Introduction to Creating WS-Management Resources using Java.net's Wiseman Project

By William Reichardt

Contents

Introduction

This document is intended for someone just starting out in the area of web services manageability. Familiarity with the Java programming language (JDK5.0) and Jakarta Tomcat or any other J2EE Servlet engine is assumed. This tutorial will cover the basics of exposing an existing resource or object inside of a java VM as a WS-Man compliant resource.

Conceiving a Resource

A resource is the term we will use to describe an object with attributes and operations, which represent the component to be managed and exposed as a web service. Resources are initially modeled using a collection of primitive types such as Strings, longs and the like which are collected into complex structures. These structures are described using a dialect of XML called XML schema.

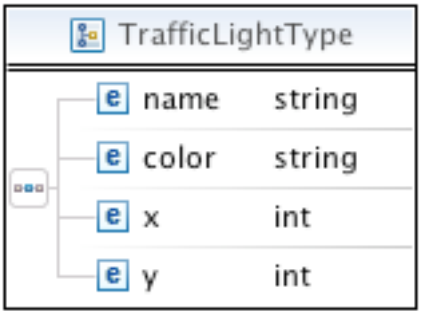
Your resource should be able to be described by and XML Schema document. There are some basic rules to follow when creating a schema document from scratch. You can start out with this template for a simple schema document.

Figure 1. A Simple XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://sun.com/traffic/light/types"
elementFormDefault="qualified" blockDefault="#all"
xmlns:tl="http://sun.com/traffic/light/types"
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>
```

Here you see an XML document which draws on the schema namespace to describe a complex type called TrafficLightType. TrafficLightType defines a template for what this XML resource should look like. In this case we are describing a resource that manages a traffic light. It has four attributes or properties. These are name, color, x and y. X and Y are the map coordinates this light has been placed at. The complex type would look like this:

Figure 2. The UML of the Traffic Light Resource



By now you probably see how you would construct a schema of your own but the structure of this document bares some explanation.

Figure 3. Structure of the schema XML document

```
schema
      complexType
            sequence
                  element
                  element
```

```
                    element
                    element
        element
```

Note that this XML document defines a schema as having two parts. The complex type, which is constructed from basic types defined in the schema specification and an element declaration. The element declaration names the element, which wraps this complex type when it is actually used in an XML document. Both parts must be present to completely describe your document.

An example of a document that could be described by this schema might look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<tl:TrafficLightType
xmlns:tl="http://sun.com/traffic/light/types">
      <tl:name>Light1</tl:name>
      <tl:color>red</tl:color >
      <tl:x>100</tl:x>
      <tl:y>210</tl:y>
</tl:TrafficLightType >
```

Here we see a sample traffic light resource. Notice that its elements are all in the namespace `xmlns:tl="http://sun.com/traffic/light/types”.` You must choose a namespace for your resource as well. Usually you will find two namespaces in a schema (XSD) document. The first is the Schema Specification namespace, `xmlns:xs="http://www.w3.org/2001/XMLSchema"` which contains the elements you will use to describe your complex type and the other will be the namespace you invent for the parts of your complex structure. Also note that schemas should also declare a target namespace. This is the namespace that all elements without an explicit namespace should fall into. In this case it should be your resource’s namespace as well. Above it is `targetNamespace="http://sun.com/traffic/light/types"`.

When building your own resources you might want to consider using a XML schema editor such as XML-Spy or Eclipse WTP. These editors will guide you and make sure your schema is valid. They will also show you all the possible primitive types offered in XML schema. Here is a short list if you intend on building your schema by hand.

http://www.w3.org/TR/xmlschema-2/#built-in-datatypes

Another alternative to hand coding your resource’s schema is to generate it using tooling. Wiseman is in the process of developing tooling to generate schema from a CIM MOF. If you are planning on exporting your CIM model you may be interested in the Wiseman Mapping tool, which is part of the Wiseman project.

What is a WSDL document and Why Do We Need One?

Now that we have described our resource you may have some questions such as: How do I declare my own operations? That is where WSDL comes in. There is far more to WSDL than you will need to deal with in this tutorial. WSDL or Web Service Description Language is used to describe your web service to the world. If someone downloads your web service's WSDL they will then know what kinds of XML documents you require and return (Via your included XSD schema file) and all of the operations you support and what parameters they require. The problem is that it is not a very human friendly XML document to work with. We will be generating our own WSDL document from the information you provided in your resources XSD document.

