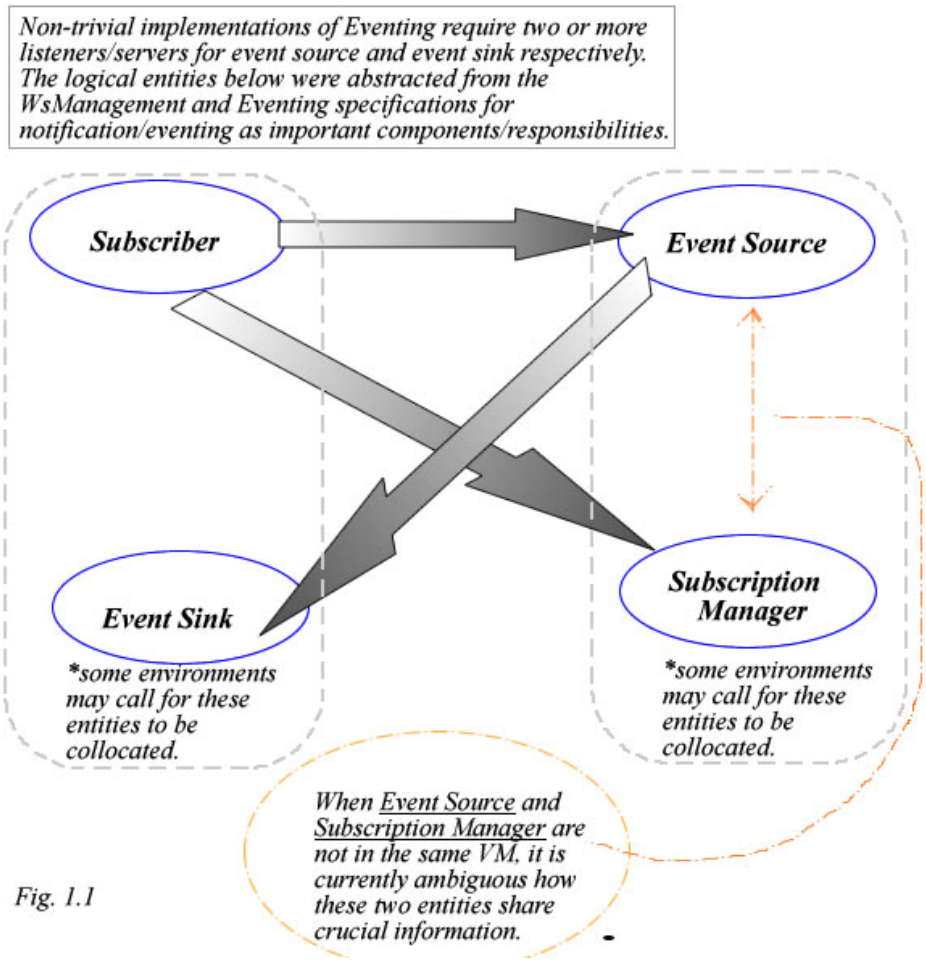
- **Subscriber**: A subscriber initiates a subscription. A subscriber can be abstracted down to the message initiating a subscription or a filtered subscription. The source of the subscribe message is irrelevant as all necessary information for action on the subscription should be included in the message itself.
- **Event Source**: An event source is responsible for receiving the subscribe requests and for generating appropriate response events.
- **Subscription Manager**: A subscription manager is used to manage subscriptions. Specifically, the subscription manager receives Renew, GetStatus and Unsubscribe messages. The subscription manager may be a separate entity or can be implemented within an Event Source. The two can be collocated, geographically dispersed across different JVMs.

  A clear need should be established to justify separating the event source and the subscription manager. Event implementations that separate the subscription manager can become very complex. Carefully consider the implications of this design choice. For example, the information received in a subscribe request needs to be added/forwarded from the event source to the subscription manager at the time that the subscription is defined. Also, every event notification increases transmission lag since the subscription manager must be contacted by the event source to either emit events or to retrieve the latest valid subscriber list. Even if the event source stores/caches events and the subscriber list, the subscription manager still needs to communicate with the event source running to determine whether Renew/Unsubscribe requests still have relevant subscriptions to apply these operations to.

- **Event Sink**: An event sink is an EPR/address where events need to be sent to when they are emitted. Subscribers register an event sink address at subscription time specifying where event notifications are to be delivered.

Figure 3 Conceptual View of Eventing



Using the Wiseman API to Implement Eventing

The following example demonstrates how to use the Wiseman APIs to accept and manage WS-Eventing subscriptions. The `EventingSupport` class has been provided to assist in this effort. This class provides the following functionality:

- Accept subscription requests and generate subscription response messages.
- Maintains a list of outstanding subscriptions.
- Automatically handles subscription deletion/timeout.
  - Notifies server application for a subscription deletion.
  - Notifies subscriber in case of timeout.
- Handles delivery of both synchronous and asynchronous messages.

