

# Wiseman

Software Version: 1.0

## Client Developer's Guide

---

Document Release Date: June 2007

Software Release Date: June 2007

## Copyright Notices

Copyright © 2007 Hewlett-Packard Development Company, L.P.

## Documentation Updates

**Table 1      Document Changes**

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

# Contents

<b>1</b>	<b>Introduction .....</b>	<b>5</b>
	Client Toolkit Overview .....	5
	Building WS-Management Clients: Overview .....	5
	Using the WSDL Approach .....	6
	Using the Metadata Approach .....	6
	Conceptual View of the Client/Server Architecture .....	7
	Client Side .....	7
	Server Side.....	8
	Understanding the Client-Side Access Model .....	8
	OutBound Messages .....	8
	WS-Management Message Payloads.....	9
	InBound Message Processing.....	10
	ResourceStateDocument/ResourceState Model.....	10
	Custom XML Binding Framework Model.....	11
	Command Line Tool .....	11
	Delivery of Client Side Access Tools: .....	11
<b>2</b>	<b>Creating Clients from WSDL .....</b>	<b>13</b>
	Retrieving a Deployed WSDL .....	13
	WSDL Generation/Modifications.....	13
	Unique Request Message Definitions .....	14
	Input Message Cannot Be Empty .....	14
	Definition of Input Messages .....	15
	Schema Imports .....	15
	Process Contents Attribute .....	15
	Third-Party SOAP Client Toolkit Integration.....	16
<b>3</b>	<b>Creating Clients from Metadata .....</b>	<b>17</b>
	Overview.....	17
	Browsing a Resource's Metadata.....	18
	Creating a Request .....	18
	Processing a Response.....	19
<b>4</b>	<b>Using The Client Command Line Tool .....</b>	<b>21</b>
	<b>Index .....</b>	<b>23</b>



# 1 Introduction

This chapter provides an overview of the client model that is provided as part of Wiseman. 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. If you are new to WS-Management, see the *Wiseman Server Developer's Guide* located in the `WISEMAN_HOME/docs` directory. The guide provides introductory material that is required to understand the concepts in this guide.

The client model is broken down into two approaches: traditional WSDL and Metadata. Each approach is described in this chapter and detailed in subsequent chapters. For those new to Web services, a conceptual view of the client/server architecture used by Wiseman is provided.

The following sections are included in this chapter:

*Client Toolkit Overview*

*Building WS-Management Clients: Overview*

*Conceptual View of the Client/Server Architecture*

*Understanding the Client-Side Access Model*

## Client Toolkit Overview


The toolkit includes:

- An integration with Web services client toolkits (Axis and JAX-WS) for generating client stubs
- A set of Utility Classes for building WS-Management messages
- A Metadata mechanism for consuming messages using metadata
- A Metadata viewer for viewing a resource's metadata
- A Command Line Tool for testing deployed services

## Building WS-Management Clients: Overview

Wiseman supports two approaches for interacting with WS-Management Web services:

- WSDL-Based – Clients use a WSDL that defines the WS-Management operations and messages that a resource supports.
- Metadata-Based – Clients use metadata that is defined within a resource's handler class implementation.

- 
- The metadata-based approach is only applicable for Wiseman server implementations and is not interoperable with other WS-Management server implementations. In addition, clients that are built using the metadata approach cannot access Wiseman server implementations that only expose a WSDL. Clients that rely on metadata can only access server implementations that include metadata.

## Using the WSDL Approach

Traditional Web services are usually described using the Web Service Description Language (WSDL). The first step in accessing/consuming Web services is to run the WSDL through a WSDL consumption toolkit/API. There are several popular toolkits, including Axis, JAX-WS, and .NET. The toolkits simplify creating valid requests for a Web service.

WS-Management Web services have some unique constructs which make them a bit more difficult to use with popular toolkits. In particular, WS-Management Web services:

- are Action Dispatching
- do not define a unique message body across all wire operation calls in the specification
- define more metadata about how a specific Web services is to be accessed than is typically defined for traditional Web services

None of the toolkits can easily consume a WS-Management WSDL without requiring additional tool-specific modifications to the client code generated from these WSDLs. However, WSDLs that are created using the tools provided by Wiseman are consumable by the toolkits. Wiseman also includes samples that demonstrate how to consume these WSDLs. See Chapter 2 for complete instructions for creating clients by consuming a WSDL.

