Wingfoot SOAP 1.03

User Guide

1. Abstract

Wingfoot SOAP 1.03 is a lightweight client implementation of SOAP 1.1 that is specifically targeted at the MIDP/CLDC platform. However, it can be used in J2SE and J2EE environments.

This release of the SOAP client is an improvement over the 1.02 release. It provides to ability to maintain session using cookies and addresses some defects identified by our customers. Details of the specific defects addressed are in Readme.txt.

The rest of this document shows the reader how to use the API to send and receive SOAP payload. This is not meant as a tutorial; the audience is expected to be familiar with SOAP.

2. Wingfoot SOAP Binary

Wingfoot SOAP 1.03 comes with two binaries. kvmwsoap_1.03.jar is targeted at the CLDC/MIDP platforms. It includes a lightweight XML parser and is 37K in size.

j2sewsoap_1.03.jar targets at the CDC/Personal Java, J2SE, and J2EE platforms. It includes a lightweight XML parser and is 34.5K in size.

The API for both binaries is identical except for the Transport implementation. kvmwsoap_1.03.jar uses HTTPTransport while j2sewsoap_1.03.jar uses J2SEHTTPTransport.

3. Supported Style and Encoding

A SOAP message can have two kinds of payload and two mechanisms to encode the data. This section explains the differences between them and the various combinations possible. It concludes by documenting the combinations supported by Wingfoot SOAP 1.03

A SOAP payload is either a RPC style or Document style. An RPC style is usually used when there is a need to invoke a remote

procedure or method. Section 7 of SOAP 1.1 document specifies the structure of a RPC style SOAP Body element (<Body>).

In a Document style, the SOAP Body contains arbitrary XML that need NOT confirm to Section 7. The SOAP client simply accepts the XML, appends it below the <Body> element and sends the payload to the SOAP server. The SOAP server passes the payload to an application and it is the responsibility of the application to parse the arbitrary XML sent in the <Body> element.

Encoding refers to the rules followed by the SOAP client and the SOAP server to interpret the contents of the <Body> element in the SOAP payload. The client and the server have to agree on one rule to ensure that either end correctly interprets the payload sent from the other end.

An encoded SOAP body indicates that the rules to encode and interpret a SOAP body are in a URL specified by the encodingStyle attribute. Wingfoot SOAP 1.03 defaults the encodingStyle to http://schemas.xmlsoap.org/soap/encoding/.

A literal encoding indicates that the rules to encode and interpret the SOAP Body are specified by a XML schema. Some services do not expect an encodingStyle attribute for literal encoding. For such instances, set the encodingStyle attribute to null (Envelope.setEncodingStyle(null))

Hence from the above discussion, there are four combinations of SOAP Style and SOAP encoding:

- RPC encoded (supported by Wingfoot SOAP 1.03)
- RPC literal (not supported by Wingfoot SOAP 1.03)
- Document encoded (not supported by Wingfoot SOAP 1.03)
- Document literal (supported by Wingfoot SOAP 1.03)

Wingfoot SOAP 1.03, like most SOAP products, supports RPC encoded and Document literal combination. For RPC encoded, the only attribute available for encodingStyle is http://schemas.xmlsoap.org/soap/encoding. Subsequent releases of our SOAP client will have support for RPC literal and Document encoded along with the ability to specify custom encodingStyle.

4. SOAP Envelope

A SOAP XML payload consists of a mandatory Envelope element, an optional Header element and a mandatory Body element. The first step is to create an Envelope. Wingfoot SOAP allows the user to use a default Envelope or create a custom Envelope.

4.1 Default Envelope

A default Envelope is created by instantiating com.wingfoot.soap.Call using the default constructor. Instances of com.Wingfoot.soap.Call are used to send a SOAP payload to the server. A default Envelope:

a. Sets the schema to http://www.w3.org/2001/XMLSchema;
b. Sets the schema-instance to http://www.w3.org/2001/XMLSchema-instance;
c. Does not allow the user to specify SOAP Header;
d. Allows only RPC-encoded SOAP Body.