The Wiseman `EventingSupport` class does not currently support the `GetStatus` and `Renew` operations.

A resource's handler class can be used to call the `subscribe()` method to take advantage of subscription management provided by the `EventingSupport` class. The method is called when a subscription request arrives from a subscriber. For example:

```
public void SubscribeOp(HandlerContext context, Eventing request, Eventing response) {

   // Call EventingSupport.subscribe() to register the subscription
   // UUID returned identifies the subscription
   // Response is automatically set by subscribe()
   final UUID subscription = EventingSupport.subscribe(context,
                                                       request,
                                                       response,
                                                       myContextListener);
   …
}
```

Note that when the subscription is created/deleted, the context listener (`myContextListener`) is called and given the UUID of the subscription context. This UUID is needed for identifying the subscription. The following is an example of the methods a context listener would implement:

```
public void contextBound(HandlerContext requestContext, UUID context) {
   // Add the context to our list of subscribers
   synchronized(subscriptions) {
      subscriptions.add(context);
   }
}

public void contextUnbound(HandlerContext requestContext, UUID context) {
   // Remove the context from our list of subscribers
   synchronized(subscriptions) {
      subscriptions.remove(context);
   }
}
```

When an event occurs, the server application sends the event to the subscriber by calling one of the `EnumerationSupport sendEvent()` methods or by creating a message and sending it to the subscriber itself. The application needs to call `sendEvent()` for each subscriber. For example:

```
final class myContextListener implements ContextListener {
   private final ArrayList<UUID> subscriptions = new ArrayList<UUID>();

  …

   // Method to handle sending events to subscriber
   // This should be called by the server thread detecting the event.
   void event(JAXBElement event) {
      synchronized (subscriptions) {
         UUID id;
         final Iterator iter = this.subscriptions.iterator();

         // Send an event to each subscriber
         while (iter.hasNext()) {
            id = (UUID) iter.next();
            EventingSupport.sendEvent(id, event);
         }
      }
   }
}
```

When a subscriber sends an unsubscribe message, the following sample code handles the unsubscribe operation.

```
public void UnsubscribeOp(HandlerContext context, Eventing request, Eventing response) {

   // Call EventingSupport.unsubscribe() to unsubscribe
   // UUID returned identifies the subscription
    // Response is automatically set by unsubscribe()
    // ListenerContext used at subscribe is called to notify service
    final boolean success = EventingSupport.unsubscribe(context,
                                                        request,
                                                        response);
    …
}
```

Subscriptions for WS-Management applications can utilize either a push or pull delivery mode. The `EnumerationSupport` class provides support for both. To enable pull mode, the server application can register an `EnumerationIterator` with `EventingSupport` to be used for handling the pull calls from the subscriber. If one is not registered, a generic `Iterator` is used to cache the events until the subscriber requests them. To register an `Iterator`, call use the `EventingSupport.registerIteratorFactory()` method.

The default iterator's cache size can be controlled while subscribing. When an event is accepted by `EventingSupport.sendEvent()` for a pull subscriber, it is added to the iterator cache. If the cache is full, the event at the top of the cache is deleted and the new event is added at the end of the cache. When the subscriber pulls for the context, the events are returned to the subscriber and deleted from the iterator's cache. The following shows an example of how to handle subscriptions for both pull and push delivery modes.

The buffer size is ignored for push subscriptions and resources that have registered their own iterator.

```
public void SubscribeOp(HandlerContext context, Eventing request, Eventing response) {

   // Indicated that the Iterator should filter the requests
   final boolean isPreFiltered = false;

   // Call EventingSupport.subscribe() to register the subscription
   // UUID returned identifies the subscription
   // Response is automatically set by subscribe()
   final UUID subscription = EventingSupport.subscribe(context,
                                                       request,
                                                       response,
                                                       isPreFiltered,
                                                       bufferSize,
                                                       myContextListener);
   …
}
```

At this time, the `EventingSupport.sendEvent()` method uses the Wiseman static `HttpClient` to send push events to the subscriber. This can cause problems with subscriber sinks that require authentication. In order for this to work properly, the server application needs to synchronize its use of the `HttpClient` within the WAR file. For example:

```
final class myContextListener implements ContextListener {
   private final ArrayList<UUID> subscriptions = new ArrayList<UUID>();
   private final Map<UUID, Authenticator > credentials = new HashMap<UUID,
      Authenticator>();

   …

   // Method to handle sending events to subscriber
```