## Using the Metadata Approach

Wiseman provides the ability to annotate/decorate service implementations with the numerous SOAP headers and metadata that is consistent with the Default Addressing Model that is described in the WS-Management specification. These annotations expose all of the information necessary to connect to a WS-Management service. In actuality, the XML message envelope is returned with all of the necessary addressing WS-Management elements already populated. The service developer is responsible for adding the annotations to the server-side implementation. This includes:

- WS-Management data necessary to successfully access a resource
- A MetadataResourceUID that uniquely identifies a resource for easier retrieval
- A class to be exposed in the metadata service

The benefits of using the metadata approach include:

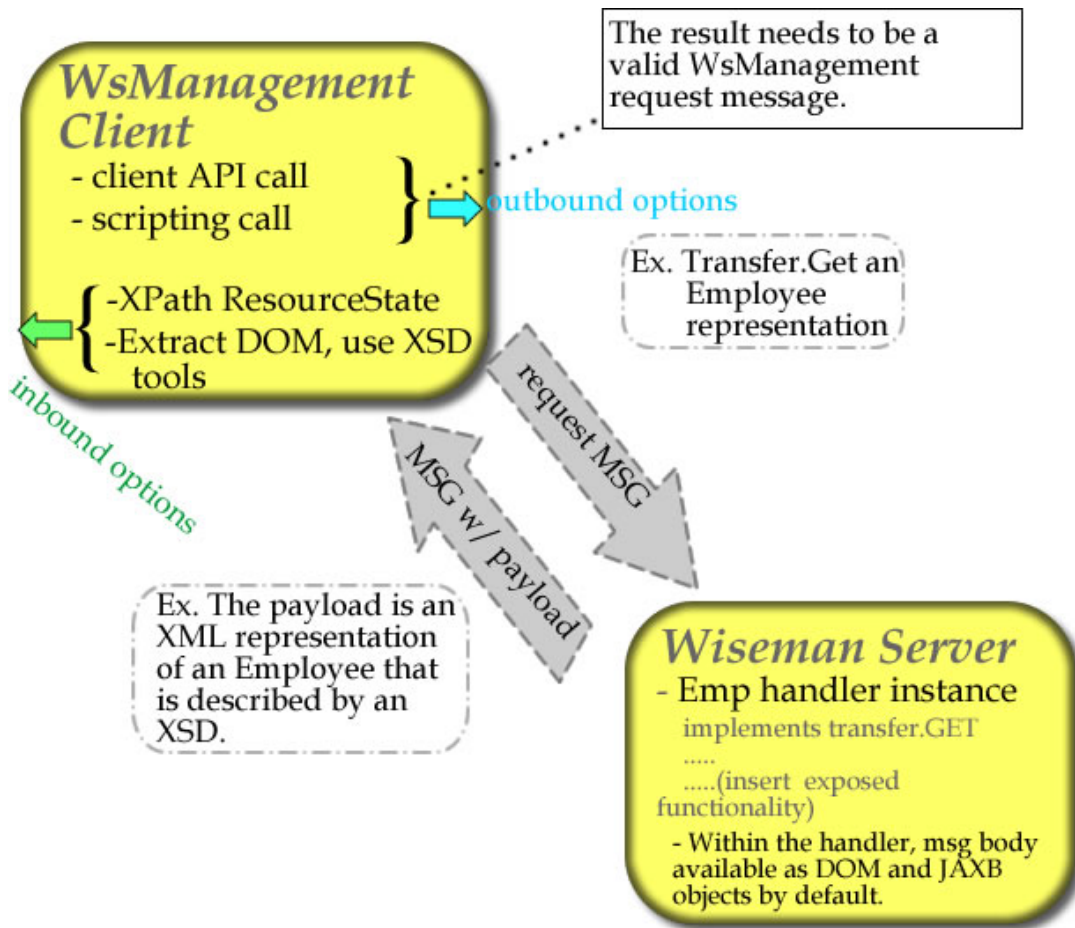
- relevant metadata about addressing for the service is exposed in a human readable fashion
- human readable descriptions about how this service should be used for best performance
- human readable descriptions of the operations available and the XSDs that describe these message payloads
- once the unique MetadataResourceUID is known, management messages, prepopulated with all the necessary connection information, are returned.

See Chapter 3 for complete instructions for creating clients by consuming Metadata.

## Conceptual View of the Client/Server Architecture

WS-Management Web services are implemented as client /server applications. The following diagram provides a conceptual view of the interaction between a client and server for a WS-Management Web service that manages an employee resource. The sample employee resource is used throughout this guide for illustrative purposes.

**Figure 1 Wiseman Client/Server Architecture**



### Client Side

The client typically uses an API call from a client library to create valid WS-Management requests. Wiseman provides several utility classes to facilitate creating WS-Management client messages. These include: the ManagementUtility class, TransferUtility class, EventingUtility class and the EnumerationUtility class. These classes are located in wiseman.jar. Server response message consumption is facilitated by integration with third-party toolkits and/or a metadata mechanism. When more in-depth content parsing is required, XSD generated artifacts for consuming these model specific response elements is encouraged.

