Service Oriented Architecture using JAVA
on NetBeans and GlassFish 3

By Eduardo Cavasotti
4/20/10
# Table of Contents

Abstract: ................................................................................................................................. 3

Introduction: ............................................................................................................................ 3

Tools: ......................................................................................................................................... 4

Getting ready ............................................................................................................................ 4

Web Service Definition Language (WSDL) .................................................................................. 4

  WSDL Example ....................................................................................................................... 5

  WSDL Example 1: SOAP 1.1 Request/Response via HTTP ....................................................... 5

  WSDL Example 2. SOAP binding of request-response RPC operation over HTTP ................. 8

Creating a Web Service (Client-side using Server WSDL) .......................................................... 10

Creating a Web Service (Server-side) .......................................................................................... 17
Abstract:

This report is focused on creating a SOA Web Service application using Java. I assume that the reader has a little background in SOA and Java, so I will skip some details and go straight to the point. My focus will be on how to actually create a Web Service application other than having multiple pages on theory we are overwhelmed with on the internet.

Introduction:

Service Oriented Architecture (SOA) is basically a collection of services that communicate with each other. However, in order to build a concise and effective SOA we need to clearly understand the real meaning of the term service. “A service is a function that is well-defined, self-contained, and does not depend on the context or state of other services. Services are what you connect using Web Services. A service is the endpoint of a connection.” (service-architecture.com)

Java has become a powerful development platform for SOA in 2006 when Java Enterprise Edition (EE) 5 and Java Standard Edition (SE) 6 significantly enhanced the power and usability of Web Services capabilities on the application server and then incorporated the majority of these capabilities into the SE of Java. With the latest Java Web Services (JWS) APIs developers will find it much easier to build Web Services applications in Java SE 6 than in previous versions of Java. What is brilliant in the JWS APIs is the power and ease of use as we will see in this report; one can create a working powerful Web Service in seconds thanks to the new APIs.

In this report I will show how to create Web Services in NetBeans and deploying them using GlassFish. I will explain in detail every step on the creation of each application. I will not, at this time, get into creating the XML files since we will be using a “Start from Java” approach and let NetBeans deal with the XMLs.
Tools:

The following tools were used in the development of all examples demonstrated in this report:

- Java JDK 1.6
- NetBeans IDE 6.8
- GlassFish 3

Getting ready

For some reason NetBeans support for SOA has been dropped on this new release along with UML and Visual Web Development. However, we can still download the plug-in and install it.

Here is the instruction on how to do this: go in Plugins ->Settings -> Add and add the following url:

http://updates.netbeans.org/netbeans/updates/6.8/uc/m1/dev/catalog.xml

This will add the old catalog with the list of all the plug-ins that aren’t available on the new one. Select and Install the SOA plug-in.

Web Service Definition Language (WSDL)

NetBeans makes it really easy and simple to implement a Client for a Web Service based on its Web Service Definition Language (WSDL often pronounced “whiz-dull”). W3C defines WSDL as “WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or
network protocols are used to communicate, however, the only bindings described in this document describe how to use WSDL in conjunction with SOAP 1.1, HTTP GET/POST” (w3.org) It basically tells the Client or whoever wants to access a Web Service, what functionality is available in that Service, what methods and types should be used in order to get server and client to talk. In Java words, it states the functions from the server classes that were made available to the public via Web Service tags. We will see those later. We will not, at this point of the project, look into building WSDL; we will take Starting from Java approach and let NetBeans create the WSDL for us.

**WSDL Example**

The following example shows the WSDL definition of a simple service providing stock quotes. The service supports a single operation called GetLastTradePrice, which is deployed using the SOAP 1.1 protocol over HTTP. The request takes a ticker symbol of type string, and returns the price as a float. A detailed description of the elements used in this definition can be found in Section 2 (core language) and Section 3 (SOAP binding).

This example uses a fixed XML format instead of the SOAP encoding (for an example using the SOAP encoding, see the next example).

**WSDL Example 1: SOAP 1.1 Request/Response via HTTP**

```xml
<?xml version="1.0"?>
<definitions name="StockQuote"

targetNamespace="http://example.com/stockquote.wsdl"
xmlns:tns="http://example.com/stockquote.wsdl"
```
<xsd:element name="TradePriceRequest">
    <xsd:complexType>
        <xsd:all>
            <xsd:element name="tickerSymbol" type="xsd:string"/>
        </xsd:all>
    </xsd:complexType>
</xsd:element>

<xsd:element name="TradePrice">
    <xsd:complexType>
        <xsd:all>
            <xsd:element name="price" type="xsd:float"/>
        </xsd:all>
    </xsd:complexType>
</xsd:element>

