

Spheon JSOAP

An implementation of SOAP 1.1 for Java



Florian Müller

June 2003

Contents

1. Introduction	4
1.1. Features and Limitations	4
1.2. Requirements	5
1.3. Feature requirements	5
1.4. Installation	5
2. How to ...	6
2.1. How to write a SOAP RPC Client	6
2.1.1. Low level RPC Client	6
2.1.2. High level RPC Client	8
2.1.3. Arrays	10
2.1.4. Document calls	10
2.2. How to set up a webservice	11
2.2.1. Arrays	11
2.2.2. Null values	12
2.2.3. In, out and in/out parameter	12
2.2.4. Returning a SOAP fault	13
2.2.5. Controlling SOAP headers: pre and post methods	13
2.3. How to write a SOAP RPC Server	13
2.3.1. Servlets	14
2.3.2. Servlet quick installation guide	16
2.3.3. Stateful webservices	17
2.4. How to write a serializer/deserializer	17
2.4.1. WSDL	17
2.5. Understanding SoapConfig	18
2.6. Understanding the configuration file	18
2.6.1. Services	18
2.6.2. Methods	19
2.6.3. Types	19
2.6.4. Attachments	20
2.6.5. XML parser	20
2.6.6. Imports	20
3. Tools	21
3.1. ConfigWizard	21
3.2. WSDLWizard	22
4. Using Spheon JSOAP in applets	23
4.1. Old browsers	23

Contents

A. Standard type mapping	25
B. Troubleshooting	26
C. License	27

1. Introduction

Spheon JSOAP is an implementation of SOAP 1.1 (Server and Client) for Java.

- Documentation (still) under construction -

Homepage: <http://spheon-jsoap.sourceforge.net/>

Contact: mueller@gotux.de

1.1. Features and Limitations

Features

- Small and fast.
- Supports in, out and in/out parameter.
- Build-in support for int, integer, long, short, decimal, float, double, boolean, string, date, dateTime, timeInstance, base64, hex, structs and beans.
- Build-in support for all kind of SOAP arrays.
- Serialization and deserialization are done automatically through reflection.
- Custom serializer could be easily plugged in.
- Supports Map and Vector from Apache SOAP.
- Not bound to a transport protocol. (Servlet and HTTP-Client included.)
- Is able to generate WSDL-Files.
- Needs only an additional JAXP-Parser (DOM and SAX support).
- Supports SOAP with attachments.
- Supports MIME and DIME.
- Supports Java 1.1 applets with few limitations (no high level client and no SSL support).

1.2. Requirements

- [Java 1.3](#) or higher for all features
- [Java 1.2](#) for none high-level clients
- [Java 1.1](#) for applets
- A JAXP compatible XML-Parser (e.g. [Xerces](#) or [Crimson](#))
- Basic knowledge about SOAP und webservices

1.3. Feature requirements

- HTTP server: A servlet engine (e.g. [Apache Tomcat](#))
- Mail server: [Apache James](#)
- Client SSL support: [Java Secure Socket Extension \(JSSE\)](#)
- Java 1.1 support: [JDK1.1 Collections package](#)

1.4. Installation

The installation is pretty easy:

Your CLASSPATH must contain `spheon-jsoap.jar`, your favorite XML-Parser and your own classes. If you want to write a SOAP RPC server you have to create a configuration file, as described in section [2.6](#) or use the `ConfigWizard` to create it (see section [3.1](#)). If you want to write a SOAP client you need a configuration file only if the webservice uses complex types (structs or beans).

That's it!

2. How to ...

The next sections illustrate the usage of Spheon JSOAP.

2.1. How to write a SOAP RPC Client

To write a SOAP RPC Client you need the following information about the webservice you like to call:

- Name of the method(s) (line 9, listing 2.1)
- Namespace(s) of the method(s) (line 8, listing 2.1)
- Parameters of the method(s) (line 13, listing 2.1)
- Endpoint (line 23, listing 2.1)
- Required SoapAction (line 24, listing 2.1)

In most cases the webservice provides a WSDL file where all these information can be found. Spheon JSOAP cannot process such a WSDL file at this time. That could be a feature in a future releases.

There are two ways to build a RPC Client: a low level and a high level client. (High level clients need Java 1.3 or higher.)

2.1.1. Low level RPC Client

Steps to create a low level SOAP RPC client (listing 2.1):

1. Import `de.fmui.spheon.jsoap.*` and `de.fmui.spheon.jsoap.transport.http.*`.
2. Create a `SoapConfig` object (line 2) and set an actor for fault messages (line 4). More information about the `SoapConfig` class can be found in section 2.5. You can use this object for more than one call.
3. Prepare the call by creating a `RPCcall` object (line 7). It takes the `SoapConfig` object, the namespace of the method and the name of the method as parameters.
4. Set the parameters of the method (line 13). The `setParameter()` method of the `RPCcall` object needs the name of the parameter, the value and the java class of the value. If you use complex types here you have to declare them to the `SoapConfig` object first (see section 2.5 and section 2.6). Arrays in SOAP are more powerful than Java arrays. Read section 2.1.3 for details. If serializing of a parameter fails the `setParameter` method throws a `SoapException`.

For most SOAP server implementations the order of parameters is important. Some implementations use the parameter name to identify the parameter. To be on the safe side, obey both rules of thumb.