Once we have a WSDL document you will be able to use it for two things. These are:

1. A WSDL document can be published in a web service registry such as UDDI so that other's can know how to make calls on your resource.
2. The WSDL document will be used to auto generate a Java Web Application which will implement your web service in any J2EE application server.

The WS-Man specification defines and requires some standard operations as well. These are what you might expect. Some examples are get, put and create which are used to access your resources. These will be imported into your WSDL from the specification WSDL so that in the end, your WSDL will contain only information specific to your resource.

Configuring ANT

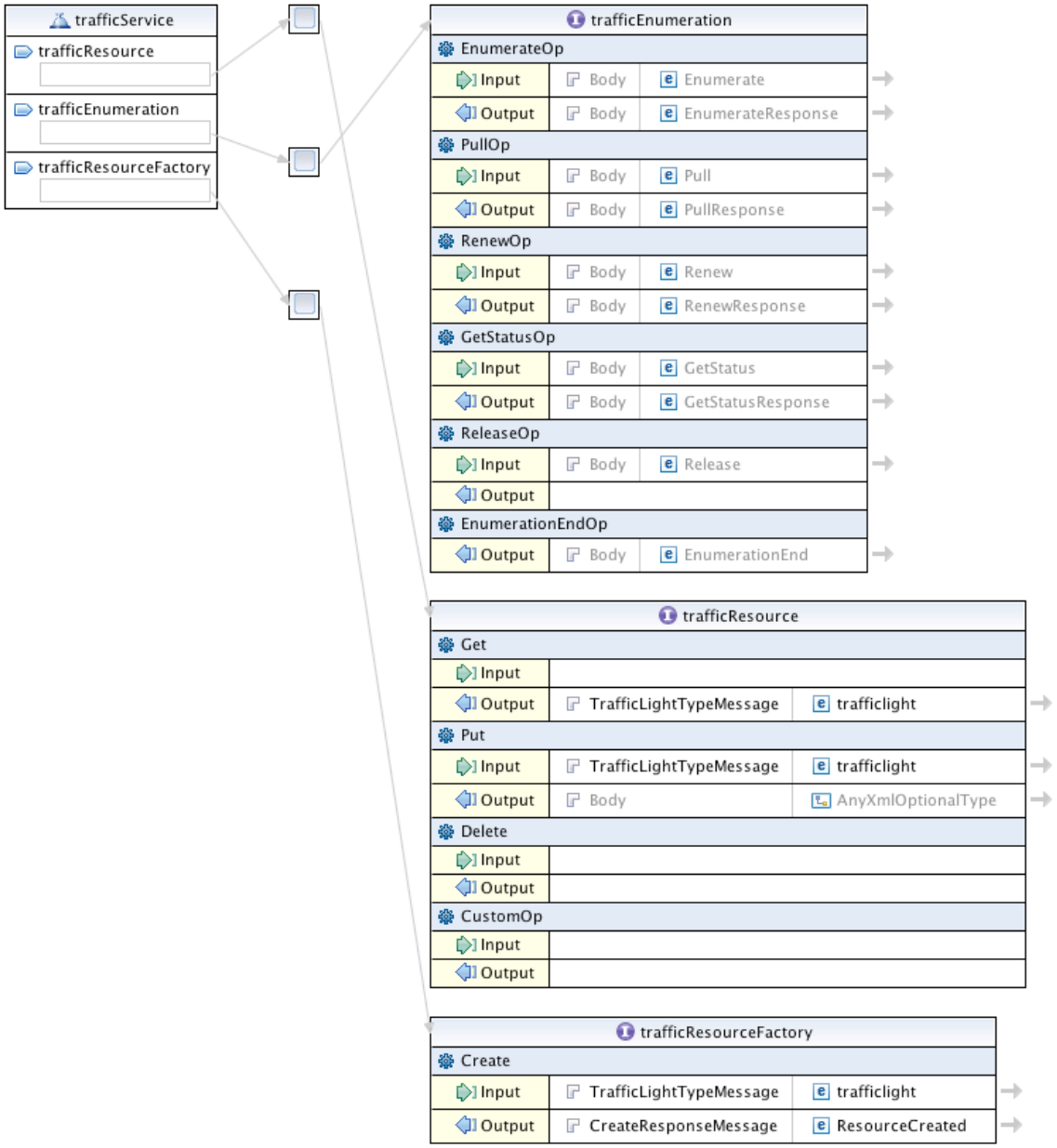
Wiseman must add jars to your Ant install before you can use its generation tools. This can be done automatically using the ant_install.bat or .sh script which can be found in the samples directory. If you are using a Unix based operating system you may need to chmod 775 ant_install.sh before running the script. The script also requires that the ANT_HOME environment variable be defined before it is run. This is the directory where ant is installed on your machine. This batch file will copy the required jars to your ANT_HOME/lib directory.