## Server Side

The bulk of the model specific processing for each defined service is described and handled on the server. Wiseman logically delineates each service as a handler instance on the server side that optionally supports Transfer, Enumeration, Eventing, and custom operations. For more information on creating server-side implementations for resources, see the *Wiseman Server Developer's Guide* located in the `WISEMAN_HOME\docs` directory.

## Understanding the Client-Side Access Model

The client-side access model can be broken down into two parts:

- Outbound – Request messages sent from the client to the server. For example, `Transfer.Get` messages.
- Inbound – Response messages from the server to the client. For example, `Transfer.GetResponse`.

### OutBound Messages

At a high level, the WS-Management specification focuses the type and kind of outbound messages that are sent. Single resources that require Create, Get, Put, and Delete functionality use the WS-Transfer specification and the WS-Management extensions. For Multi-instance resources, Enumerations/Collections are used and defined by WS-Enumeration specification. Notifications are defined by WS-Eventing. To help with correctly generating the outbound messages from the client side, Wiseman has provided the following utility classes to ease message creation:

- `TransferUtility` and `TransferMessageValues` – Singular access methods
- `EnumerationUtility` and `EnumerationMessageValues` – Collection access
- `EventingUtility` and `EventingMessageValues` – Notification access
- `ManagementUtility` and `ManagementMessageValues` – DefaultAddressing details

The typical pattern for building a Transfer message, for example, is to create a `TransferMessageValues` instance, which is a message property container with getter and setter methods for a Transfer message.

```
TransferMessageValues settings = TransferMessageValues.newInstance();
//insert the WsManagement server endpoint details.
settings.setResourceUri("wsman:contacts");
settings.setTo("http://localhost:8080/users/");
//Specify the ActionDispatching action to be used
settings.setTransferMessageType(Transfer.GET_ACTION_URI);

//Define the instance level discriminant. AKA which specific one to get.
HashMap<String,String> contactKeys = new HashMap<String, String>();
contactKeys.put("firstname", "Bill");
contactKeys.put("lastname", "Gates");
Set<SelectorType> selectorSet =
    ManagementUtility.createSelectorType(contactKeys);
settings.setSelectorSet(selectorSet);
```



After creating the `<AccessType>MessageValues` instance with the appropriate specific details, the `MessageValues` property setup is run through the `<AccessType>Utility.buildMessage()` method which will take those elements and generate the correct message.

```
Transfer transfer = TransferUtility.buildMessage(null, settings);
```

At this point you can use the default `HttpClient` provided to send the messages off to the server. The patterns described also apply for the Enumeration and Eventing message types as well.

## WS-Management Message Payloads

However to understand what type of outbound messages the client would want to send, you inevitably need to know some context about the back end model. To bring this point home, a client may want to retrieve a book, item or employee description served up by some WS-Management server. To request any of these items, you will need to know how to correctly specify the instance-level discriminates for the server and any other special parameters that may be required. A valid request requires some amount of a priori information. Basically, the client has to know some identifying information about the book, perhaps a title, or a first and last name for a person. However, with WS-Management endpoints/services, the default addressing model described in the specification does define additional information and mechanisms for uniquely identifying the resource and additional processing/filtering that may accompany this request. A real world example of this is if you have an employee database and to access the employee data you will need to know:

- The address of server: how to locate the server on the network
- The `resourceUri` of the handler: how to locate a specific handler responsible for the service
- Any instance-level discriminates: selectors like `firstname`, `lastname` identifying specific instance that we want the resource information for.
- Filtering: whether you need to further filter all matches with an additional criteria
- WS Addressing Action: WS-Management is action-dispatching and defines this type of operation as a `Transfer.GET` so we know what the action needs to be as held in the headers of the SOAP envelope.
- Payload processing instructions: optional and helpful is the XSD that describes what the `Transfer.GetResponse` payload is going to look like. Or, use the `ResourceStateDocument` mechanism, which is an API used to retrieve a message response payload.

In Wiseman all of this information is considered useful, and in most cases required, Metadata information. There are several options to gain knowledge of the required metadata: query a resource's WSDL and use the defined endpoint reference (EPR), have preexisting knowledge of the service implementation, or have the service developer team define all or most of these details for you as metadata using Wiseman annotations. See the Wiseman Server Developer's guide, for instructions on using annotations to add metadata to a service and using the metadata viewer. Instructions for using the Metadata Viewer is also provide in Chapter 3, "Creating Clients from Metadata."