5. At this point you are able to send the call the appropriate way. You can get the SOAP message by the `getEnvelope()` method.

Spheon JSOAP has a built-in HTTP and HTTPS client, that could be used for most webservices. Create an object of `SoapHttpClient` (resp. `SoapHttpsClient`) (line 21) and call the `invokeHTTP()` (resp. `invokeHTTPS()`) method. Use the endpoint, the `SOAPAction` and the envelope of your call as parameters to send (line 22). The return value is the answer of the called webservice as an envelope. If the process of transforming the answer to an envelope fails (e.g. if the SOAP message is invalid) a `SoapException` will be thrown.

The default timeout for an answer is 60 seconds. The timeout can be set with the `setTimeout()` method.

6. Check for faults by using the `hasFault` method of the `Envelope` object (line 28). If the fault exists get the fault information and don't use this object anymore.
7. Get the result from the received envelope (line 37). The method `getResults()` of the `RPCcall` object detaches the return values from the envelope and creates a `ParamBag` object. If the type of the return value is known you can cast them directly for further processing (line 37).

The method `getResults()` uses the type information from the SOAP message. If the types of the return values are known, use the method `getExpectedResults()` instead. It ignores the type information from the SOAP message and uses the given array of `Class`. This works for server implementations that doesn't send type information within the SOAP message, too.

Listing 2.1: SOAP RPC Client

```
1 // get configuration
2 SoapConfig sc = new SoapConfig();
3 // set the actor for fault messages
4 sc.setActor("/sample/client");
5
6 // prepare a call
7 RPCcall call = new RPCcall( /* SoapConfig */ sc,
8                             /* namespace */ "urn:mySampleService",
9                             /* method */ "sampleMethod");
10
11 // set parameters
12 try {
13     call.setParameter( /* name */ "mystring",
14                       /* value */ "sample",
15                       /* type */ String.class);
16     call.setParameter( /* name */ "count",
17                       /* value */ new Integer(5),
18                       /* type */ int.class);
19 }
```

```
20 // execute call (via HTTP)
21 SoapHttpClient shc = new SoapHttpClient(sc);
22 Envelope answer = shc.invokeHTTP(
23     /* server */ new URL("http://localhost:8080/servlet/RPC"),
24     /* SOAPAction*/ "",
25     /* envelope */ call.getEnvelope());
26
27 // check for a fault message
28 if(answer.hasFault()) {
29     Fault f = answer.getFault();
30     System.out.println("Faultcode: " + f.getFaultcode());
31     System.out.println("Faultstring: " + f.getFaultstring());
32     System.out.println("Faultactor: " + f.getFaultactor());
33     return;
34 }
35
36 // get the results
37 ParamBag p = call.getResults(answer);
38 // ParamBag p = call.getExpectedResults(answer, new Class { String.
39     class });
40
41 // display the results
42 for(int i=0; i<p.size(); i++) {
43     System.out.print(p.getName(i) + "\t");
44     System.out.print(p.getType(i).getName() + "\t");
45     System.out.print(p.getValue(i) + "\n");
46 }
47 // If we know, the first return value is a string ...
48 String result = (String) p.getValue(0);
49 System.out.println("Result: " + result);
50 }
51 catch(Exception e) {
52     System.out.println("Exception: " + e);
53 }
```

2.1.2. High level RPC Client

A high level client hides the SOAP details from the programmer. The methods of a SOAP service are mapped in a Java interface, so that the programmer can use them like local Java methods.

Steps to create a SOAP RPC client interface:

1. Create a new interface with the same method signatures as the service you like to call.
2. Every method needs 3 constants:

method name_NAMESPACE
namespace of the method (line 9)

method name_SOAPACTION
SOAPAction of the method (line 10)

method name_PARAMETER
parameter names in the correct order (line 11)

3. You may also set the constant `SERVER_URL` (line 3). This will be used if no endpoint information is present when the `ProxyHelper` object will be created.
4. If the webservice uses complex types (structs), create a String array `TYPES` (line 5). There must be exact 3 items:
 - a) Namespace
 - b) SOAP type
 - c) Java type
5. If you want to use Java beans instead of structs, create a String array `BEANS`. There must be exact 3 items:
 - a) Namespace
 - b) SOAP type
 - c) Java type

Listing 2.2: Service interface

```
1 public interface ServiceInterface
2 {
3     public String SERVER_URL = "http://localhost:8080/servlet/RPC";
4
5     public String[][] TYPES = {
6         { "http://spheon.bov.de", "ChemistryElement", "samples.online.
7           ChemistryElement" }
8     };
9
10    public String sampleMethod_NAMESPACE = "urn:mySampleService";
11    public String sampleMethod_SOAPACTION = "";
12    public String[] sampleMethod_PARAMETER = {"mystring", "count"};
13    public String sampleMethod(String mystring, int count);
14 }
```

Steps to create a SOAP RPC client with this interface:

1. Import `de.fmui.spheon.jsoap.*`.
2. Create a `ProxyHelper` object with your interface (line 2).
3. Get an object of your interface from the `ProxyHelper` with the `bind()` method (line 3).
4. Now you can use this object to call the webservices (line 5).