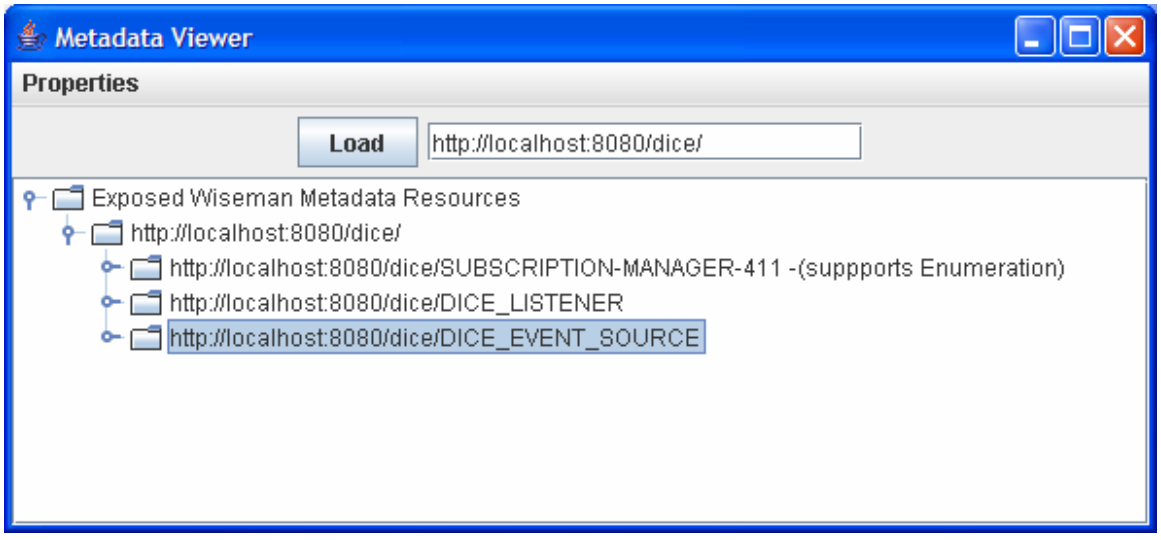
```
    // This should be called by the server thread detecting the event.
    void event(JAXBElement event) {
       synchronized (subscriptions) {
          UUID id;
          final Iterator iter = this.subscriptions.iterator();

          // Send an event to each subscriber
          while (iter.hasNext()) {
             id = (UUID) iter.next();
             HttpClient.setAuthenticator((Authenticator)credentials.get(id));
             EventingSupport.sendEvent(id, event);
          }
       }
    }
}
```

Implementing Eventing with Metadata

The Wiseman Eventing API can be used together with the Wiseman Metadata feature to create a simple eventing implementation. In this approach, the subscription manager and the event source are implemented as two separate Web services. The Wiseman Metadata mechanism and the WS-Management protocol (WS-Transfer and WS-Enumeration) are used to implement the communication protocol between the two services. For more information about the Wiseman Metadata feature, see chapter 6 “Providing Metadata”.

The sample demonstrates how a single subscription manager implementation can be used by many different varied event sources and can simultaneously serve as an enumeration of available event sources. The example is simplistic but does demonstrate eventing functionality to facilitate model specific implementations of Eventing using Wiseman.

The sample code is taken from `WISEMAN_HOME/samples/template/dice_eventing_remote_subscripton_manager_sample.jar`. The dice sample uses three separate Wiseman Resources representing the subscription manager, event sink, and event source respectively.