In Wiseman, metadata can be viewed graphically using the Metadata Viewer or retrieved programmatically using the provided APIs. The following example demonstrates programmatic access to metadata.

```
String serviceAddress="http://localhost:8080/users/";  
String metaDataUID=serviceAddress+"Bill.Gates";  
  
//Locate the metadata necessary to make the call
```

```

Management get = AnnotationProcessor.findAnnotatedResourceByUID(
    metaDataUID,serviceAddress);

//set Action to request of the service
get.setAction(Transfer.GET_ACTION_URI);

//run this Management instance through ManagementUtility
get = ManagementUtility.buildMessage(get, null);

//Use the default HttpClient or your own here to send the message off
Addressing response =HttpClient.sendRequest(get);

```

For those methods where you will need to build a message where you will be sending new content for the server side to process, you will usually need to involve your Java-toXML binding technology of choice to provide valid XML message content and add it to the outbound message.

## InBound Message Processing

There are three major mechanisms to access inbound messages from Wiseman services on the client side. As an example, the following sample Get Response message payload is listed is used:

```

...
</env:Header>
<env:Body>
  <ns9:user xmlns="http://www.w3.org/2003/05/soap-envelope"
    xmlns:ns2="http://schemas.xmlsoap.org/ws/2004/08/addressing"
    xmlns:ns3="http://schemas.xmlsoap.org/ws/2004/08/eventing"
    xmlns:ns4="http://schemas.xmlsoap.org/ws/2004/09/enumeration"
    xmlns:ns5="http://schemas.xmlsoap.org/ws/2004/09/transfer"
    xmlns:ns6="http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd"
    xmlns:ns7="http://schemas.xmlsoap.org/ws/2004/09/mex"
    xmlns:ns8="http://schemas.wiseman.dev.java.net/metadata/messagetypes"
    xmlns:ns9="http://examples.hp.com/ws/wsman/user">
    <ns9:firstname>Bill</ns9:firstname>
    <ns9:lastname>Gates</ns9:lastname>
    <ns9:address>149 Bilney Avenue</ns9:address>
    <ns9:city>Stanford</ns9:city>
    <ns9:state>IA</ns9:state>
    <ns9:zip>39000</ns9:zip>
    <ns9:age>34</ns9:age>
  </ns9:user>
</env:Body>
</env:Envelope>

```

## ResourceStateDocument/ResourceState Model

The ManagementUtility.getAsResourceState(Addressing) API call is used to retrieve a message response payload into a container that can be parsed by XPATH. The following example shows what a sample client get-response code and processing looks like.

```

Addressing response =HttpClient.sendRequest(get);

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

```

```
//With the content returned, convert it to ResourceStateDocument
// so that we can use XPath to process the results
ResourceStateDocument resState = ManagementUtility.getAsResourceState(response);

//We'll use the [local-name] syntax to avoid specifying namespaces
String billsAge = resState.getValueText("//*[local-name()='age']");
```

## Custom XML Binding Framework Model

If the ResourceStateDocument/ResourceState mechanism does not provide enough control or easy enough access to your model details, you may want to use your custom XML binding of choice to convert the returned payload into the familiar java objects. An example using JAXB is provided below:

```
//No faults so attempt to extract the contents returned
Management contactMessage = new Management(response);

//extracted content as org.w3c.dom element
Node content = contactMessage.getBody().getFirstChild();

XmlBinding binding = new XmlBinding(null);
JAXBElement<UserType> unmarshal = (JAXBElement<UserType>) binding.unmarshal(content);
JAXBElement<UserType> userReturnedElement = unmarshal;
UserType returnedUser = (UserType) userReturnedElement.getValue();
System.out.println("@@@ A Priori: Bill's Age:"+returnedUser.getAge());
```

## Command Line Tool

The final mechanism is the command line tool. The tool is typically used as a simple way to view message contents in the console and perhaps redirect such output to a file. See Chapter 4 for complete instructions for using the command line tool.

## Delivery of Client Side Access Tools:

Given the lightweight nature of the client toolkit model, it is suggested that the developer or client side developer, abstract out a few API elements/messages that they feel the clients will want to use to access the service and deliver these elements as a JAR or executable JAR to the end consumers.

For example, a Contacts/People database being exposed using a WS-Management service may dictate that most access to the service will be read only and only searching using first and last names from a Web site is supported.

The developer may provide customized access JARs to be used from within a servlet or other Web container which directly makes these API calls. The JARs can also be exposed on the end Web site where the customers are expected to query the service. With the usage of Java and JNLP mechanisms, complex GUIs that automatically download and seamlessly access the backend models are possible.