The following code fragment creates an instance of Call with default Envelope:

```
Call theCall = new Call ();
```

4.2 Custom Envelope

A custom Envelope is created by instantiating com.wingfoot.soap.Envelope. Using a custom Envelope allows the user to specify

a. an alternative schema (example http://www.w3.org/1999/XMLSchema);
b. an alternative schema-instance (example http://www.w3.org/1999/XMLSchema-instance);
c. SOAP Headers;
d. RPC-encoded or Document-literal SOAP Body.

Once a custom SOAP Envelope is created, it is passed as a parameter to the constructor of com.wingfoot.soap.Call.

The following code fragment creates a custom Envelope and passes it to the Call object

```
Envelope customEnvelope = new Envelope ();
customEnvelope.setSchema (
        "http://www.w3.org/1999/XMLSchema");
customEnvelope.setSchemaInstance (
        "http://www.w3.org/1999/XMLSchema-instance");
Call theCall = new Call (customEnvelope);
```

5. SOAP Header

A SOAP Header element is optional, and is specified by creating a custom Envelope. A Header element can have a number of user-defined headers each of which is encapsulated in com.wingfoot.soap.HeaderEntry.

The following code fragment uses the custom Envelope created in the previous section to specify two headers:

```
HeaderEntry he1 = new HeaderEntry ("header1", "value1");
HeaderEntry he2 = new HeaderEntry ("header2", "value2");
he2.setMustUnderstand (true); // default is false.
customEnvelope.addHeader (he1);
customEnvelope.addHeader (he2);
```

6. SOAP Body

As discussed in Section 4 of this document, Wingfoot SOAP 1.03 supports RPC encoded and Document literal SOAP Body. This section looks at the Wingfoot SOAP 1.03 API to create these two styles of SOAP body.

6.1 RPC encoded style

The following is a sample of RPC encoded SOAP payload

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
  <SOAP-ENV:Body>
      <m:GetStockQuote xmlns:m="urn:somens"> <symbol>C</symbol></m:GetStockQuote>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

This is an encoded SOAP body because the encodingStyle attribute specifies a URL with the rules to interpret the SOAP body. Wingfoot SOAP 1.03 defaults the encodingStyle to http://schemas.xmlsoap.org/soap/envelope/.

This is a RPC style SOAP body because the structure of the XML between <SOAP-ENV:Body> and </SOAP-ENV:Body> is dictated by SOAP Section 7. In the example above, GetStockQuote is the method name. For Java based SOAP servers, the namespace "urn:somens" maps to a Java class which has a method named GetStockQuote. This method expects one parameter named symbol. C is the value for the parameter symbol.

To create a RPC encoded SOAP request, the default Envelope or a Custom Envelope can be used. The following stub uses the default Envelope to create a RPC encoded style SOAP Body.

```
Call theCall = new Call();
theCall.addParameter("symbol", "C");

Using custom Envelope, the stub looks like:

Envelope custom = new Envelope();
/**
 *Set alternate schema and schema instance here if necessary
**/
Call theCall = new Call(custom);
theCall.setMethodName("GetStockQuote");
theCall.setTargetObjectURI("urn:somens");
theCall.addParameter("symbol", "C");
```

6.2 Document literal style

In a Document literal style, the XML between <SOAP-ENV:Body> and </SOAP-ENV:Body> is supplied to the SOAP API by the client application. If the encodingStyle is not required (can be determined from WSDL), it is set to null. A custom Envelope is required for Document literal style. The stub below sends the XML stub in section 7.1 above as a Document literal.

```
Envelope custom = new Envelope();
/**
 *Set alternate schema and schema instance here if necessary
```

```
**/
String str = "<m:GetStockQuote xmlns:m="urn:somens">
<symbol>C</symbol>"
request.setBody(str);
request.setEncodingStyle(null);

Call theCall = new Call(custom);
```

This will produce a XML payload identical to the example in 7.1 above. The difference is any response returned by the server is expected to be in Document literal style and is not automatically deserialized; the contents between <SOAP-ENC:Body> and </SOAP-ENC:Body> is returned back as a string to the application; the application is responsible to parse the XML.