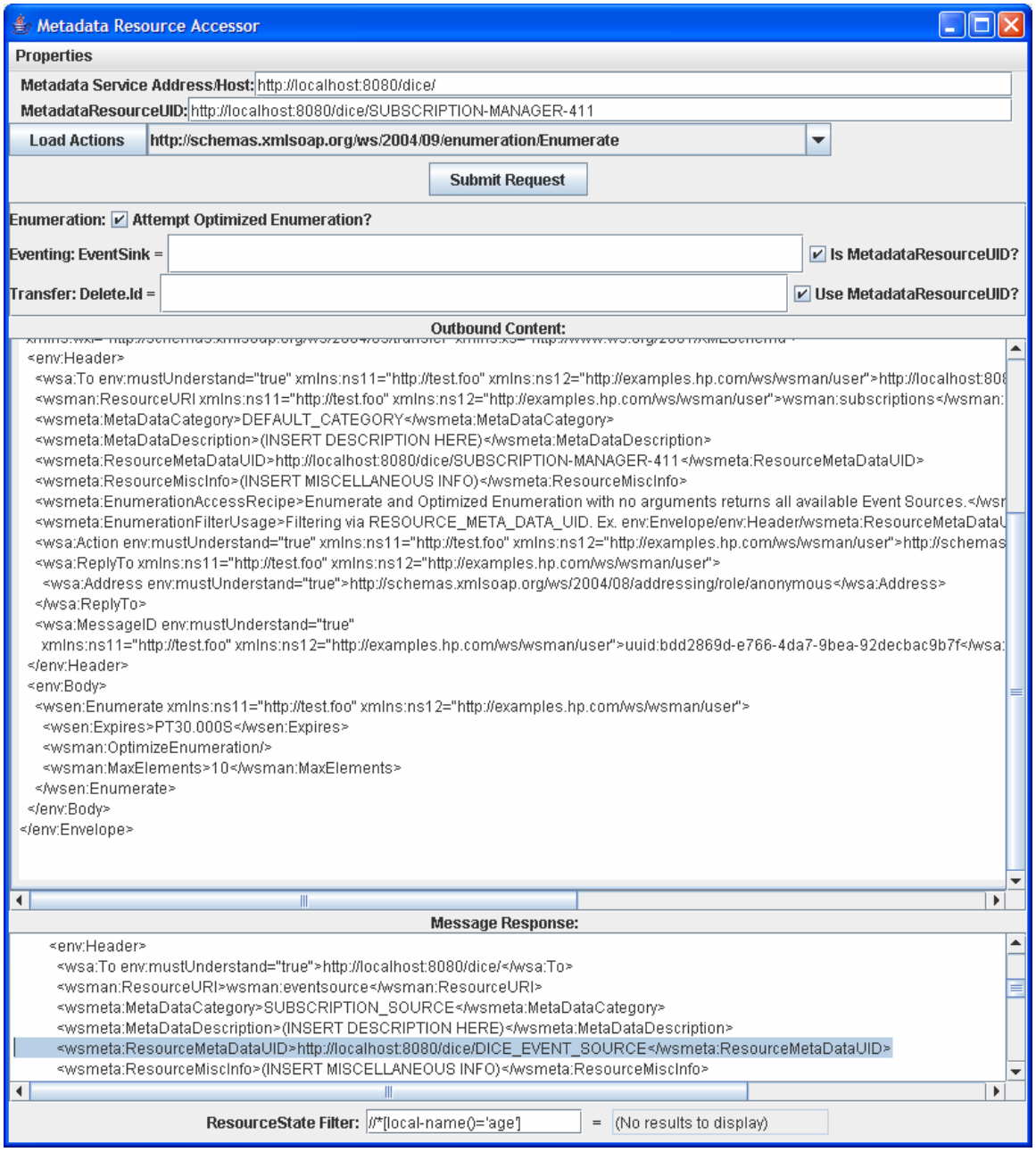
The `SubscriptionManager` class implements `com.sun.ws.management.framework.eventing.SubscriptionManagerInterface`. The `EventSource` class implements

com.sun.ws.management.framework.eventing.EventSourceInterface. Both interfaces demonstrate design choices that can be used when implementing eventing.