The client-side framework can be used to hide the binding/model specific processing behind an XPath exterior. By delivering an executable or extractable Java bundle, the client-side framework can be used on any platform that can run Java.



## 2 Creating Clients from WSDL

This chapter provides instructions for creating a WS-Management client from a WS-Management-based WSDL. In most cases, the WSDL is run through a client toolkit in order to generate client stubs that are used for Java-to-XML binding.

The following sections are included in this chapter:

*Retrieving a Deployed WSDL*

*WSDL generation/modifications*

*Third-Party SOAP Client Toolkit Integration*

### Retrieving a Deployed WSDL

Wiseman Web service implementations expose a WSDL that is used to discover what operations are available to a client. The following procedure is used to retrieve the WSDL. The server implementation must be deployed before a WSDL can be retrieved.

To retrieve a deployed WSDL:

- 1 Open a Web Browser.
- 2 Browse to the following address:

`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.

### WSDL Generation/Modifications

In order for the Wiseman generated WSDLs to function with as many third party client toolkits as possible, some modifications have been made to the original WS-Management schema and WSDL documents. The current release of Wiseman supports client stub generation through the Axis and JAX-WS SOAP toolkits, along with the Wiseman supplied client libraries.

Other WSDL toolkits were not tested. However, it is expected that other WSDL toolkits will be able to consume the Wiseman WSDLs but will require some work to figure out the tool specific manner in which the headers should be added.

The following is a list of the changes made by Wiseman to the default WS-Management WSDLs and schemas to allow interoperability with the Axis and JAX-WS SOAP client toolkits.

## Unique Request Message Definitions

All doc/literal Web services must have unique requests on the wire to be BP compliant (which JAXWS client generation currently requires). To accomplish this, duplicate input messages had to be defined for each SOAP message.

### Old definition:

```
<operation name="Put">
  <input message="tns:TrafficLightTypeMessage"
    wsa:Action="http://schemas.xmlsoap.org/ws/2004/09/transfer/Put"/>
  <output message="wxf:OptionalXmlMessage"
    wsa:Action="http://schemas.xmlsoap.org/ws/2004/09/transfer/PutResponse"/>
</operation>
<operation name="Delete">
  <input message="wxf:EmptyMessage"
    wsa:Action="http://schemas.xmlsoap.org/ws/2004/09/transfer/Delete"/>
  <output message="wxf:EmptyMessage"
    wsa:Action="http://schemas.xmlsoap.org/ws/2004/09/transfer/DeleteResponse"/>
</operation>
```

### New definition:

```
<operation name="Put">
  <input message="wxf:AnyXmlMessage"
    wsa:Action="http://schemas.xmlsoap.org/ws/2004/09/transfer/Put"/>
  <output message="wxf:OptionalXmlMessage"
    wsa:Action="http://schemas.xmlsoap.org/ws/2004/09/transfer/PutResponse"/>
</operation>
<operation name="Delete">
  <input message="tns:DeleteRequestMessage"
    wsa:Action="http://schemas.xmlsoap.org/ws/2004/09/transfer/Delete"/>
  <output message="wxf:EmptyMessage"
    wsa:Action="http://schemas.xmlsoap.org/ws/2004/09/transfer/DeleteResponse"/>
</operation>
```

### Added elements:

```
<message name="DeleteRequestMessage">
  <part name="DeleteRequestMessage" element="wxf:AnyXmlOptional"/>
</message>
```

## Input Message Cannot Be Empty

The `EnumerationEnd` operation has no input message defined, but both the Axis and JAX-WS toolkits require all operations to have an input operation. Since the `EnumerationEnd` operation is not recommended to be implemented, it was removed from the generated WSDLs.

### Old definition:

```
<operation name="EnumerationEndOp">
  <output message="wsen:EnumerationEndMessage"
    wsa:Action="http://schemas.xmlsoap.org/ws/2004/09/enumeration/EnumerationEnd"/>
</operation>
```

### New definition:

```
<!--operation name="EnumerationEndOp">
  <output message="wsen:EnumerationEndMessage"
    wsa:Action="http://schemas.xmlsoap.org/ws/2004/09/enumeration/EnumerationEnd"/>
</operation-->
```

## Definition of Input Messages

Get and delete input message must be defined as a schema element, not a type.