<message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
</message>

<message name="GetLastTradePriceOutput"/>
<part name="body" element="xsd1:TradePrice"/>

</message>

<portType name="StockQuotePortType">
  <operation name="GetLastTradePrice">
    <input message="tns:GetLastTradePriceInput"/>
    <output message="tns:GetLastTradePriceOutput"/>
  </operation>
</portType>

<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http" />
  <operation name="GetLastTradePrice">
    <soap:operation
      soapAction="http://example.com/GetLastTradePrice" />
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>

<service name="StockQuoteService">
  <documentation>My first service</documentation>
  <port name="StockQuotePort" binding="tns:StockQuoteBinding">
    <soap:address location="http://example.com/stockquote"/>
  </port>
</service>
This example describes that a GetTradePrice SOAP 1.1 request may be sent to a StockQuote service via the SOAP 1.1 HTTP binding. The request takes a ticker symbol of type string, a time of type timeInstant, and returns the price as a float in the SOAP response.

**WSDL Example 2. SOAP binding of request-response RPC operation over HTTP**

```xml
<?xml version="1.0"?>
<definitions name="StockQuote"

targetNamespace="http://example.com/stockquote.wsdl"
xmns:tns="http://example.com/stockquote.wsdl"
xmns:xsd="http://www.w3.org/2000/10/XMLSchema"
xmns:xsd1="http://example.com/stockquote.xsd"
xmns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmns="http://schemas.xmlsoap.org/wsdl/">

<message name="GetTradePriceInput">
  <part name="tickerSymbol" element="xsd:string"/>
  <part name="time" element="xsd:timeInstant"/>
</message>

<message name="GetTradePriceOutput">
  <part name="result" type="xsd:float"/>
</message>
```
<portType name="StockQuotePortType">
  <operation name="GetTradePrice">
    <input message="tns:GetTradePriceInput"/>
    <output message="tns:GetTradePriceOutput"/>
  </operation>
</portType>

<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
  <soap:binding style="rpc"
  transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="GetTradePrice">
    <soap:operation soapAction="http://example.com/GetTradePrice"/>
    <input>
      <soap:body use="encoded"
      namespace="http://example.com/stockquote"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output>
      <soap:body use="encoded"
      namespace="http://example.com/stockquote"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
</binding>
The example I found to be easy to implement and also to demonstrate this process of creating a client to a server that is already up and running out there was to use a currency exchange Web Service. If you look it up online there are many Web Services running to the public with simple but useful functionality. One can use their WSDL to implement a custom Client for it; a java application or a web page. NetBeans will help us in both cases, but for now we will focus on the java application.

**Creating a Web Service (Client-side using Server WSDL)**

First we Create a new Project from Java/Java Application:
Lets name it CurrencyApp and we don’t need to have a Main class created, so uncheck that:
This is the WSDL we are going to use:  
http://www.webservicex.net/CurrencyConvertor.asmx?WSDL

Now we are going to create the Web Service based on the WSDL. Right click on the Project CurrencyApp then click on “New” then either “Web Service Client” or “Other” to get a window like this:

![New File window](image)

By clicking Next it will give you this screen:
Note I have selected WSDL URL and I have pasted the WSDL URL from the CurrencyConvertor Web Service. It is very important to select a package name for the Web Service because it is based on that that we will be able to import the Web Service’s classes that NetBeans will generate from this WSDL. I picked “CEWebService” where CE stands for Currency Exchange. Click Finish. You will note on the Output window that NetBeans generates all the classes after we clicked on Finish. Feel free to look over those classes.
You can see on the Project bar all the generated classes inside our Package along with all the Web Service References which are all the endpoints from that Web Services (all the functions available).

We are now going to create a java class to handle the operation we need. In this case I have a GUI Application that gets all Currencies available in the Web Service. What I did was create a GUI that looks like this:
I populate each JComboBox with the possible Currencies the Web Service provides us (See generated Currency.java for details on all the symbols.) with the following code:

```java
private void formWindowOpened(java.awt.event.WindowEvent evt) {
    this.currencyFromBox.setModel(new DefaultComboBoxModel(Currency.values()));
    this.currencyToBox.setModel(new DefaultComboBoxModel(Currency.values()));
    this.currencyFromBox.setSelectedItem(Currency.USD);
    this.currencyToBox.setSelectedItem(Currency.BRL);
}
```

Note that I have USD and BRL which are American Dollar and Brazilian Real respectively as the default currency to appear on the JComboBox. But the big trick is on the “Get Rate!” JButton event which makes all the calls to the Web Service. In order to do that, go to your class right where you want the code to be (line 132):

```java
this.currencyToBox.setSelectedItem(Currency.BRL);
```
The browse through the Project area over Web Service References:

Drag and Drop the reference to ConversionRate into the code at line 132:

```java
private void goB tnActionPerformed(ActionEvent evt) {
    try { // Call Web Service Operation
        CEWebService.CurrencyConverter service = new CEWebService.CurrencyConverter();
        CEWebService.CurrencyConverterSoap port = service.getCurrenc yConverterSoap();
        // TODO initialize WS operation arguments here
        CEWebService.Currency fromCurrency = null;
        CEWebService.Currency toCurrency = null;
        // TODO process result here
        double result = port.conversionRate(fromCurrency, toCurrency);
        System.out.println("Result = " + result);
    } catch (Exception ex) {
        // TODO handle custom exceptions here
    }
}
```

And this is the result we get. Note that we have to use the package (`CEWebService`) we created when creating the web service in order to find the classes. We could; however, import the package and not have to worry about it, but since we are just making a few calls to it and I wanted to show them on the code I left it like that. The object service is created and the port which is how we will access the functionality of the service is also created. We can call every method of the service by calling the object port and the method name. However, we need to make some changes to it so it matches our GUI and give us the result we need:
And the final result looks somewhat similar to this:

![Currency Exchange Result](image)

**Creating a Web Service (Server-side)**

I am going to create a simple Math Operations (Addition, Subtraction, Division and Multiplication) Web Service Server using *Start from Java* approach and apply the appropriate Web Services tags. NetBeans will then use these tags to create the WSDL for our WebService. We start by creating a new Web Service Project:
Click Next > and give it a name. In this case I named it *MathWebService*:

Click Next > again and make sure *Personal GlassFish v3 Domain* is selected as the server and *Java EE 6 Web* as the Java EE Version. Then click Finish.
NetBeast will automatically open the index.jsp, but we don’t want that for now cause we are going to create the java class with the Web Service functionality, so close the index.jsp.

Right click the java folder inside the MathWebService folder and select New -> Java Class:
Name it *Calculator* and create the Package *calcpkg*:

![Image of Name and Location dialog box]

Ok, now we have an empty Calculator.java class ready for implementation of our Web Service functionality. So, now we are going to implement our calculator just like a regular java class. After implementing the simple operations of our calculator, our class should look somewhat like this:

```java
public class Calculator {
    public double add(double a, double b){
        return a + b;
    }

    public double subtract(double a, double b){
        return a - b;
    }

    public double divide(double a, double b){
        return a / b;
    }

    public double multiply(double a, double b){
        return a * b;
    }
}
```

As of now we have no Web Service functionality of course, but we want to export these methods as web services. We will need two things in order to export this class and methods as a Web Service, an import statement for the `javax.jws.WebService` package and a `@WebService` annotation at the beginning that tells the Java interpreter that you intend to publish the methods of this class as a web service.
```
6    package calcpkg;
7
8    import javax.jws.WebService;
9
10   @WebService
11    public class Calculator {
```

Only with the `@WebService` annotation we are already able to deploy and test it. Click the Projects tab, right click MathWebService and click Deploy.

Now unfold WebServices, right click on Calculator and click on “Test WebService”. 
And there we have all our methods and the spaces to input the parameters. Input 4 and 5 into the add method and click add.
We get the response with the result 9.0 as well as the two parameters and down below we can see the SOAP messages request and response:

**SOAP Request**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header/>
  <S:Body>
    <ns2:add xmlns:ns2="http://calcpkg/">
      <arg0>4.0</arg0>
      <arg1>5.0</arg1>
    </ns2:add>
  </S:Body>
</S:Envelope>
```
As we can see our class is not well documented as far as annotations go. We will now implement some more annotations in order to make our SOAP Message more readable by including the `@WebMethod` and `@WebParam` annotation as well as imports for the packages `javax.jws.WebMethod`, `javax.jws.WebParam` as follows:

```java
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

@WebService
public class Calculator {

    @WebMethod(operationName = "addition")
    public double add(@WebParam(name = "a") double a, @WebParam(name = "b") double b){
        return a + b;
    }
}
```

Now we will deploy the Web Service again, test it and see what the SOAP Envelope looks like. You should notice the button that before was named `add` now is called `addition` that is the result of the `@WebMethod` annotation and the SOAP Request now looks better:

```
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:addResponse xmlns:ns2="http://calc pkg/">
            <return>9.0</return>
        </ns2:addResponse>
    </S:Body>
</S:Envelope>
```
SOAP Request

```xml
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ns2="http://calcpkg/"
>
  <S:Header/>
  <S:Body>
    <ns2:addition>
      <a>4.0</a>
      <b>5.0</b>
    </ns2:addition>
  </S:Body>
</S:Envelope>
```

We can customize the parameters name and the method/function name as well. This will make it easier for the Client to read the SOAP message and makes our message more professional.