Listing 2.3: SOAP RPC Client

```
1 try {
2     ProxyHelper ph = new ProxyHelper(ServiceInterface.class);
3     ServiceInterface service = (ServiceInterface) ph.bind();
4
5     String result = service.sampleMethod("sample", 5);
6     System.out.println("Result: " + result);
7 }
```

```
7 }
8 catch(SoapException se) {
9     if(se.containsFault()) {
10         Fault f = se.getFault();
11         System.out.println("Faultcode: " + f.getFaultcode());
12         System.out.println("Faultstring: " + f.getFaultstring());
13         System.out.println("Faultactor: " + f.getFaultactor());
14     }
15 }
16 catch(Exception e) {
17     System.out.println("Exception: " + e);
18 }
```

2.1.3. Arrays

Arrays in SOAP can be divided into 3 different groups:

- “Normal” Arrays (as used in Java).
- Partially Transmitted Arrays (p-t-a) (with an offset).
- Sparse Arrays (only a few indices are set).

Because Partially Transmitted Arrays and Sparse Arrays cannot be mapped into Java arrays without a loss of information, Spheon JSOAP provides the two classes `de.fmui.spheon.jsoap.util.OffsetArray` and `de.fmui.spheon.jsoap.util.SparseArray`.

The `OffsetArray` class is used for “Normal” Arrays (with `offset=0`) and Partially Transmitted Arrays; the `SparseArray` class for Sparse Arrays. The class `de.fmui.spheon.jsoap.SoapArray` provides an interface for these classes and adds some extra functionality.

In contrast to Java arrays these array classes must be initialized with a lower and an upper bound. There are no multi dimensional arrays in Java, but arrays of arrays. To support (real) multi dimensional arrays in SOAP these classes can handle multi dimensional lower and upper bounds.

If an incoming SOAP message contains an array Spheon JSOAP always creates a `SoapArray` object. If you want to send an array you can either use a Java array (multi dimensional arrays will be serialized as arrays of arrays) or a `SoapArray` object.

2.1.4. Document calls

Spheon JSOAP supports literal calls on the client side. Just exchange the `RPCCall` class with the `DocumentCall` class if you use a low level client. The high level client will do that automatically.

A document call needs and returns a `org.w3c.dom.Document` object, that can be set with the `setDocument()` method of `DocumentCall`.

2.2. How to set up a webservice

A webservice is a collection of stateless methods. To develop a webservice with Spheon JSOAP you have to program your methods into one or more classes. All methods which should be accessible via SOAP must be declared `public`.

It is possible to use class variables, but it makes no sense in this context because the object will be created when a call arrives and destroyed after finishing (exceptions see section 2.3.3).

To use the new webservice you have to create a configuration file (see section 2.6). The ConfigWizard may help you with this task (see section 3.1). The RPC Server must read this configuration file (see section 2.3). Your class must be in the CLASSPATH of the RPC Server!

Listing 2.4: A simple webservice

```
1 import java.util.*;
2
3 public class MyWebservice()
4 {
5     public String getMyTime()
6     {
7         return (new Date()).toString();
8     }
9     :
10
11    public int max(int a, int b)
12    {
13        if(a > b)    { return a; }
14        else        { return b; }
15    }
16    :
17 }
```

2.2.1. Arrays

If you need an array as a parameter use the `de.fmui.spheon.jsoap.SoapArray` class (see section 2.1.3). It is able to give you all data from the SOAP message even if they are outside the possibilities of Java arrays.

If you use Java arrays instead of `SoapArray` as parameter, your method will not be found!

Listing 2.5: Iteration through an array of integers with one dimension

```
1 public int sum(de.fmui.spheon.jsoap.SoapArray intArray)
2     throws de.fmui.spheon.jsoap.SoapFaultException
3 {
4     if(intArray.isNull()) { return 0; }
5     if(intArray.getType() != int.class) {
6         throw new de.fmui.spheon.jsoap.SoapFaultException("Expected an
7             array of integer!");
8     }
9     int result = 0;
10
11    for(int i = intArray.getLower()[0]; i <= intArray.getUpper()[0]; i++) {
12
```

```
13     if(intArray.exists(i)) {
14         result += intArray.getInt(i);
15     }
16 }
17
18 return result;
19 }
```

2.2.2. Null values

In contrast to SOAP primitive types in Java cannot have null values. If you need this use the wrapper types as parameters and/or return value. Make sure that you declare this in the configuration file. Otherwise your method will not be found!

2.2.3. In, out and in/out parameter

Java doesn't support "out" and "in/out" parameters. To use such parameters for webservices, the class `SoapReturn` is used. This class can be used in a webservice method (section 2.2) as a return value and enables the programmer to realize "out" and "in/out" parameters in SOAP messages. Spheon JSOAP detects such a return value and builds the correct SOAP fragment.

The constructor of `SoapReturn` takes the number of parameters for this method plus the number of additional "out" parameters (line 3, listing 2.6). By default all parameters are "in" parameters.

There are four methods of `SoapReturn` that specify the parameters:

`ret()`

Sets the value and the type of the return value.

`in()`

Declares the given parameter (\leq number of method parameters) as an "in" parameter.

`inout()`

Declares the given parameter (\leq number of method parameters) as an "in/out" parameter and set the value and the type. The RPC Server knows the name of this parameter from the request.

`out()`

Declares the given parameter ($>$ number of method parameters) as an "out" parameter and set the name, the value and the type.