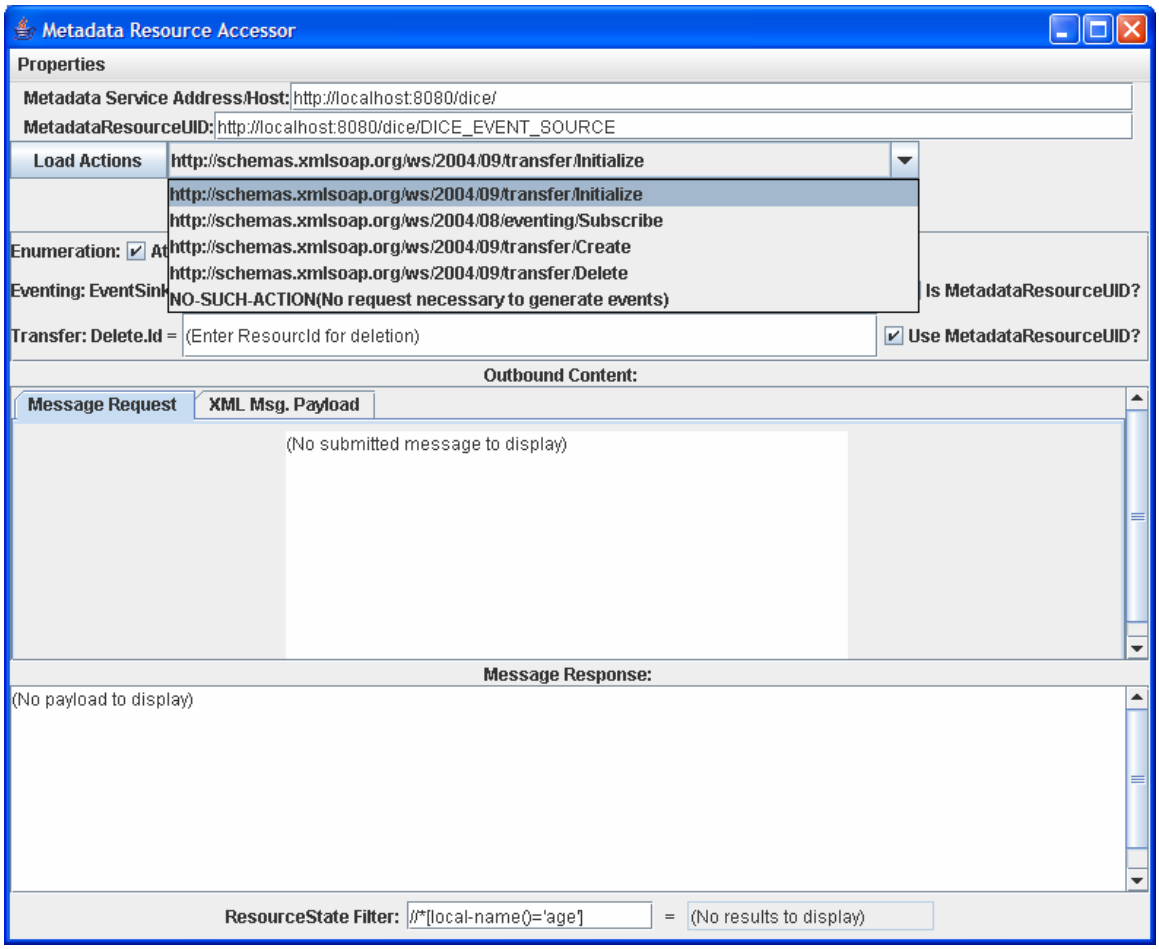
```
 * must support SUBSCRIBE.  How an Event Source and it's
 * associated Subscription Manager communicate crucial information
 * when they are VM different, is currently ambiguously defined by
 * the specifications.
 *
 * Provide implementations for:
 *   isAlsoTheSubscriptionManager()
 *   getSubscriptionManager()
 *
 * @author Simeon
 */
public interface EventSourceInterface {

    public Management subscribe(HandlerContext context, Management eventRequest,
            Management eventResponse) throws SOAPException, JAXBException,
            DatatypeConfigurationException, IOException;

    /* Implementors may optionally support this item to allow other entities
     * the ability to create events upon request from this entity.
     */
    public void create(HandlerContext context, Management request,
            Management response) throws Exception;

    //Subscription Manager interaction details.
    /** Flag indicating whether this EventSource is also the
     *  SubscriptionManager instance as well. No soap message
     *  communication with SubscriptionManager instance necessary.
     *
     * @return boolean flag indicating the above status.
     */
    boolean isAlsoTheSubscriptionManager();

    public Management getMetadataForEventSource() throws SOAPException,
    JAXBException, DatatypeConfigurationException, IOException;
    public Management getMetadataForSubscriptionManager() throws SOAPException,
    JAXBException, DatatypeConfigurationException, IOException;
    public void setRemoteSubscriptionManager(Management subscriptionManagerMetaData);
```

A metadata aware tool (Metadata Resource Accessor) is used to show how these elements interact. In the sample, the `DICE_EVENT_SOURCE` has set its `Metadata Category` to `EVENT_SOURCE` and automatically displays as an available event source when an enumerate message is sent to the SUBSCRIPTION-MANAGER resource. As the SubscriptionManager stores a record of all subscriptions to unsubscribe from, it is helpful if it can expose the list of available event sources that a subscriber can register/unregister with/from.

The following example shows enumerate request/response and lists available event sources



With this located `EventSource` described by the `MetadataResourceUID`, the available actions off of that resource can be populated as shown below, after specifying the Metadata Service Host and the Metadata identifier for the event source, in a drop down list.

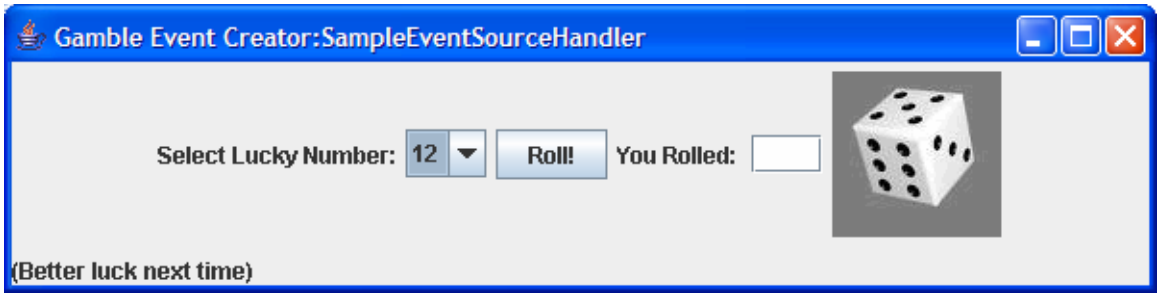


A custom action for initializing the event source before it is able to accept subscription requests is implemented. Because the `DICE_EVENT_SOURCE` and `SUBSCRIPTION_MANAGER-411` both implement the `EventSourceInterface` and the `SubscriptionManagerInterfaces`, respectively, the `MetadataUtility.registerEventSourceWithSubscriptionManager` method can be used to hide the communication details of linking the two Web services.

