Converting Java and CORBA components to a WSDL Representation

Kevin Curley <kevin.curley@capeclear.com>

Abstract

WSDL (Web Services Description Language) has become the standard way to describe the interface to a particular Web Service. Here we will start off with an explanation of the main sections that make up a WSDL file and its importance in a Web Service deployment.

Many people have deployed enterprise applications using a variety of technologies, such as J2EE and CORBA. One of the steps for exposing these applications as Web Services is to provide a WSDL description of this service. We will firstly show and discuss how one can describe an Enterprise Java Bean in WSDL. We will then have a look at how some commercial tools deal with the problem.

For a more complex example, we will look at how one can describe a CORBA IDL interface in WSDL. We will look at the issues, around multiple objects in a single Web Service and techniques for mapping CORBA IDL inheritance hierarchies to WSDL.

Table of Contents

1. Introduction ................................................................. 1
2. From WSDL to SOAP ..................................................... 2
3. Mapping an EJB to WSDL ............................................... 3
4. Mapping a Set of CORBA Objects to WSDL ..................... 4
5. Some more advanced usage ........................................... 5
6. Finishing up ................................................................. 5
7. Appendix 1 ................................................................. 5
8. Appendix 2 ................................................................. 8

1. Introduction

Let's start by asking a few questions. To begin with, what is a Web Service? This is a question that has been asked a lot recently and if there is one indisputable answer, I haven't heard it yet. One answer is that a Web Service is a piece of business logic or software component that can be accessed using SOAP and has an interface defined in WSDL (Web Service Description Language). Additionally, you can make your Web Service available using UDDI (Universal Description, Discovery, and Integration).

I'm writing some business logic, what benefit do I get from making it a Web Service? The biggest advantage is that because it runs on HTTP and SMTP, it is now accessible to many more types of applications and machines. Almost every type of application has an HTTP library available to it, from Java to Microsoft Visual Basic to C++ to Perl. Additionally, your business logic becomes more available to clients that are resident on a different network. Furthermore, because it is being accessed using HTTP or SMTP, you don't need to worry (at the moment) about firewalls.

What if I've got some existing business logic that I want to deploy as a Web service? What if I have spent the last few years implementing business logic in CORBA, Java and EJB? Fortunately, there are many companies that provide a solution to this problem. The first requirement is a SOAP stack that converts SOAP requests into a native request to your legacy system and converts the reply into a SOAP response. You will also need a tool to represent your current system as WSDL so that clients can create the correct SOAP messages. Once you have your SOAP
stack/server running and your WSDL at a well-known web address or registered in a UDDI repository, it's straightforward for clients to access your service.

Let's look at an example. I have an Enterprise Java Bean (EJB) that provides access to customer records and I want to be able to access and manipulate the data in a Visual Basic client. Turning the logic into a Web Service seems like the best solution. I buy an off-the-shelf solution and create WSDL from my EJB. I then configure the server to connect to my EJB and start it. The guy building the VB application then imports the WSDL into his environment and using either .NET or the Microsoft SOAP Toolkit can easily create requests to my service. The following is a sample of his Visual Basic code:

```
set soapclient = CreateObject("MSSOAP.SoapClient")
Call soapclient.mssoapinit("http://localhost:8080/wSDL/records.wSDL", "records", "Records")
soapclient.getCustomerEmail("CapeClear")
```

About 3 lines of VB script and that's it. Altogether, it took about 10 minutes to get a VB client talking to my EJB. For many, they are happy that it works and has been extremely painless. Others, like me, would like to know how it all works. The rest of this paper concentrates on WSDL and how these tools convert EJB and CORBA interfaces into WSDL.

2. From WSDL to SOAP

Let's look at a SOAP request for the previous example.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ns0="capeconnect:records:Record"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
 <ns0:getCustomerContact>
  <arg0 xsi:type="xsd:string"></arg0>
 </ns0:getCustomerContact>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Let’s look at this message and see how it relates to the WSDL for the service. The main items of note are in the Binding section of the WSDL. Here is the relevant section from our WSDL.

```xml
<binding name="RecordBinding" type="tns:Record">
 <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
 <operation name="getCustomerContact">
  <soap:operation soapAction="capeconnect:records:Stock#getCustomerContact" />
  <input>
   <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="capeconnect:records:Stock"/>
  </input>
  <output>
   <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="capeconnect:records:Stock"/>
  </output>
 </operation>
```

The SOAP encoding style is taken from the encoding style attribute in the operation. A namespace is declared for the namespace attribute of the WSDL operation. The actual name of the operation is then the name of the operation node in the binding (the operation name in the binding and port have to be the same) and is in the namespace we just declared. If you go back from the binding to its port type and from there to the messages you will then see the arguments (or parts) of the operation, in this case, a string. Now that we have had a quick look at the parts of
WSDL and how they relate to each other, let's move on to seeing how one of the above mentioned tools would create one of these WSDL files from an EJB.