Listing 2.6: In out and in/out parameter

```
1 public de.fmui.spheon.jsoap.SoapReturn testSoapReturn(int i,
2 String s, double d) {
3     de.fmui.spheon.jsoap.SoapReturn sr = new de.fmui.spheon.jsoap.
4         SoapReturn(4);
5     sr.ret("return my!", String.class); // sets the return value
6     sr.in(1); // not necessary
7     sr.inout(2, (new StringBuffer(s).reverse()).toString(), String.class);
8         // sets an in/out value
```

```
8     sr.out(4, "mult", new Float(i * d), float.class); // sets an out value
9
10    return sr;
11 }
```

2.2.4. Returning a SOAP fault

It is possible to return a SOAP fault from a webservice by throwing the exception `de.fmui.spheon.jsoap.SoapFaultException`. See the JavaDoc documentation for details.

2.2.5. Controlling SOAP headers: pre and post methods

Before and after the real method is called, Spheon JSOAP tries to call the method `preMethodName` respectively `postMethodName` within the object. The given `SoapContext` object contains the received envelope. That could be used to extract the SOAP header. The `postMethodName` contains the return envelope. Its possible to change this envelope to add information to the SOAP message.

Listing 2.7: pre and post methods

```
1 public int mult(int a, int b)
2 {
3     return a * b;
4 }
5
6 public void premult(de.fmui.spheon.jsoap.SoapContext context) {
7     System.out.println("before call 'mult'");
8     System.out.println("SOAPAction: " + context.getInputEnvelope().
9         getSoapAction());
10    System.out.println("Input envelope:" + context.getInputEnvelope());
11 }
12 public void postmult(de.fmui.spheon.jsoap.SoapContext context) {
13     System.out.println("after call 'mult'");
14     System.out.println("Output envelope:" + context.getOutputEnvelope());
15 }
```

2.3. How to write a SOAP RPC Server

The development of a SOAP RPC Server depends on the transport protocol. In this section only the core Spheon JSOAP server code will be explained. Section 2.3.1 deals with servlets.

Steps to create the SOAP RPC server (listing 2.8 and listing 2.9):

1. In the initialization phase of your server create a `SoapConfig` object (line 1) and set an actor for fault messages. More information about the `SoapConfig` class can be found in section 2.5.
2. Add the configuration files of your webservises to the configuration (line 3).

3. An incoming request (=SOAP message) should arrive as string. If it is UTF-8 encoded use `Util.fromUTF8()` to decode it. At this point there are two possibilities:

- a) Call the method `SoapParser.executeRPC()` and pass the `SoapConfig` object and the string as parameters (line 10, listing 2.8). The method analyzes the request, calls the correct webservice method, takes the result, creates an answer SOAP message and returns it. If something fails, it creates a matching SOAP fault message. The result can be directly send to the requesting client. Use `Util.toUTF8()` to encode it if necessary.
- b) Call the method `SoapParser.read()` and pass the string (line 10, listing 2.9). `SoapParser.read()` transforms the string into an `Envelope` object. You can use this `Envelope` object to get header information or other special attributes or nodes. After that, call `SoapParser.execute()` (line 15) which behaves like `SoapParser.executeRPC()`.

Listing 2.8: SOAP RPC Server 1

```
1 SoapConfig sc = new SoapConfig();
2 sc.setActor("/sample/server");
3 sc.addToConfig("sample.xml");
4 sc.addToConfig("mywebservice.xml");
5
6     :
7
8 String soapContent = ...
9
10 String result = SoapParser.executeRPC(sc, soapContent);
```

Listing 2.9: SOAP RPC Server 2

```
1 SoapConfig sc = new SoapConfig();
2 sc.setActor("/sample/server");
3 sc.addToConfig("sample.xml");
4 sc.addToConfig("mywebservice.xml");
5
6     :
7
8 String soapContent = ...
9
10 Envelope env = SoapParser.read(sc, soapContent);
11 Header header = env.getHeader();
12
13     :
14
15 String result = SoapParser.execute(sc, env);
```

2.3.1. Servlets

Spheon JSOAP comes with a ready-to-run servlet (`de.fmui.spheon.jssoap.transport.http.RPC`) which can be used or can be copied and modified.

2. How to ...

If you want to use it configure your servlet engine and set the servlet parameter “jsoapconfig” or “jsoapconfigdir” to the path of your configuration file (see [2.3.2](#)) resp. to a directory that only contains configuration files.

If you want to build your own servlet take `de.fmui.spheon.jsoap.transport.http.RPC` as a base. It should be easy to understand after you read section [2.3](#).

Be sure that `spheon-jsoap.jar`, an XML-Parser and the webservices classes are in the CLASSPATH!

2.3.2. Servlet quick installation guide

1. Create a new context or use an existing one.
2. Copy `spheon-jsoap.jar` to `{context-dir}/WEB-INF/lib/`.
3. Copy your favorite JAXP-parser (e.g. `xerces.jar`) to `{context-dir}/WEB-INF/lib/`.
4. Copy your webservice classes to `{context-dir}/WEB-INF/classes/` or to `{context-dir}/WEB-INF/lib/` if you have a jar file.
5. Copy your webservice configuration file to `{context-dir}/WEB-INF/`.
6. Add this to the file `{context-dir}/WEB-INF/web.xml` (replace `sample.xml` with your configuration file):

```
<servlet>
  <servlet-name>RPC</servlet-name>
  <servlet-class>de.fmui.spheon.jsoap.transport.http.RPC</servlet-class>
  <init-param>
    <param-name>jsoapconfig</param-name>
    <param-value>{CTX_ROOT}/WEB-INF/sample.xml</param-value>
  </init-param>
</servlet>
```

7. Reload the context or restart the servlet engine.
8. URL endpoint of the webservices is,
 - if you modified `web.xml` of ROOT context:
`http://myserver.com/servlet/RPC`
 - if you modified `web.xml` of examples context:
`http://myserver.com/examples/servlet/RPC`

put your hostname/IP or “localhost” where `myserver.com` is written.