The following example demonstrates event source initialization where `EventSink` is registered with `SubscriptionManager`.

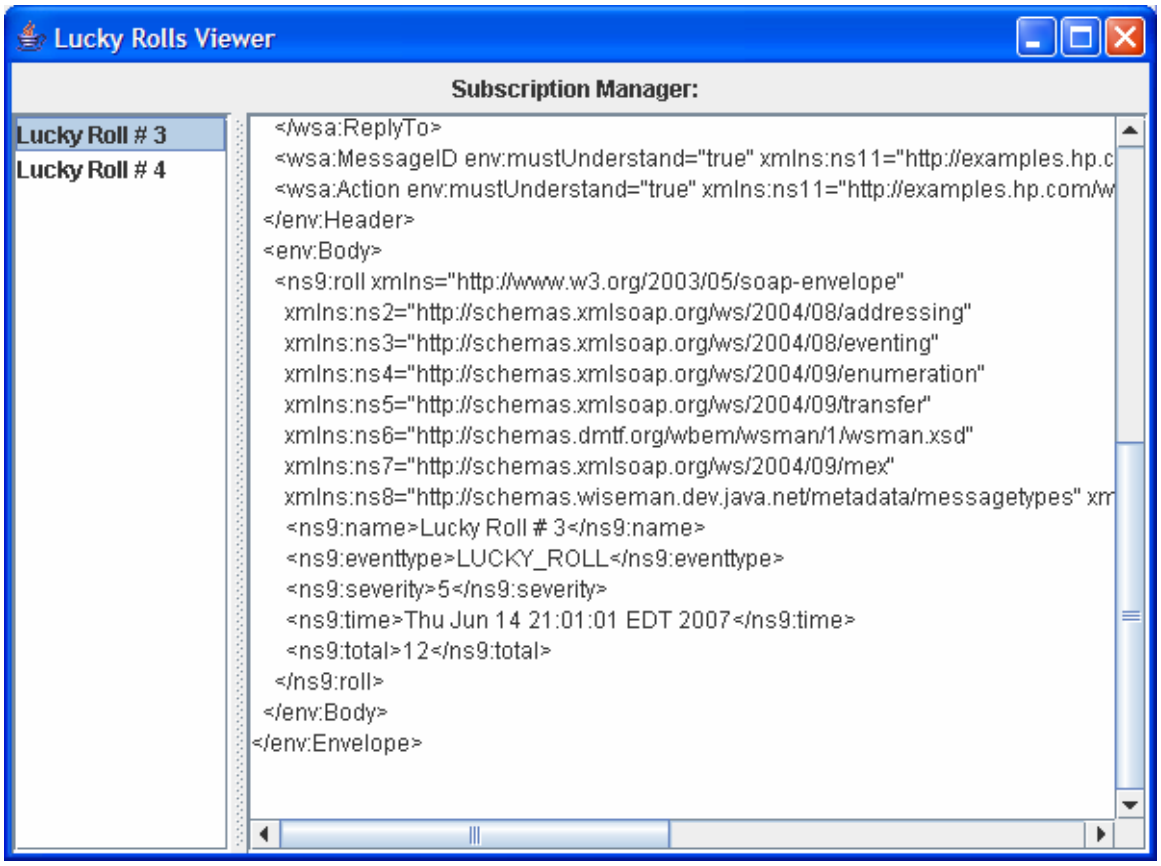
```
/**This method should be implemented to lazily instantiate.  This method should
 * only be executed once.  This method is meant to be used for initialization
 * tasks that:
 * -could require SOAP communication with other handlers
 * -could require expensive database initialization
 *
 * @param context
 * @param request
 * @param response
 * @throws IOException
 * @throws DatatypeConfigurationException
 * @throws JAXBException
 * @throws SOAPException
 */
public Management initialize(HandlerContext context, Management request,
        Management response) throws SOAPException, JAXBException,
        DatatypeConfigurationException, IOException {
    if(subscription_source_id==null){
        //Then must locate the subscription manager details
        //Send create request for EVENT_SOURCE to register with SUB_MAN
        //Extract the created id and initalize the sub_source reference
        //?? Wrap in it's own thread?
        String srcId = MetadataUtility.registerEventSourceWithSubscriptionManager(
                this, true, true);
        subscription_source_id = srcId;
        response.setAction(Metadata.INITIALIZE_RESPONSE_URI);
        response.setMessageId(ManagementMessageValues.DEFAULT_UID_SCHEME+UUID.randomUUID());
        response.setReplyTo(Addressing.ANONYMOUS_ENDPOINT_URI);
        if((drGui==null)||(!drGui.isDisplayable())){
          drGui = new DiceEvents();
        }
```

After the initialization step is completed, the following graphical event creator instance that is linked to the backend `DICE_EVENT_SOURCE` resource is now ready to emit events.