### Old definition:

```
<wsdl:message name="AnyXmlMessage">
  <wsdl:part name="Body" type="tns:AnyXmlType"/>
</wsdl:message>
<wsdl:message name="OptionalXmlMessage">
  <wsdl:part name="Body" type="tns:AnyXmlOptionalType"/>
</wsdl:message>
```

### New definition:

```
<wsdl:message name="AnyXmlMessage">
  <wsdl:part name="Body" element="tns:AnyXml"/>
</wsdl:message>
<wsdl:message name="OptionalXmlMessage">
  <wsdl:part name="Body" element="tns:AnyXmlOptional"/>
</wsdl:message>
```

## Schema Imports

Import location incorrect for addressing.xsd in eventing.xsd file. This causes duplicate type definitions in client generation. All the other WS-Management schemas already import the addressing schema to the new address below.

### Old definition:

```
<xs:import namespace="http://schemas.xmlsoap.org/ws/2004/08/addressing"
  schemaLocation="http://schemas.xmlsoap.org/ws/2004/08/addressing"/>
```

### New definition:

```
<xs:import namespace="http://schemas.xmlsoap.org/ws/2004/08/addressing"
  schemaLocation="http://schemas.xmlsoap.org/ws/2004/08/addressing/addressing.xsd"/>
```

Change schema inclusion in eventing.wsdl from include to import for correct resolution of schema elements. This matches the pattern of the other WS-Management WSDL schema imports.

### Old definition:

```
<wsdl:types>
  <xs:schema targetNamespace="http://schemas.xmlsoap.org/ws/2004/08/eventing">
    <xs:include schemaLocation="eventing.xsd"/>
  </xs:schema>
</wsdl:types>
```

### New definition:

```
<wsdl:types>
  <xs:schema>
    <xs:import namespace="http://schemas.xmlsoap.org/ws/2004/08/eventing"
      schemaLocation="eventing.xsd"/>
  </xs:schema>
</wsdl:types>
```

## Process Contents Attribute

Add processContents attribute in the addressing.xsd file. Per the schema, this should default to lax, but Axis client generation is not processing it correctly.

**Old definition:**

```
<xs:complexType name="ReplyAfterType">
  <xs:simpleContent>
    <xs:extension base="xs:nonNegativeInteger">
      <xs:anyAttribute namespace="##other"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

**New definition:**

```
<xs:complexType name="ReplyAfterType">
  <xs:simpleContent>
    <xs:extension base="xs:nonNegativeInteger">
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

## Third-Party SOAP Client Toolkit Integration

In order to simplify integrating Wiseman Web services with the JAX-WS and Axis SOAP client toolkits, Wiseman has provided sample code and two utility classes located in `WISEMAN_HOME\docs\third-party-client-samples.zip`. There are two source files, `WsmanAxisUtils.java` and `WsmanJaxwsUtils.java`, which contain some helper functions for SOAP clients. These utility classes currently are focused on adding the necessary SOAP headers to the client method calls, but may be expanded at a later date as needed.

There are also two sample code classes, `TestAxisClient.java` and `JAXWSClient.java`, which show examples of client code integration using the utility classes. Each of the code samples shows a simple call sequence to the traffic light sample service using the different SOAP client toolkit APIs. Each segment creates a traffic light and then does a get to retrieve the newly created light. An enumerate call and a subsequent pull on the enumerate results is then attempted.



## 3 Creating Clients from Metadata

This chapter provides instructions for creating WS-Management clients from metadata that is exposed by a resource's WS-Management Web service. Wiseman's metadata mechanism is offered as a convenience and is not part of the WS-Management specification.

The following sections are included in this chapter:

*Overview*

*Browsing a Resource's Metadata*

*Creating a Request*

*Processing a Response*

### Overview

Wiseman provides a metadata mechanism that can be used to create WS-Management clients. The metadata mechanism is provided in order to address the complexity of creating clients from a WS-Management WSDL. In order to build a client using metadata, a resource's WS-Management Web service implementation must first be annotated with metadata. See the *Wiseman Server Developer's Guide* located in the `WISEMAN_HOME\docs` directory for more information on adding metadata to a WS-Management Web service implementation.

The Wiseman metadata mechanism simplifies the steps required to expose a WS-Management Web service to clients. Metadata replaces the need to expose or maintain a WSDL for each service. The WSDL is replaced by a human readable document/description of the required schemas and what each exposed WS-Management service operation expects. This mechanism also returns an XML message envelope that contains all of the necessary addressing WS-Management elements already populated.

The following is a high-level view of using the metadata mechanism to consume WS-Management Web services.

- The WS-Management service developer annotates their service implementation to define all the metadata required to access the service using `MetadataResourceUID`. See the *Wiseman Server Developer's Guide* located in the `WISEMAN_HOME\docs` directory for more information on adding metadata to a service implementation.
- The client developer browses the metadata repository and locates the metadata and its associated `MetadataResourceUID`. Using Wiseman APIs and the `MetadataResourceUID`, the consumer is able to get a Management message with all the relevant addressing and descriptive SOAP elements already populated.
- The client developer sets the appropriate WS Addressing Action to be applied to the service and any other model specific elements desired for processing.