9. (optional) Spheon JSOAP uses the encoding of the request message to encode the response message (MIME → MIME, DIME → DIME). In some cases this behavior is not appropriated. There are two servlet parameter to control the response encoding:

- `encodeSOAP`: messages without attachments. Allowed values: XML, MIME, DIME
- `encodeAttachment`: messages with attachments. Allowed values: MIME, DIME

```
<init-param>
  <param-name>encodeSOAP</param-name>
  <param-value>MIME</param-value>
</init-param>
<init-param>
  <param-name>encodeAttachment</param-name>
  <param-value>DIME</param-value>
</init-param>
```

2.3.3. Stateful webservices

Stateful webservices are not standardised. You can use them with Spheon JSOAP, but you have to build your own session management. This could be realized for example with an information in the SOAP header or using the servlet session handling functions.

Caution: Not all SOAP clients can use such a webservice!

To support this `SoapParser.execute()` and `SoapParser.executeRPC()` accept the webservice object as a third parameter. Your session management is responsible to pass the right object.

2.4. How to write a serializer/deserializer

A serializer transforms a java type into a SOAP type, a deserializer vice versa. Spheon JSOAP supports primitive types, structs and beans by default (see section A). In most cases it is not necessary to write your own serializer or deserializer. Anyway, if you want to write your own serializer/deserializer, you need some knowledge about the SOAP protocol!

Your own serializer or deserializer has to be derived from `de.fmui.spheon.jsoap.AbstractEncoding`. It consists of the following methods:

`getType()`

This method must return the serialized java class.

`getSoapType()`

This method must return the name of the SOAP type.

`getXSDdef()`

This method is needed to create WSDL files. If you don't want to create a WSDL file return an empty string. Otherwise read section 2.4.1.

`getValue()`

This method takes an `entry` object and must return an object of the java class returned by `getType()`. Use the methods `getValue()` and `getChildren()` of the `entry` object to get the content.

`getEntry()`

This method takes a java object and must return an `entry` object. It is important to set the `xsi:type` attribute correctly.

See section 2.6.3 for Information about configuring your webservices to use your serializer or deserializer.

2.4.1. WSDL

The `getXSDdef()` method should return a string which describes your SOAP type as defined in the WSDL specification in section “2.2 Types”.

See <http://www.w3.org/TR/wsdl.html> and <http://www.w3.org/TR/xmlschema-1/> for detailed information.

2.5. Understanding SoapConfig

An object of the class `SoapConfig` is needed by all operations. It automatically loads the configuration of the built-in types, manages your type and service definitions and is responsible for the name resolution of types and services.

These methods are important (see javadoc documentation for a detailed description):

`addToConfig()`

Loads a configuration from a file (see section 2.6).

`setActor()`

Sets the actor for fault messages.

`addService()`

Adds a service to the configuration.

`addMethod()`

Adds a method to the configuration.

`addType()`

Adds a type to the configuration.

`addStructType()`

Adds a struct to the configuration.

`addBeanType()`

Adds a bean to the configuration.

2.6. Understanding the configuration file

The configuration files contain the connection between your webservice class, the webservice URN and the connection between your (complex) SOAP types, your java type classes and the namespace. It is possible to omit a configuration file and directly use the the methods of the `SoapConfig` class (see section 2.5), but it is not recommended.

The configuration files are XML files with the root element `<spheon-jsoap>` `</spheon-jsoap>`.

2.6.1. Services

A service declaration connects the URN of the webservice with a Java class. To do that insert the following line into your configuration file and replace the attributes `urn` and `class` with your data:

Listing 2.10: Defining services

```
1 <service urn="urn:mySampleService" class="samples.SampleService" />
```

2.6.2. Methods

Spheon JSOAP uses the Java Reflection API to invoke a method. It takes the parameters from the SOAP message, deserializes them and pass them in the same order to the Java method. This works well if the SOAP message includes type information. In this case no further configuration is necessary.

If your webservice should be compatible with SOAP client implementations that doesn't send type information, you have to define the types in the configuration file. There are some SOAP implementations which doesn't care about the order of the parameters but use the name of the parameter to identify it. These must also be defined.

To declare a method embed the following XML fragment within the proper `<service>` tag:

Listing 2.11: Defining methods (parameter order)

```

1 <method name="sampleMethod" paramnames="0">
2     <param name="mystring">java.lang.String</param>
3     <param name="count">int</param>
4 </method>
```

The name attribute of the `<method>` tag identifies the method. The `paramnames` attribute can be "0" (the order is important) or "1" (the names are important). The `<param>` tags declare the parameters of the method and must be in the same order as in the signature of the method. The name attribute sets the name of the parameter, the tag value must be the type of the parameter. Use the whole package names with all classes!