6.3 RPC encoded – Primitive parameter

Wingfoot SOAP 1.03 automatically serializes and deserializes instances of:

- java.lang.String
- java.lang.Byte
- java.lang.Short
- java.lang.Integer
- java.lang.Long
- java.lang.Boolean
- java.util.Date
- com.wingfoot.soap.encoding.Float
- com.wingfoot.soap.encoding.Base64

CLDC does not support java.lang.Float; hence Wingfoot SOAP 1.03 uses com.Wingfoot.soap.encoding.Float to represent Float and Double data types. The Base64 class is used to send binary data as a base64 encoded String. It also encapsulates base64 data type returned by the server.

Instances of these objects are added as parameter using the Call.addParameter(String name, String value) method.

6.4 RPC encoded – Primitive Array parameter

Wingfoot SOAP 1.03 automatically serializes and deserializes arrays of primitive types (listed above in section 7.3). Arrays are passed as parameters using Call.addParameter as the example below documents:

```
Integer intArray[] = new Integer[] {new Integer(100),
                                    new Integer(200)};
Call.addParameter("arrayParameter", intArray);
```

Wingfoot SOAP 1.03, returns arrays sent by the server as an array of Objects (Object[]).

6.5 RPC encoded – Role of TypeMappingRegistry

As discussed above in 7.3 and 7.4, primitive objects are automatically serialized and deserialized by Wingfoot SOAP 1.03. User defined java objects are a different story; they are not automatically serialized and deserialized by Wingfoot SOAP 1.03. It is impossible for a SOAP toolkit to know how to represent a user defined Java object.

An instance of TypeMappingRegistry gives the SOAP toolkit enough information to convert a custom Java object to a XML structure and vice-versa.

Wingfoot SOAP 1.03 provides an interface called WSerializable. Concrete instances of this interface represent a JavaBean. Such a concrete instance is a user defined Java object and hence is not automatically serialized and deserialized by the toolkit. Hence a TypeMappingRegistry is necessary.

The method mapTypes in TypeMappingRegistry is used to provide necessary information to the toolkit to serialize and deserialize a user defined Java object. It expects five parameters:

1. A namespace;
2. A data type;
3. an instance of the concrete user defined object;
4. an instance of Class that contains the rules to convert the concrete user defined object to XML;
5. an instance of Class that contains the rules to convert XML structure to concrete user defined Java object.

The namespace and data type is usually provided by the service (this is usually part of WSDL). The toolkit provides a class named BeanSerializer that has the rules to convert user defined Java object to XML and vice-versa. This is passed as the fourth

and fifth parameter. The third parameter is the object to be converted to XML or vice-versa using the rules specified in fourth and fifth parameter.

If a custom Java object is encountered in the payload returned by the server that is not mapped in TypeMappingRegistry, the toolkit has not way to determine the class to deserialize the structure into. In such an instance, the custom class is deserialized into an UntypedObject. The methods in UntypedObject are now used to retrieve the data elements of the structure.

6.6 RPC encoded – Bean Parameter

This section provides an example of passing an instance of WSerializable as a parameter. It uses the concepts discussed above in section 7.5.