Creating your WSDL Document.

In the samples directory of your Wiseman install you will find a project called trafficlight *server. It contains an Apache ANT build script which you may want to re-use for your own projects. It also contains a copy of the traffic.xsd schema document you will be working with. Note that if you want to change the name of the schema file you are using to generate from you must change it in the project.properties file which is in the same directory as the build.xml. To generate your WSDL document from your schema, enter the following command in the samples/trafficlight*server directory: `ant.`

The default target generates WSDL from your schema and then generates Java source for a web application to expose this resource in the gensrc directory. The code in the gensrc directory is intended to be edited by you to connect the application to your actual backend and by default will respond with a "Not Implemented" fault for all of its actions until you do. We will talk more about this later.

Below is a graphic representation of the generated WSDL document, which should help to explain it more clearly.



As you can see, the WSDL generation process produced three interfaces or “ports” in web service speak. These ports are not important other than they organize the specification operations into categories. There is a factory to create new traffic lights, an interface to get properties and one to enumerate a list of lights if that is what this resource supports.

Each spec operation requires one XML document as an input and one as an output. The format of these documents if defined in the service’s WSDL, inside its own embedded schema document. You will also notice there is one operation called CustomOp. CustomOp is a sample custom operation, which is generated for you automatically. If you intend to implement your own custom operation you may copy this one as your starting point.

For this tutorial we will proceed with the generated WSDL unmodified.

Generating a Java Web Service Implementation of this WSDL

We would now like to generate a Java Application from this WSDL document. You might wonder why you might not use Apache AXIS or the JWSDP to convert this WSDL into a Java application as these product offer code generation tools. The reason is that the WS-Man 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-Man documents may have no document body at all which is not permitted by WS-I. Because AXIS and JWSDP do not yet support addressing based dispatching of XML documents, Wiseman uses its own code generation framework to create web services.

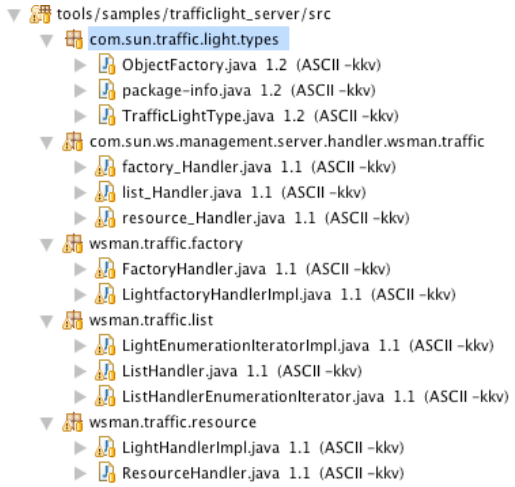
You have already generated this application in the gensrc directory when you ran ant above. Note that the process of generating the Java code for your application will require access to the Internet to download specification WSDLs as part of the generation process.

You may have also noticed that a War file was produced by the default ant target. This war is fully functional and could be deployed now because it did not use the generated application in gensrc but instead used the fully implemented version which can be found in the src directory. If you wish to skip the implementation steps below you can deploy this war now.

Connecting your generated application to your back end.

Take a look at the src and gensrc directories. They are very similar. Src is already implemented. We will now go though the steps to implement the code we generated in gensrc. It is important to remember that the ant build will overwrite gensrc as part of the build process so you may want to do your implementation work in a copy of this directory or change the build.xml.