2.6.3. Types

To declare a new type to Spheon JSOAP add a `<xsd>` tag to the configuration file. It needs an attribute `uri` which defines the namespace of the included types. Embed into the `<xsd>` tag `<soap>` and `<java>` tags. A `<soap>` tag defines the decoding class for the SOAP type specified by the `type` attribute. A `<java>` tag defines the encoding class for the Java type specified by the `type` attribute.

In the majority of cases you don't need to write your own serializer and/or deserializer (see section 2.4 for more information). See section A for built-in types. If you want to define a class as a struct, set `"@de.fmui.spheon.jsoap.encoding.xsdStructWrapper,your_SOAP_type,your_java_class"` as serializer and deserializer. Substitute "your_SOAP_type" and "your_java_class" with the correct values.

Listing 2.12: Defining types

```

1 <xsd uri="http://mytypes.company.com">
2     <soap type="myStruct">@de.fmui.spheon.jsoap.encoding.xsdStructWrapper,
3         myStruct,samples.SampleStruct</soap>
4     <java type="samples.SampleStruct">@de.fmui.spheon.jsoap.encoding.
5         xsdStructWrapper,myStruct,samples.SampleStruct</java>
6     <soap type="myStruct2">@de.fmui.spheon.jsoap.encoding.xsdStructWrapper,
7         myStruct2,samples.SampleStruct2</soap>
8     <java type="samples.SampleStruct2">@de.fmui.spheon.jsoap.encoding.
9         xsdStructWrapper,myStruct2,samples.SampleStruct2</java>
10 </xsd>
```

2.6.4. Attachments

Spheon JSOAP is able to understand attachments with the SOAP messages. The Content-Type of those attachments controls the mapping to Java types. The standard mappings are:

MIME type	Java type
text/plain	java.lang.String
text/html	java.lang.String
text/xml	java.lang.String
image/gif	byte[]
image/jpeg	byte[]
application/octet-stream	byte[]
<i>other</i>	byte[]

To add a new MIME type you have to write a new class derived from `de.fmui.spheon.jsoap.transport.AttachmentDecoder` and connect it to the MIME type like this:

Listing 2.13: Defining MIME types

```

1 <attachments>
2   <mime type="text/sgml">de.fmui.spheon.jsoap.transport.AttachmentText</
   mime>
3   <mime type="image/png">de.fmui.spheon.jsoap.transport.AttachmentBinary
   </mime>
4   <mime type="application/x-your-subtype">your.own.Attachment.
   decoderClass</mime>
5 </attachments>
```

There are two such classes which you can use:

```
de.fmui.spheon.jsoap.transport.AttachmentText
    Converts attachments to java.lang.String.
```

```
de.fmui.spheon.jsoap.transport.AttachmentBinary
    Converts attachments to byte[ ].
```

2.6.5. XML parser

By default Spheon JSOAP uses a DOM XML parser to parse the SOAP messages. To use a SAX XML parser insert `<parser type="SAX" driver="my.sax.driver" />` into your configuration file and replace `my.sax.driver` with the class name of your favorite SAX parser.

2.6.6. Imports

It is possible to import other configuration files.