The first task is to define a Bean. Due to the absence of full-blown reflection in CLDC, a JavaBean has to implement an instance of WSerializable. The stub below defines an Employee Bean (in the interest of brevity, only important parts are shown; the import statements are left out).

```
public class Employee implements WSerializable {

private Object name;
private Object age;

public int getPropertyCount() {return 2;}

public String getPropertyName(int index) {
   if (index == 0)
       return "name";
   else if (index == 1)
       return "age";
}

public Object getPropertyValue(int index) {
   if (index == 0)
       return name;
   else if (index == 1)
       return age;
```

```
public void setPropertyAt(Object newProperty, int index) {
   if (index == 0)
       name=newProperty;
   else if (index == 1)
       age=newProperty;
}

public void setProperty(String name, Object value) {
   if (name.equals("name"))
       this.name=value;
  else if (name.equals("age"))
       this.age=value;

public String getName() {
    return (String) name;
}

public Integer getAge() {
    return (Integer) age;
}

} /* class Employee */
```

The following stub shows how to send this Bean to the SOAP server.

```
WSerializable myEmployee = new Employee();
ws.setProperty("name", "Tiger Woods");
ws.setProperty("age", new Integer(25));

TypeMappingRegistry registry = new TypeMappingRegistry();
registry.mapTypes("employeeNS",
                  "employeeType",
                   myEmployee,
                   new BeanSerializer().getClass(),
                   new BeanSerializer().getClass());

Call theCall = new Call();
theCall.setMappingRegistry(registry);
theCall.addParameter("employee", myEmployee);
theCall.setMethodName("getEmployeeDetails");
theCall.setTargetObjectURI("urn:employeeClass");
```

```
// theCall.invoke(Transport) is invoked here to call the server.
```

This produces the following SOAP payload (for brevity, namespaces and other attributes are not shown).

```
<Envelope>
<Body>
<ns1:getEmployeeDetails xmlns:ns1=”urn:employeeClass”>
<employee xmlns:ns2=employeeNS xsi:type=”ns2:employeeType>
<name xsi:type=xsd:string>Tiger Woods</name>
<age xsi:type=xsd:int>25</age>
</employee>
</getEmployeeDetails>
</Body>
</Envelope>
```

6.7 RPC encoded – Array of Beans parameter

The toolkit can send and receive an array of Bean parameters. To send such an array, the first task at hand is to define the Bean and the TypeMappingRegistry. The next step is to add the array of Beans using call.addParameter.

6.8 Untyped Objects

Wingfoot SOAP 1.03 tags each parameter with explicit data type. A call to theCall.addParameter(“golfer”, “Tiger Woods) results the following XML stub

```
<golfer xsi:type=”xsd:string”>Tiger Woods</golfer>
```

Not all SOAP servers are required to provide explicit typing information. Some SOAP servers return the following stub:

```
<golfer>Tiger Woods</golfer>
```

In such instances, the Wingfoot SOAP 1.03 converts the parameter to a String. Consider the following stub:

```
<age>25</age>
```

Although age would represent an Integer, the toolkit has no way of knowing this; hence it will convert the age parameter as a String and present the String to the user.

If a structure returned from a server is not typed, the structure is deserialized into an instance of UntypedObject. UntypedObject implements WSerializable and the parameter names and their values within the structure are retrieved using the methods provided in the WSerializable interface. Consider the following XML stub

```
<Envelope>
<Body>
<ns1:getEmployeeDetails xmlns:ns1="urn:employeeClass">
<employee>
<name>Tiger Woods</name>
<age>25</age>
</employee>
</getEmployeeDetails>
</Body>
</Envelope>
```

Here, the element employee has a structure that has two properties: name and age. However these elements are not typed. The toolkit creates an instance of UntypedObject for the employee element; the instance of UntypedObject contains two elements. UntypedObject.getPropertyValue(int index) returns the Tiger Woods (index 0) and 25 (index 1) as String.

Users can avoid this unpleasant behavior by mapping the element to a data type. In this case, the element name is mapped to a String, the element age is mapped to Integer and the element employee is mapped to Employee (Employee is implemented in section 7.6 above).

Elements are mapped using the mapElements method in TypeMappingRegistry. It expects three parameters in the following order

1. the element name
2. instance of Class that represents the class that the value of the element is to be converted.
3. instance of Class that represents, if required, a custom Bean Deserializer. This is required only for instances of WSerializable.