On the right is a view of the src directory which is already implemented. It is similar to the gensrc directory you just created. The com.sun.traffic.light.types package was generated by JAXB and contains a class which you can use to represent the XML document for your resource's internal state. The com.sun.ws.management.server.handler package contains the root for wiseman handlers in your classpath. For an object with a Resource URI of wsman:traffic/resource you would see a package fragment of wsman.traffic.resource_Handler inside this root path. This is how wiseman maps



resource URI's to real handler classes by default. A handler class's job is to determine which operation to perform based on the WS Man action field provided in the WS Addressing header of the received SOAP request. The handler's generate defer all your requests to much shorter package names which mirror the Resource URI paths.

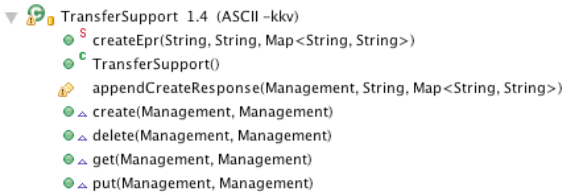
Com.sun.ws.management.server.handler.wsman.traffic.resource_Handler.class defers all its request to wsman.traffic.resource.ResourceHandler.class. You will never be expected to edit the handlers in com.sun.ws.management.server.handler. They are just there to support the wiseman dispatch mechanism.

There are three different handler types. The factory which creates new instances (in wsman.traffic.factory) , the resource itself which supports getting and setting values (in wsman.traffic.resource) and the list which is in charge of maintain of list of items of this type (wsman.traffic.list). Each has its own Resource URI. They are in order wsman:traffic/factory, wsman:traffic/resource and wsman:traffic/list. This separation of functionally by Resource URI is not mandated by the spec but helps separate a resource from a list of resources and this functionally can become easily blurred when a resource can also be a list of resources.

You do not have to implement handlers for behavior you don't plan on using. As generated this service would compile and deploy and return a not implemented fault for every request. If you only want to expose a resource that already exists (like a singleton) then just implement the Resource Handler. If you need to create lists of resources then you have to implement the list handler.

How Do I Implement Handlers?

Take a look at your generated FactoryHandler class is wsman.traffic.factory. This class extends TransferSupport which you can see below implements the Ws Transfer actions.



In this case we are only using its create implementation. You may be wondering why your generated factory class does not extend create but instead generates a Create (in uppercase) which calls create. It could very well extend create and if you were using this class by hand to

implement a service you would do this. However, because we are supporting WSDL we must provide a mapping from whatever you call your operation which means create to an actual implementation. You could have defined an operation called makeStuff and declared that it implemented the Ws Transfer create action in your WSDL and the tool would have to support it. Take a look at this WSDL snippet from the generated Traffic Light WSDL.

```
<binding name="trafficResourceFactoryBinding"
type="tns:trafficResourceFactory">
          <wsoap12:binding transport="http://schemas.xmlsoap.org/soap/http"
style="document"/>

          <operation name="Create">
                <wsoap12:operation soapAction="
http://schemas.xmlsoap.org/ws/2004/09/transfer/Create "/>
                <input>
                        <wsoap12:body use="literal"/>
                </input>
                <output>
                        <wsoap12:body use="literal"/>
                </output>
          </operation>
</binding>
```

Above you can see that we are supposed to bind the Create operation to the WS Transfer action http://schemas.xmlsoap.org/ws/2004/09/transfer/Create. The choice of uppercase Create is defined in the provided WSDL. If you obtain your WSDL from another tool, it may allow you to choose any operation name.

To Implement support for the http://schemas.xmlsoap.org/ws/2004/09/transfer/Create action you must write your implementation in FactoryHandler.create().

It currently looks like this:

```
/**
 * FactoryHandler deligate is responsible for processing WS-Transfer actions.
 * @GENERATED
 */
public class FactoryHandler extends TransferSupport
```

```
{
    //Log for logging messages
    private Logger log = Logger.getLogger(FactoryHandler.class.getName());

       private static XmlBinding binding;
       private static final String RESOURCE_JAXB_PACKAGE =
          "com.sun.traffic.light.types";
       {
              try {
                     binding = new XmlBinding(null,RESOURCE_JAXB_PACKAGE);
              } catch (JAXBException e) {
                     throw new InternalErrorFault(e.getMessage());
              }
       }
    private static final com.sun.traffic.light.types.ObjectFactory resourceFactory
      = new com.sun.traffic.light.types.ObjectFactory();
    public static final org.xmlsoap.schemas.ws._2004._08.addressing.ObjectFactory
     addressingFactory = new org.xmlsoap.schemas.ws._2004._08.addressing.ObjectFactory();
    public static final org.dmtf.schemas.wbem.wsman._1.wsman.ObjectFactory
     managementFactory = new org.dmtf.schemas.wbem.wsman._1.wsman.ObjectFactory();



     public void Create( String resource, Management request, Management response )
     {
         //TODO IMPLEMENT
         create( request, response);
     }
}
```