Relative to the origin file: `<import file="../ws2.xml" />`

Absolute file names: `<import path="/home/jsoap/webservices/ws2.xml" />`

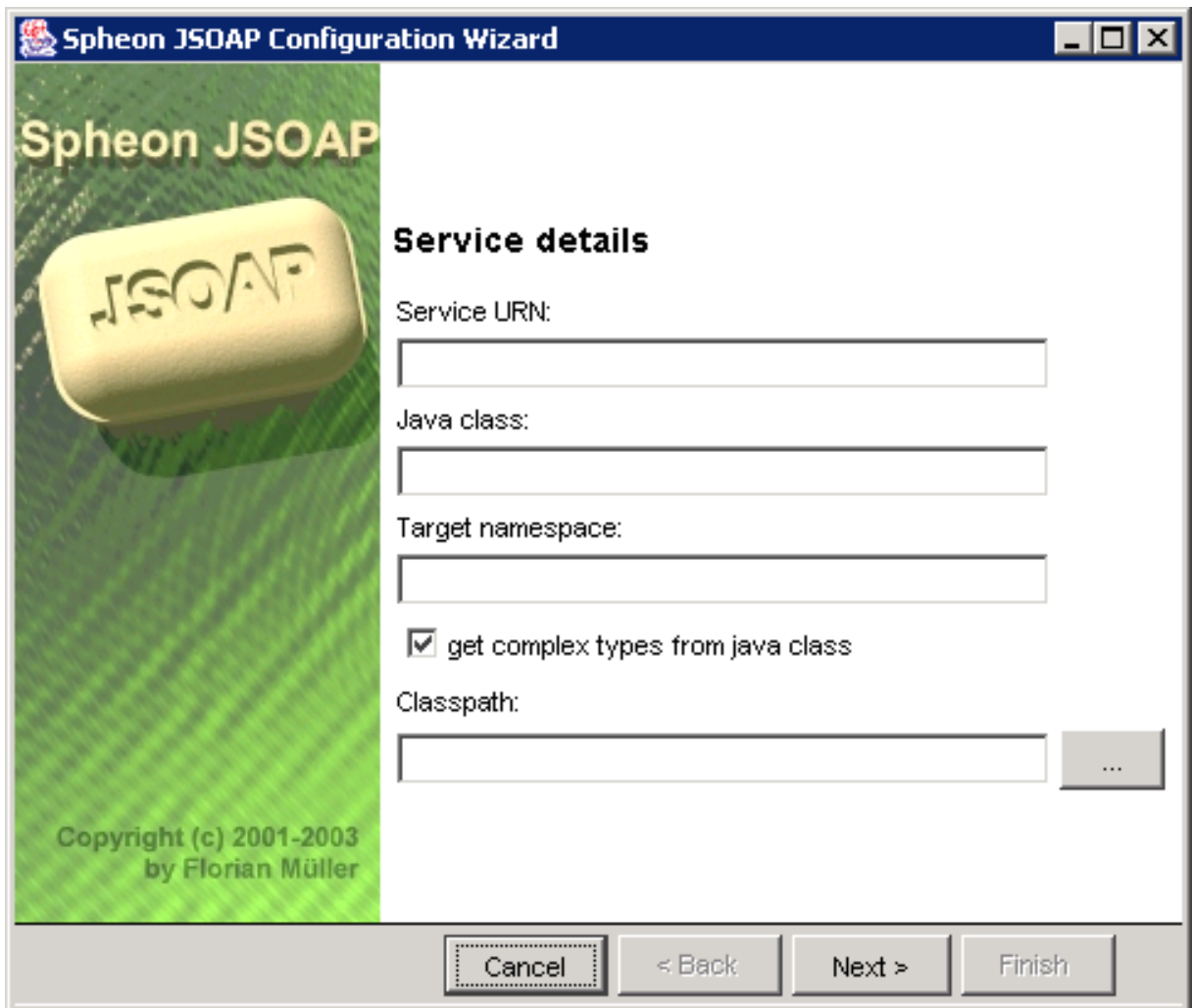
3. Tools

3.1. ConfigWizard

Caution: This is a beta version in the current state!

The ConfigWizard should help you to create a configuration file for Spheon JSOAP. It analyzes your webservice class and suggests a configuration.

```
java -classpath spheon-jsoap.jar:spheon-jsoap-tools.jar:<XMLParser>  
de.fmui.spheon.jsoap.tools.ConfigWizard
```



The screenshot shows a Windows-style dialog box titled "Spheon JSOAP Configuration Wizard". On the left side, there is a green textured panel with the text "Spheon JSOAP" at the top and a 3D rendering of a yellow soap bar with "JSOAP" embossed on it. Below the soap bar, it says "Copyright (c) 2001-2003 by Florian Müller".

The main area of the dialog is titled "Service details" and contains the following fields and controls:

- Service URN: [Text input field]
- Java class: [Text input field]
- Target namespace: [Text input field]
- get complex types from java class
- Classpath: [Text input field] with a browse button (...)

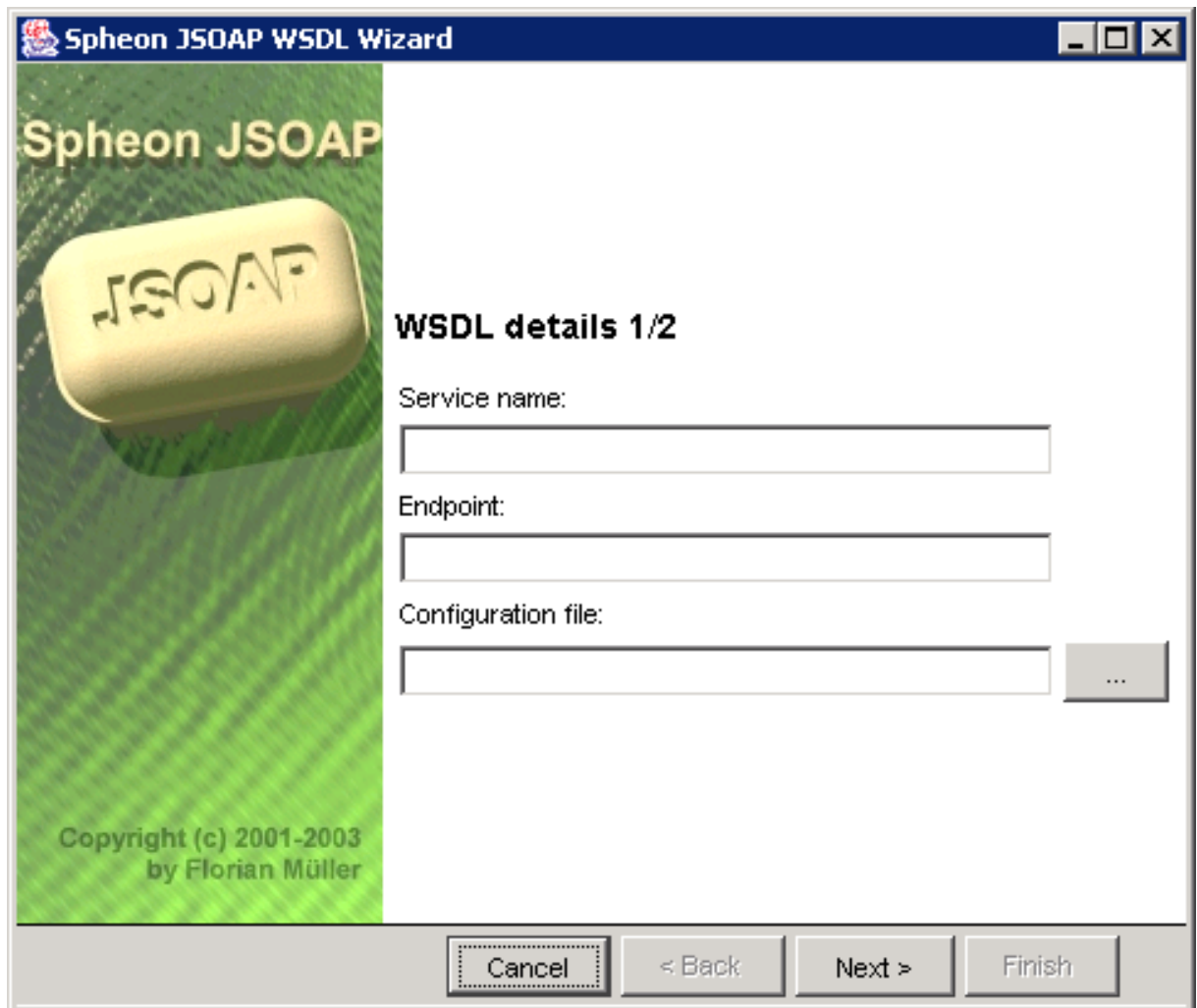
At the bottom of the dialog, there are four buttons: "Cancel", "< Back", "Next >", and "Finish".