## Browsing a Resource's Metadata

Wiseman includes a metadata viewer that is used to browse a resource's service implementation metadata. The viewer is included in `WISEMAN_HOME\lib\mViewer.jar`. 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 application and requires the following system properties to be defined as command line parameter or set within the GUI:

```
-Dwsman.basicauthentication=true  
-Dwsman.user=wsman  
-Dwsman.password=secret
```

To browse a resource's metadata:

- 1 Start the Metadata Viewer and supply the required security system properties.
- 2 In the text field, enter the URL to the Wiseman server. For example:  
`http://<host>:<port>/user/`
- 3 Click **Load**. The Metadata Viewer displays all the discovered metadata. The metadata includes:
  - All of the default addressing model components (TO, ResourceURI, SelectorSets, etc...) which are likely instance-level discriminates.
  - All of the schemas (XSD's) that were used to represent the back end model on the server side.
  - All of the operations that the clients should be able to access. Both for the request messages (write/update activities) and for the responses defining the XSD types that represent the payload returned using the WS-Management messages.

A client developer now knows the requirements for correctly defining message requests and parsing message responses.

## Creating a Request

The `AnnotationProcessor.findAnnotatedResourceByUID()` method is used to find the required metadata to create a request. As part of the request, include the server URL and any HTTP credentials that are required to access the metadata exposed. A management instance is returned and is populated with all of the required default addressing model components already filled in. Next, run this instance through the `ManagementUtility` class to populate all the other components (like the `MessageId` field). You now have the envelope and all of its required data for submission. Next, decide which of the exposed service's mechanisms are to be exercised, set the `Management.Action` accordingly and insert the relevant message body if any.

The following example demonstrates a typical `Transfer.Get` request. The example is taken from the Contacts sample included in the `WISEMAN_HOME\samples` directory.

```
String serviceAddress="http://localhost:8080/users/";  
String metaDataUID=serviceAddress+"Bill.Gates";  
  
//Locate the metadata necessary to make the call  
Management get = AnnotationProcessor.findAnnotatedResourceByUID(
```

```

        metaDataUID,serviceAddress);

//set Action to request of the service
get.setAction(Transfer.GET_ACTION_URI);

//run this Management instance through ManagementUtility
get = ManagementUtility.buildMessage(get, null);

//Use the default HttpClient or your own here to send the message off
Addressing response =HttpClient.sendRequest(get);

```

## Processing a Response

The following example shows a response payload for the request created in the preceding section.

```

...
<env:Body>
  <ns9:user xmlns="http://www.w3.org/2003/05/soap-envelope"
    xmlns:ns2="http://schemas.xmlsoap.org/ws/2004/08/addressing"
    xmlns:ns3="http://schemas.xmlsoap.org/ws/2004/08/eventing"
    xmlns:ns4="http://schemas.xmlsoap.org/ws/2004/09/enumeration"
    xmlns:ns5="http://schemas.xmlsoap.org/ws/2004/09/transfer"
    xmlns:ns6="http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd"
    xmlns:ns7="http://schemas.xmlsoap.org/ws/2004/09/mex"
    xmlns:ns8="http://schemas.wiseman.dev.java.net/metadata/messagetypes"
    xmlns:ns9="http://examples.hp.com/ws/wsman/user">
    <ns9:firstname>Bill</ns9:firstname>
    <ns9:lastname>Gates</ns9:lastname>
    <ns9:address>149 Bilney Avenue</ns9:address>
    <ns9:city>Stanford</ns9:city>
    <ns9:state>IA</ns9:state>
    <ns9:zip>39000</ns9:zip>
    <ns9:age>34</ns9:age>
  </ns9:user>
</env:Body>
</env:Envelope>

```

This response indicates that the `Transfer.Get` request succeeded in locating a contact resource instance and returned the instance to the client.

The following example shows how to process the response payload to get the contacts age. Notice that `XPath` is used to select the age element:

```

Addressing response = HttpClient.sendRequest(get);

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

//With the content returned, convert it to ResourceStateDocument
// so that we can use XPath to process the results
ResourceStateDocument resState = ManagementUtility.getAsResourceState(response);

//We'll use the [local-name] syntax to avoid specifying namespaces
String billsAge = resState.getValueText("/*[local-name()='age']");

```



## 4 Using The Client Command Line Tool

The WisemanCmdLine command line tool provides a simple method for sending requests to WS-Management Web services for testing. Get, Put, Create, Delete, and optimized Enumeration are currently supported.