Create is calling create which in turn provides an action not supported fault. Not very useful so we will have to replace it. Handler methods usually receive a request and response object. They use the pre-existing request to build the response document, which will be returned. You can take a look at the implementation of this class in the src directory, which is provided by an implementation class called wsman.traffic.factory.LightfactoryHandlerImpl.java. This was done for convenience since FactoryHandler is often overwritten during Wiseman's development process. You would normally be expected to write your implementation directly in the FactoryHandler.

Implementing the Create Action

Create is implemented for you in the src directory. Compare it to the generated version in the gensrc directory. Below is how it is implemented.

```
public void Create( String resource, Management request, Management response )
     {
       // Added an implementation from another class since this code gets re-generated
often
        HashMap<String, String> selectors = LightfactoryHandlerImpl.createLight(request,
binding);
              try {
                     appendCreateResponse(response, RESOURCE_URI,selectors);
              } catch (Exception e) {
                     log.log(Level.SEVERE,"An error occured during the construction of a
```

```
response to a create request.",e);
                        throw new InternalErrorFault();
                }
    }
```

Since our create defers to another class for the details it makes a good outline of what needs to be done. First the model we are exposing must take some action to create something real inside of it. This is done by `LightfactoryHandlerImpl.createLight()` then we must product an Xml response document signaling that this has occurred which is done in `appendCreateResponse` which is a utility method.

If during this process any runtime exceptions are thrown they will be caught and expressed as SOAP faults. If you need to control which fault is thrown as the spec often dictates look in the com.sun.ws.management package of the Wiseman jar. Exceptions for specific spec faults are available there.

Since createLight manipulates a back end model which is using Swing to manufacture lights, its implementation is not important as your model will be used instead. What is important is that it creates a HashMap<String,String> which is a Name, Value map that will become the selector used to uniquely identify this new instance of this object from any other. In this case we are using only one selector which is name=light1 but there is no limit and selectors are often chosen from the values of fields in the newly created resource.

appendCreateResponse is a utility method in the base class that builds the required Xml response from the selectors you provide. It writes them to response and your work is done. You never actually had to build an XML document for create. That is because its state is stored in your model, whatever that might be. It does not have to be persisted as Xml when it is created. This will be the job of the get action, which we will implement next.

What is JAXB and why is it needed?

We need to produce Xml documents whose structure is governed by schemas. Schemas control what elements can contain and which elements follow each other. These rules are very complex to enforce using strings in Java. JAXB uses schema documents to create Java objects which can represent Xml documents which conform to the original schema. You manipulate the Object and it knows how to generate its own Xml. The process of creating an Xml document from a Java object is called **Marshalling**. The reverse is called **Unmarshalling**. JAXB Java objects are called **Types** and when a type is marshaled it becomes an **Element**. Elements can be easily saved as Xml or be embedded in other Elements and then persisted. This terminology is important to keep in mind as we proceed to use JAXB.

JAXB has two common use patterns in Wiseman. The first is creating a Type, filling in the blanks inside it by setting it fields and then Marshalling it and embedding it inside

your SOAP response. The other is taking a SOAP request and Unmarshalling its body into a Type so that its fields can be extracted and used in creating a response.

In the next section we will be using the Marshalling pattern to create a traffic document to convert our model information into an Xml response.

Implementing the Get Action

Below is the implemented Get operation and the class it is contained in. This class was generated with some useful tools to help you get started. It has a JDK logger. This will allow you to avoid using System.out.print in your handlers. It also has an Xml binding object. This object is how you will be accessing JAXB. The binding knows how to Marshall and Unmarshall your Xml document which represents your state.