The application is a graphical way for users to roll dice until they roll a combination that matches the lucky number displayed on the left. When this occurs, a WS-Management event is emitted to all subscribers. As no-one has yet subscribed to this Event Source, no actual notifications are generated.

Using the Metadata Resource Accessor, a subscribe request to the event source with the `DICE_LISTENER` as the event sink. Upon the first event sent to the registered subscriber, another simple graphical user interface showing the successfully received events is displayed.



The `SubscriptionManager` implementation also exposes the `Unsubscribe` operation. As shown above, the Metadata Resource Accessor is used to then submit an unsubscribe request to the `SubscriptionManager` and the notifications stop.

8 Deploying Management Web Services

This chapter provides instructions for packaging and deploying a resource's management Web service. To deploy the management Web service, a Web container (such as Tomcat) is required.

The following sections are included in this chapter:

Overview

Wiseman-based WS-Management Web services are packaged as a Web application and deployed to a Java EE-compliant Web container. The Web application includes all the libraries that are required for a complete Web services runtime environment. Wiseman is designed to leverage the Java EE Web container platform and subsequently fits seamlessly into any Java EE application server. Developers can choose to deploy the Wiseman WAR into any Java EE application server and leverage any functionality provided by the server (for example, transactioning, pooling, etc…).

Currently, an application server's native Web services runtime environment cannot be used to host Wiseman-based WS-Management Web services. This includes using Axis.

For convenience, the Wiseman template Ant build script can be used to package and deploy a resource's WS-Management Web service. However, this chapter also provides instructions for manually packaging and deploying a resource's WS-Management Web service.

Using the Template Ant Script

The instructions assume that the steps in Chapter 3, "Using the Template Ant Script," were used to create a working directory and generate source files for a resource.

To use the template Ant script:

1. From a command prompt, change directories to `WORK_DIR`.
2. Issue the following command:

   `ant`

   A WAR is created and written to the `WORK_DIR/dist` directory.
3. Deploy to the WAR to your Java EE Web container.
4. Using a Web browser, verify that the deployment succeeded by accessing the WS-Management WSDL for the resource. For example:

   `http://<host>:<port>/<application_context>/wsdls/<wsdl_name>.wsdl`

Replace <*host*>:<*port*> with the host name and port number for the Web container where the implementation is deployed. Replace <*application_context*> with the application context used to access the Wiseman Web application. Lastly, replace <*wsdl_name*> with the name of the WSDL.

The application context and WSDL name used in an implementation can be found in the `WORK_DIR/project.properties`.

The `web.xml` file defines security constraints for the `wsman` role. Either modify your application server's roles to include this role, or modify the `web.xml` with a valid role already defined on the application server.

Manually Packaging and Deploying Web Services

This section describes how to manually package and deploy a resource's management Web service. The instructions assume that the steps in Chapter 3, "Using the Template Ant Script," were used to create a working directory and generate source files for a resource.

To manually package and deploy management Web services:

1. Create a Web application directory structure. For example:

   ```
   .
   ./WEB-INF
   ./WEB-INF/lib
   ./WEB-INF/classes
   ```

2. Copy the following libraries from `WISEMAN_HOME/lib` to the Web application's `WEB-INF/lib` directory.
   - wiseman.jar
   - velocity-1.4.jar
   - velocity-dep-1.4.jar
   - jaxws/activation.jar
   - jaxws/jaxb-api.jar
   - jaxws/jaxb-impl.jar
   - jaxws/jsr173_api.jar
   - jaxws/saaj-api.jar
   - jaxws/saaj-impl.jar
3. From `WORK_DIR/gen-src`, compile the resource's implementation classes and copy them (including the full package structure) to the Web application's `WEB-INF/classes` directory.
4. Copy any supporting libraries that are required by the resource's implementation classes to the Web application's `WEB-INF/lib` directory.
5. From `WORK_DIR`, copy the `xsd`, `wsdls`, and `schemas` directory to the Web application's root directory.
6. From `WISEMAN_HOME/etc`, copy `web.xml` to the Web application's `WEB-INF` directory.

The `web.xml` file defines security constraints for the `wsman` role. Either modify your application server's roles to include this role, or modify the `web.xml` with a valid role already defined on the application server.

7 JAR or ZIP the Web application directory and change the extension to `.war`.

8 Deploy the WAR to an application server's Web container.

9 Using a Web browser, verify that the deployment succeeded by accessing the WS-Management WSDL for the resource. For example:

`http://<host>:<port>/<context>/wsdls/<wsdl_name>.wsdl`

9 Developer Tips

This chapter provides several tips that can be used when working with Wiseman.