3.2. WSDLWizard

Caution: This is a beta version in the current state!

The WSDL Wizard generates a WSDL file from your webservice class and your configuration file.

```
java -classpath spheon-jsoap.jar:spheon-jsoap-tools.jar:<XMLParser>  
de.fmui.spheon.jsoap.tools.WSDLWizard
```



4. Using Spheon JSOAP in applets

To be compliant with the majority of browsers, applets should run in a Java 1.1 environment. Spheon JSOAP comes with a limited applet version that is compiled with Java 1.1 (`spheon-jsoap-applet.jar`). With this version you are not able to connect to a SSL wrapped service and you cannot use the high level client.

Since Spheon JSOAP originally was not designed to run with Java 1.1, it needs additional classes from the [JDK1.1 Collections package](#). This package contains all collections classes that are introduced in Java 1.2. Spheon JSOAP just need this classes from the package:

```
com.sun.java.util.collections.Map
com.sun.java.util.collections.Set
com.sun.java.util.collections.List
com.sun.java.util.collections.AbstractSet
com.sun.java.util.collections.HashMap
com.sun.java.util.collections.ArrayList
com.sun.java.util.collections.Iterator
com.sun.java.util.collections.Vector
com.sun.java.util.collections.UnsupportedOperationException
```

4.1. Old browsers

Some old browsers (e.g. Netscape 4.x) do not allow to read data files from archives. In this case every SOAP call must fail because Spheon JSOAP cannot read the configuration file from `spheon-jsoap-applet.jar`.

Here is a workaround for this problem:

Extract the file `jsoapconfig.xml` from `spheon-jsoap-applet.jar`, place it beside your applet on the server and use the following code snippet (listing 4.1):

Listing 4.1: Old browser workaround

```
1 private void loadJSOAPConfig(SoapConfig sc) {
2     // check, if the configuration file is already loaded
3     if(!sc.getXsdMap().isEmpty()) {
4         return;
5     }
6
7     // Ok, it's an old browser.
8     // Load the configuration file from URL.
9     try {
10        URL configUrl = new URL(getDocumentBase(), "jsoapconfig.xml");
```

4. Using Spheon JSOAP in applets

```
11
12     InputStream configStream = configUrl.openStream();
13     sc.addToConfig(configStream);
14     configStream.close();
15 }
16 catch(Exception e) {
17     System.err.println("Cannot load Spheon JSOAP config: " + e);
18 }
19 }
```

A. Standard type mapping

SOAP type	Java type	encoding	decoding
string	java.lang.String	•	•
int	int	•	•
int	java.lang.Integer	•	
integer	int		•
long	long	•	•
long	java.lang.Long	•	
short	short	•	•
short	java.lang.Short	•	
byte	byte	•	•
byte	java.lang.Byte	•	
float	float	•	•
float	java.lang.Float	•	
double	double	•	•
double	java.lang.Double	•	
decimal	java.math.BigDecimal	•	•
boolean	boolean	•	•
boolean	java.lang.Boolean	•	
timeInstant	java.util.Date		•
dateTime	java.util.Date	•	•
date	java.util.GregorianCalendar	•	•
anyType	java.lang.Object	•	•
base64	byte[]	•	•
base64Binary	byte[]		•
hexBinary	java.lang.Byte[]	•	•
Map	java.util.Hashtable	•	•
Vector	java.util.Vector	•	•

B. Troubleshooting

Problem	Comments
Struct serialization doesn't work.	All fields have to be <code>public</code> .
My webservice method will not be found.	Check the signature of your method, especially if you use arrays. All array parameters has to be <code>SOAPArray</code> ! See section 2.1.3 for details.
I get an exception like this: <code>de.fumi.spheon.jsoap.SoapException</code> [Could not encode Parameter 'xyz': <code>java.lang.NullPointerException</code>]	Make sure, that the configuration file <code>jsoapconfig.xml</code> is in the <code>CLASSPATH</code> . If you use Spheon JSOAP in an applet, see section 4.1 .

C. License

Spheon JSOAP

Copyright (c) 2001-2003 Florian Mueller
All rights reserved.

Florian Mueller <mueller@gotux.de>

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. Neither the name of Florian Mueller nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.