JAXB Types, or Java Objects, which represent your document should not be created with new. Each package contains is own resource factory based on the target namespace of the document. Rather than you having to locate these factories, your generated handler has your document's factory declared statically for you. It is called resourceFactory. This class will be used to create new Types.

```
/**
 * ResourceHandler deligate is responsible for processing WS-Transfer actions.
 * @GENERATED
 */
public class ResourceHandler extends TransferSupport
{
    //Log for logging messages
    private Logger log = Logger.getLogger(ResourceHandler.class.getName());

       private static XmlBinding binding;
       private static final String RESOURCE_JAXB_PACKAGE =
         "com.sun.traffic.light.types";
       {
              try {
                     binding = new XmlBinding(null,RESOURCE_JAXB_PACKAGE);
              } catch (JAXBException e) {
                     throw new InternalErrorFault(e.getMessage());
              }
       }
       private static final com.sun.traffic.light.types.ObjectFactory resourceFactory
         = new com.sun.traffic.light.types.ObjectFactory();
    public static final org.xmlsoap.schemas.ws._2004._08.addressing.ObjectFactory
     addressingFactory = new org.xmlsoap.schemas.ws._2004._08.addressing.ObjectFactory();
    public static final org.dmtf.schemas.wbem.wsman._1.wsman.ObjectFactory
     managementFactory = new org.dmtf.schemas.wbem.wsman._1.wsman.ObjectFactory();



     public void Get( String resource, Management request, Management response )
     {
         //TODO IMPLEMENT
         get( request, response);
     }
```

The actual implementation calls out to an implementation class as you can see below.

```
public void Get( String resource, Management request, Management response )
     {
         LightHandlerImpl.get(request,response,log,binding);
     }
```

This is how it is implemented.

```
static void get(Management request, Management response, Logger log, XmlBinding binding)
{
        // Use name selector to find the right light in the model
        String name=null;
        try {
        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);
        }
        TrafficLightType tlType = createLightType(light);

        // Convert JABX to an element and copy it to the SOAP
        // Response Body
                Document soapBodyDoc = Management.newDocument();
                binding.marshal(trafficLightFactory.createTrafficlight(tlType),
                                soapBodyDoc);
                response.getBody().addDocument(soapBodyDoc);
        } catch (JAXBException e) {
                log.log(Level.SEVERE,
                "An error occured while creating a traffic light SOAP body for the "
                                                + name + " resource", e);
                throw new InternalErrorFault();
        } catch (SOAPException e) {
                log.log(Level.SEVERE,
                "An error occured while creating a traffic light SOAP body for the "
                                                + name + " resource", e);
                throw new InternalErrorFault();
        }
}
```

The good news is that most of this code is exception handling. Keep in mind that exceptions/faults are for your clients to see and log information is for you to read. Try to keep details that only you would understand in the log and keep your faults simple and helpful to your clients.

These are the general steps:

1. Extract the Selectors from the SOAP header. This tells you which instance of your resource is being asked to provide its state.

2. Query your model and find the resource instance. Hopefully it is already in the form of a Java object.
3. Create the JAXB Type for your resources XML document from its Factory which was pre-generated in your handler. This is done in the createLightType operation which looks like this:

```
private static TrafficLightType createLightType(TrafficLight light) {
                // Create a new, empty JAXB TrafficLight Type
        TrafficLightType tlType = trafficLightFactory.createTrafficLightType();

        // Transfer State from model to JAXB Type
        tlType.setName(light.getName());
        tlType.setColor(light.getColor());
        tlType.setX(light.getX());
        tlType.setY(light.getY());
        return tlType;
}
```

4. Copy your model's state into the fields of the type object. Above you can see this happening in the lines such as `tlType.setName(light.getName());`
5. Create an empty SOAP document with `Document soapBodyDoc = Management.newDocument();`. This document will be used to store your Marshalled Xml document.
6. Marshall your Type into your new document.
7. Use this document as your SOAP body. Insert it into the response with: `response.getBody().addDocument(soapBodyDoc);`

Now you have placed your document into the response body you are done. Take a look at how put is implemented. It is almost the exact reverse of this process.

Implementing your Enumeration/List of Resources

As part of the generation process, an implementation of a list of your resources was generated. List uses the WS Enumeration specification to expose a collection of resources of your type. Enumerations can be filtered or unfiltered. We will be implementing an unfiltered one in this example. Essentially there exists a resource, which is itself a collection of your resources. It has its own Resource URI and is usually assumed to have only one default or singleton instance and therefore selectors are not required as part of your requests. The resource URI for our enumeration is wsman:traffic/list.

There are two steps required to implement an enumeration. First is to implement a Handler similar to the ones required by the resource and the factory and the other is to implement an Iterator which allows the collection to access each of the members in the list.