The following sections are included in this chapter:

XML Schema Definition

The following mechanism defines a complex type `String` node that contains text and also has attributes.

```
<xs:complexType name="SchemaType">
   <xs:simpleContent>
      <xs:extension base="xs:string">
         <xs:attribute name="prefix" type="xs:string" />
      </xs:extension>
   </xs:simpleContent>
</xs:complexType>
```

To create the following XML node:

```
<ns8:schema prefix="tl">http://schemas.wiseman.dev.java.net/traffic/1/light.xsd
</ns8:schema>
```

JAXB Tips

Problems Unmarshalling XML Content

The following problem often occurs when unmarshalling XML content to schema defined types:

```
"unable to marshal type "org.blah.MessageType" as an element
because it is missing an @XmlRootElement annotation"
```

This issue arises because JAXB encountered a type that may be used as a nested child element. JAXB cannot make the assumption that the element will be a root node.

Solution:

To get JAXB to automatically apply the `XMLRootElement` annotation, redefine:

```
<xs:complexType name="MessageDefinitionType">
   <xs:sequence>
      <xs:element name="schemas" type="mdo:SchemasType"/>
      <xs:element name="operations" type="mdo:OperationsType" />
```

```
    </xs:sequence>
 </xs:complexType>
 <xs:element name="message-definitions" type="mdo:MessageDefinitionType"/>
```

as

```
 <xs:element name="message-definitions">
    <xs:complexType>
       <xs:sequence>
          <xs:element name="schemas" type="mdo:SchemasType"/>
          <xs:element name="operations" type="mdo:OperationsType" />
       </xs:sequence>
    </xs:complexType>
 </xs:element>
```

See the following link for more detailed information:

http://weblogs.java.net/blog/kohsuke/archive/2006/03/why_does_jaxb_p.html

Miscellaneous

When troubleshooting an unmarshalling problem, it is often good to verify that the `org.w3c.dom.Node` or `Document` that you think that you have is indeed the node that you expect. The following code is useful in JDK 5 code to quickly take a Node/Document and convert this DOM to a text readable string.

```
public static String domToString(Node node) {
   try {
      Source source = new DOMSource(node);
      StringWriter stringWriter = new StringWriter();
      Result result = new StreamResult(stringWriter);
      TransformerFactory factory = TransformerFactory.newInstance();
      Transformer transformer = factory.newTransformer();
      transformer.transform(source, result);

      return stringWriter.getBuffer().toString();
   } catch (TransformerConfigurationException e) {
      e.printStackTrace();
   } catch (TransformerException e) {
      e.printStackTrace();
   }
   return null;
}
```

WS-Management Schemas and WSDLs

In order to provide interoperability with various other 3rd party client SOAP toolkits, such as Axis and JAX-WS, Wiseman provides its own modified versions of the WS-management schema and WSDL documents. These files are located in the `<webapp>/schemas/wiseman` and `<webapp>/wsdls/wiseman` directories. These modified files are strongly recommended when a Web service interface is consumed by these third-party client toolkits. These files are automatically placed in appropriate directory when using the Wiseman project template.

Client Utility Classes

Several client utility classes have been provided to assist Wiseman client developers. There are MessageValues classes and MessageUtility classes for each of the main Wiseman functional areas (Transfer, Eventing, Enumeration). For more information on developing clients, see the *Wiseman Client Developer's Guide* located in the `WISEMAN_HOME/docs` directory.

The MessageValue classes provide a way to group together the values of the various parameters to the SOAP calls. The MessageUtility classes produce the correct SOAP message based on the values set in the supplied MessageValue class. For example:

```
protected Management sendGetRequest(String destination, String resourceUri,
Set<SelectorType> selectors, Set<OptionType> options) throws SOAPException,
JAXBException, DatatypeConfigurationException, IOException
{

   // Build Request Document
   TransferMessageValues settings =  TransferMessageValues.newInstance();
   settings.setResourceUri(resourceUri);
   settings.setTo(destination);
   settings.setSelectorSet(selectors);
   settings.setTransferMessageActionType(Transfer.GET_ACTION_URI);

   if (options != null) {
      settings.setOptionSet(options);
   }
   final Transfer mgmt = TransferUtility.buildMessage(null, settings);

   // Send the get request to the server
   Addressing response = HttpClient.sendRequest(mgmt);

      // Look for returned faults
      if (response.getBody().hasFault())
      {
         SOAPFault fault = response.getBody().getFault();
         throw new SOAPException(fault.getFaultString());
      }

   return new Management(response);
}
```

Third-Party Client Developer Utilities

For users that prefer to use third-party SOAP toolkits to develop their client code, we have provided two utility classes to assist with the Wiseman method calls – WsmanAxisUtils and WsmanJaxwsUtils. These classes can be used to add the required WS-Managment request headers to the Wiseman method calls. Two sample classes that use these utility classes have also been included. The samples can be found in `WISEMAN_HOME/docs/third-party-client-samples.zip`.

For more information on developing clients, see the *Wiseman Client Developer's Guide* located in the `WISEMAN_HOME/docs` directory.

Index

I

J

L

M

N

O

P

R

S

T

U

V

W

X