A batch file (`wise.bat`) and shell script (`wise.sh`), located in the `WISEMAN_HOME\tools` directory, can be used to call the WisemanCmdLine class with the necessary classpath. The tools usage follows.

```
Usage: wise -a -d -r [ -s | -b | -x | -m | -u | -p ]
```

**Table 1** Command line arguments

Argument	Description
-a	The WS-Management action to be performed (get, put, create, delete, and enumeration). This argument is required.
-d	The URL destination for the Wiseman request. This argument is required.
-r	The resource URI value of the Wiseman request. This argument is required.
-s	The name and value for the item requested. The first value is the name of the key field; the second is the value of the key field.
-b	The file name of the XML body of the request. The file must contain valid XML. This argument is required for Create and Put requests.
-x	The XPATH expression to apply to result set. This argument is used for a fragment transfer request.
-m	The maximum number of items to return. This argument is valid only for Enumeration requests.
-u	The user name to connect to the Web service.
-p	The password to connect to the Web service.

The following examples demonstrate how to use the WisemanCmdLine tool to contact the sample traffic light service:

```
wise -a get -d "http://localhost:8080/traffic/" -r
"urn:resources.wiseman.dev.java.net/traffic/1/light" -s name Light1

wise -a put -d "http://localhost:8080/traffic/" -r
"urn:resources.wiseman.dev.java.net/traffic/1/light" -s name Light1 -b
"c:\putbody.xml"
```



# Index

## A

- access model, 8
  - inbound, 8, 10
  - outbound, 8
  - outbound payload, 9
- action dispatching, 6
- addressing action, 9, 17
- annotations, 9
- Axis, 6, 13
- Axis utility classes, 16

## B

- benefits
  - metadata, 6
- binding framework, 11

## C

- client
  - inbound messages, 8, 10
  - outbound message payload, 9
  - outbound messages, 8
- client and server architecture, 7
- client command line tool, 21
- client toolkit, 5
  - client command line tool, 21
  - metadata mechanism, 17
  - Metadata Viewer, 18
  - sample classes, 16
  - utility classes, 16
- client toolkits
  - Axis, JAX-WS, 6
- client, building
  - metadata approach, 6, 17
  - WSDL approach, 6, 13
- client, building overview, 5
- client-side
  - access model, 8
  - conceptual view, 7
- command line tool, 11
- conceptual view
  - client and server architecture, 7

## D

- default addressing model, 9, 18

- doc/literal, 14
- document changes, 2
- documentation updates, 2

## E

- endpoint reference, 9
- EnumerationMessageValues, 8
- EnumerationUtility, 7, 8
- EPR, 9
- EventingMessageValues, 8
- EventingUtility, 7, 8

## I

- inbound messages, 8, 10
- instance-level discriminates, 9
- integration
  - SOAP client toolkits, 16

## J

- JAX-WS, 6, 13
- JAX-WS utility classes, 16
- JAXWSClient.java, 16

## L

- legal notices, 2

## M

- ManagementMessageValues, 8
- ManagementUtility, 7, 8, 18
- message payloads, 9
- metadata
  - browse, 18
  - client approach, 6, 17
  - create request, 18
  - process response, 19
- Metadata
  - MetadataResourceUID, 17
- Metadata Viewer, 9
  - security properties, 18
  - starting, 18
- MetadataResourceUID, 6, 17
- mViewer.bat, 18
- mViewer.sh, 18

## O

outbound messages, 8

## R

ResourceStateDocument, 9

ResourceStateDocument/ResourceState, 10

resourceURI, 9

## S

sample classes

Axis, JAX-WS, 16

security properties

Metadata Viewer, 18

server-side

conceptual view, 7

SOAP toolkits, 13, 16

## T

TestAxisClient.java, 16

third-party-client-samples.zip, 16

toolkits

Axis, JAX-WS, 5, 6, 13

transfer message, 8

Transfer.Get, 8, 9, 18

Transfer.GetResponse, 8, 9, 10

TransferMessageValues, 8

TransferUtility, 7, 8

## U

updates to doc, 2

utility classes

Axis, JAX-WS, 16

## W

wise.bat, 21

wise.sh, 21

wiseman.jar, 7

WisemanCmdLine, 21

WSDL, 6

client approach, 6, 13

generation/modifications, 13

input messages, 14, 15

ProcessContents, 15

schema imports, 15

wsman.basicauthentication, 18

wsman.password, 18

wsman.user, 18

WsmanAxisUtils.java, 16

WsmanJaxwsUtils.java, 16

## X

XML binding, 11