Implementing ListHandler

Most of the default functionality in your generated list handler does not require any editing. Two things must be accomplished in your handler before it can be used. First it must establish an Enumeration Context. An enumeration context is a class that will be

passed into the Iterator each time this is called. If you think of the Iterator as containing information on which item in the list you are currently working with, the context can be though of as a reference to list itself. Often it is just a reference to your own list model. In our case it is all of the traffic lights in a list. The context is set in the constructor of the ListHandler class as shown below.

```
public class ListHandler extends EnumerationHandler
{
    //Log for logging messages
    private Logger log = Logger.getLogger(ListHandler.class.getName());

    //the namespace map should be populated to handle filter requests
    private Map<String, String> namespaces = null;

    //private Object m_clientContext;//TODO initialize this

    public ListHandler()
    {
        super(new ListHandlerEnumerationIterator());


        setNamespaces(namespaces);
        /**
         * TODO set the client context object
         * The clientContext is the dataset used in the Enumeration.
         */
        // All you have to do in this class is set the context to some object
        // that the Interator knows how to use.
        setClientContext( TrafficLightModel.getModel().getList());
    }
```

The call to setClientContext() allows you to set any Java object as the context for the EnumerationIterator.

The other thing that should be done is based on the enumeration methods you intend to support. If you intend to support enumerate objects then you must override `enumerateObjects()` in the ListHandler. You may then perform any specific actions you must do to prepare for this type of enumeration. You must override at least one of the three functions shown below or none of the enumeration methods will be supported.

```
    // If you don't implement one of these then all enumeration requests
    // will return unsupported. We are enumerating EPRs
      @Override
       public void enumerateEprs(Enumeration enuRequest, Enumeration enuResponse) {
              // Do nothing here. We don't require any setup code for EPR enumeration
       // Default behavior is to return unsupported.
              // super.enumerateEprs(enuRequest, enuResponse);
       }

    //  Uncomment these to support these other modes
       @Override
       public void enumerateObjects(Enumeration enuRequest, Enumeration enuResponse) {
       }
```

```
        @Override
        public void enumerateObjectsAndEprs(Enumeration enuRequest, Enumeration
enuResponse) {
        }
```

Implementing ListHandlerEnumerationIterator

These is far more work that must be done in the EnumerationIterator than in the ListHandler. In our sample in the src directory you will find implementations are contained in a class called LightEnumerationIteratorImpl. You must provide implementations for next(), hasNext() and estimateTotalItems().

Below is the definition of next including its implementation by its static impl class.

```
public List<EnumerationItem> next(final DocumentBuilder db, final Object context, final
boolean includeItem,
     final boolean includeEPR, final int startPos, int count){
        return LightEnumerationIteratorImpl.next(binding, this, db, context,
includeItem, includeEPR, startPos, count);
    }
```

The objective here is given the context which is a reference to a list in your model, an index into your list and an item count, you are required to manufacture a List of EnumerationItems. An Enumeration item can contain both a document, which is an EPR to a resource and a document representing its state. Which you actually have to provide is determined by the Booleans includeItem and includeEpr. If one of these is set you must include them inside your EnumerationItem.

In our simple example we are going to assume you will create both an Object and an EPR but your client will only be asking for EPRs. Let's take a look at the implementation of next, below.

```
        public static List<EnumerationItem> next(XmlBinding
binding,ListHandlerEnumerationIterator iterator,DocumentBuilder db, Object context,
boolean includeItem, boolean includeEPR, int startPos, int count) {
        // Create a set of elements to be returned
         // In this case each will be an EPR starting in the
         // list from startPos totalling count items
        Map<String,TrafficLight> lights=(Map<String,TrafficLight>)context;
         int returnCount = Math.min(count, lights.size() - startPos);
         List<EnumerationItem> items = new ArrayList<EnumerationItem>(returnCount);
         Set<String> keysSet = lights.keySet();
         Object[] keys = keysSet.toArray();
         for (int i = 0; i < returnCount && !iterator.isCancelled(); i++)
         {
             String lightName = keys[startPos + i].toString();
             TrafficLight light = lights.get(lightName);
             Map<String, String> selectors = new HashMap<String, String>();
             selectors.put("name", light.getName());
             try
             {
```

```
                EndpointReferenceType epr = TransferSupport.createEpr(null,
WSMAN_TRAFFIC_RESOURCE, selectors);

                // Make a traffic light type to support epr or element enum
                Element lightElement = LightHandlerImpl.createLightElement(light,binding);

                EnumerationItem item = new EnumerationItem(lightElement,epr);
                 items.add(item);
            }
            catch (JAXBException e)
            {
                throw new InternalErrorFault(e.getMessage());
            }

        }
        return items;
    }
```