The following code stub demonstrates the mapping of elements

```
Call theCall = new Call();
TypeMappingRegistry registry = new TypeMappingRegistry();
registry.mapElements(name, " ".getClass(), null);
```

```
registry.mapElements(age,
Class.forName("java.lang.Integer"),null);
registry.mapElements(employee, myEmployee.getClass(),
                              Class.forName(Employee));
theCall.setMappingRegistry(registry);
```

When the toolkit encounters either an employee, name or age element, it deserializes it into String, Integer and Employee respectively.

7. SOAP Transport

A call to a SOAP server is made using the *invoke* method in Call. It expects an instance of Transport as a parameter. The toolkit provides two implementation of Transport: HTTPTransport provides a means for MIDP applications to send a SOAP payload over HTTP; J2SEHTTPTransport provides PersonalJava/CDC, J2SE and J2EE applications to send a SOAP payload over HTTP. Users can implement alternative transport layers (SMTP as an example) by implementing the Transport interface.

Call.invoke returns an instance of Envelope. This encapsulates the response sent by the SOAP server. Section 9 below details how to interrogate the response.

HTTP is a stateless protocol and hence the request-response cycle is independent and isolated. Cookies are a popular mechanism to maintain state across requests. The server generates a unique identifier (a cookie) for each session. The server passes this identifier back to the client as pars of the response to the client's first request. The client is then responsible for passing it back to the server one ach subsequent request. The server sends the cookie using the Set-Cookie HTTP header and the client sends the cookie back to the server using the Cookie HTTP header.

The content of the cookie sent by the server depends on the kind of server. For J2EE based servers, the cookie looks as follows:

```
Set-Cookie: JSESSIONID=123dkdfk%8erterrvxvmKK08;path=/wingfoot
```

For J2EE based server, the cookie is always named JSESSIONID. For other servers, it varies. The Set-Cookie header might consist of many other optional name-value pairs separated by semicolon.

The useSession(boolean, String) method in HTTPTransport and J2SEHTTPTransport allows an application to specify that it is interested to maintain state across HTTP requests. The String (second parameter) allows the application to specify the name of the cookie. If null, it is assumed to be JSESSIONID.

If the boolean is true and wSOAP detects a Set-Cookie header, it scans the string to determine if the specified key (the second parameter in the useSession method) is present. If present, it extracts the value of the cookie and sends it back in subsequent requests to the server. Hence the subsequent request to the server might look as follows:

```
Cookie: JSESSIONID=123dkdfk%8erterrvxvmKK08
```

8. Interrogating Response

The Envelope returned from Call.invoke encapsulates the response from the SOAP server. The SOAP server can return SOAP Header(s), a Fault indicating an exception and return parameters. This section examines how to retrieve them from the response Envelope.

8.1 Retrieving Headers

The getHeader method in Envelope returns a Vector of HeaderEntry.

```
Vector v = response.getHeader();
```

Each instance of this Vector is a HeaderEntry. Methods in HeaderEntry are used to retrieve information about the Header.

8.2 Retrieving Fault

The isFaultGenerated method in Envelope returns a boolean to indicate if a Fault element is present in the SOAP response. If the Fault element is present, it is retrieved using getFault method. The code fragment to retrieve the Fault from the response generated in Section 7 is:

```
if (response.isFaultGenerated) {
    Fault fault = response.getFault();
}
```

8.3 Retrieving Parameters

Envelope contains methods to retrieve parameters. For RPC style request (which returns a RPC style response), the following methods used are getParameterCount, getParameterName and getParameter.

For Document style request (which returns a Document style response) getBody returns the contents between <Body> and </Body> elements as a String.

9. Resources

Wingfoot SOAP maintains a moderated mailing list to discuss all aspects of its products including technical how to and suggestions for future enhancements. You can find additional information about joining the newsgroup at http://www.wingfoot.com/mailinglist.jsp.