In the above example we use a for loop to iterate over just the items in the model that we are asked to provide in the current next request. For each one we create an EPR using a utility method in TransferSupport that contructs EPRs from a set of selectors.

We then re-use a method developed for the resource handler, LightHandlerImpl.createLightElement() which converts a single light in our model into a JAXB element which represents our state/object in XML.

Both the EPR and the Object are used to create an EnumerationItem which will allow the Iterator to choose one of these to representations to return to the client based on if they asked for and object, and EPR or both as a response.

Next we need to implement hasNext(). HasNext simply has to indicate of this iteration will have more items in it the next time next() is called. Since we are given the current position in the iteration, this answer is simple to determine. See the implementation below.

```
public static boolean hasNext(final Object context, final int startPos){
       Map<String,TrafficLight> lights=(Map<String,TrafficLight>)context;
       if (startPos >= lights.size())
        {
            return false;
        }
        else
        {
            return true;
        }

}
```

The last thing we must implement is estimateTotalItems. This method provides an estimate as to the total number of items that may occur in this iteration. It is very useful for displaying status on the client side. In the implementation below we simply return the exact size of the list of traffic lights from the model.

```
        public static int estimateTotalItems(Object context){
                Map<String,TrafficLight> lights=(Map<String,TrafficLight>)context;
                return lights.size();
        }
```

Again, you do not have to add these implementations yourself to deploy and run this example. While doing this tutorial you have been generating your source in the gen-src directory. The fully implemented examples are actually being build into the generated war file and they can be found in the src directory.

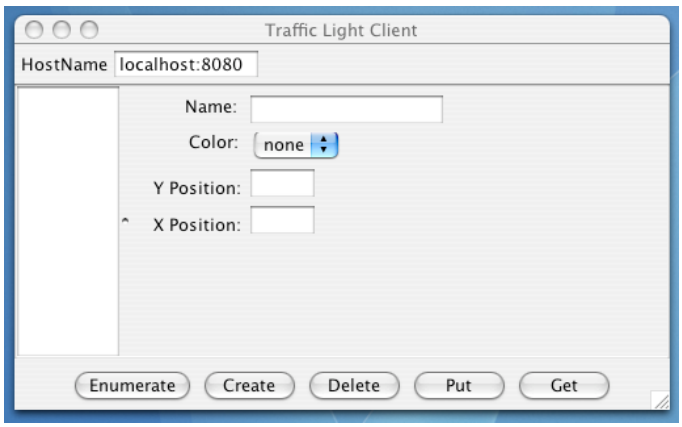
Deploying and Testing

When you ran the ant build.xml for the traffic light server, you created a traffic.war web application. You can now copy this war file manually to your application server's deployment directory or you can run the deploy target in the build.xml if you are planning on deploying to tomcat.

A swing based client that uses the Wiseman client api has been provided for this application. To build it, run ant in the samples/trafficlight_client directory. This will create a jar called traffic-client.jar in the root directory of your Wiseman install. This jar is executable. You can start the client with this command line.

java –jar traffic-client.jar

The traffic light client application UI is shown below.



It will allow you to exercise the traffic light resource. If you wish to capture the document exchange between the client and the server, try using a free tool available from the Apache foundation called tcpmon which will allow you to inspect and record the communications between your client and server. It can be downloaded at http://ws.apache.org/commons/tcpmon/.

Unit Testing

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

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

Wiseman also makes use of unit tests as well. Often times new features are documented in our unit tests before any written documentation exists so make sure to look at each components test directory if you need a code example.

What's Next?

This tutorial's purpose was to take you from XML schema to an exposed WS Management resource. Along the way we have touched briefly on the building of client applications and unit tests for your exposed resource.

What we have not yet talked about are such things as supporting:

- Filtered Enumerations
- Optimized Enumerations
- Fragment Transfers
- Eventing

These things will be covered in out next tutorial.