3. Mapping an EJB to WSDL

First let's look at a sample EJB.

```java
package xmleurope;

import javax.ejb.*;
import java.rmi.RemoteException;
import java.rmi.Remote;

public interface Record extends EJBObject
{
    public String getCustomerEmail( String company ) throws RemoteException;
    public Address getCustomerAddress( String company ) throws RemoteException;
    public Name getCustomerContact( String company ) throws RemoteException;
}
```

This is just a simple customer record with 3 methods. The Address and Name classes are just simple objects with getters and setters for String values. The generated WSDL is below in Appendix 1. Here is a quick look at the structure of Name and Address. The full classes are listed below in appendix 2.

```java
public class Name
{
    private String _firstName;
    private String _secondName;
}

public class Address
{
    private String _street1;
    private String _street2;
    private String _city;
    private String _state;
    private String _zipcode;
}
```

We'll examine the mapping in the order we discussed the WSDL basics earlier. The Types section is an XML schema definition for all the user-defined types in the service. In our example, we have two such types, Name and Address. You can see from the WSDL in appendix 1 that there are two new XML schema types called Name and Address. The actual make up of the XML schema definition is Web Service-platform specific. A particular platform will have its own way of representing the types it handles. While all parts of the generated WSDL are platform-specific, the sections of generated WSDL (other than the XML schema) are always very similar regardless of the platform or tool used to generate them.

The next generated items are the messages. A SOAP request and a SOAP reply are regarded as two messages in WSDL. If you look at our generated WSDL, we have six messages, two each for each method in our EJB interface. The parameters of the EJB methods are converted into parts of the message.

A port type is then generated for the EJB. Each of the EJB’s operations will have a corresponding operation in the port. The input and output section of the operation is mapped to the correct message. A corresponding binding section is also generated. Again, an operation is generated for each of the EJB’s operations. Finally a service is generated which lists the ports in the application, in our example the one port, Record.
4. Mapping a Set of CORBA Objects to WSDL

Mapping a set of CORBA object definitions in an IDL file to WSDL is very similar to completing the mapping from EJB to WSDL. However, some common IDL usage makes things a little more complicated. Specifically, I'm referring here to inheritance. If I have an interface Base and another interface Derived that inherits from Base and I want to represent them in WSDL, how do I do this? I have two options.

If I have only one servant on the back-end which implements Derived then one could group all the messages of both interfaces into one port. Alternatively, if you have servants implementing both Base and Derived, you need to create a port type for each and then have the actual port be of the type of both of them. Consider the following example:

```java
interface Base
{
    void doWork();
}

interface Derived : Base
{
    void doSomeOtherWork()
}
```

One option is to generate a port for each, Base and Derived. Then put the doWork operation in one and both doWork and doSomeOtherWork in the other. This works for the example but there is a large amount of replication with the base classes methods being in both ports.

A better strategy is to create two independent port types and bindings. Then, when declaring the actual port in the service elements, make the port of type both bindings. For example:

```xml
<service name="DerivedTest">
    <documentation>XML Europe Demo</documentation>
    <port binding="tns:BaseBinding, tns:DerivedBinding" name="DerivedTest">
        <soap:address location="http://ktulu:8000/ccx/DerivedTest"/>
    </port>
</service>
```

For one looks at a Java application where inheritance is used, this second approach will only work if the Java class does not overload any methods from the base class. If overloading does occur, then there will a serious problem regarding polymorphism. Another technique required for a CORBA mapping is including multiple objects in one Web Service. When mapping from EJB to WSDL, we have only had one EJB to map. CORBA applications are made up of multiple IDL interfaces. In this case, each IDL interface would have a binding and an associated port. All the operations of each interface will have messages in the WSDL. This pattern would be the same if one was generating WSDL for a Java application where multiple Java classes make up the section of the application to be exposed as a Web Service. The same is true if you are trying to create a WSDL file to represent multiple EJBs.

CORBA parameters can be of type out or inout. An out type is a variable that is initialized and set in a server and sent back to the client. An inout type is a type that is initialized in the client and set but possibly reset in the server. This makes things a bit more complicated in the WSDL for methods with these types.

If an operation has out parameters then these parameters appear as parts in the response message. If an operation has inout parameters then we will have a part with the same name in both the request and response message. In many cases the platform will need to know the order of parameters. For this reason, the operation node in the port has an attribute called parameterOrder. This attribute's value is a comma-separated list of parameters in the order they occur in the method signature.
5. Some more advanced usage

Method overloading is a feature commonly used in Java and EJB programming. In this case, I have one or more methods in my class with the same name but differing only by parameter type or parameter number. In WSDL, messages can’t have the same name. The same is true of operations in a particular port.

To address this issue, generate a unique message and operation name for each overloaded method. For this to work, the Web Services platform that converts the SOAP request in a method invocation would have to work a specific way. It would have to determine that the SOAP message is for a particular operation and not a request to a non-existent operation. This is a tricky problem for Web Services platforms to implement.

Another familiar problem, typically referred to as namespace pollution, is how to handle the case of having two types in a system that have the same name. C++ has namespaces to solve this problem and Java has packages. What can we do in WSDL to solve the problem?

The WSDL specification has no built in mechanisms to handle namespace pollution. One answer might be to use the correct name for one type and use some name mangling algorithm to create a name for the other clashing types. For example, suppose you have two Java types: one is xmleurope.Name and the other org.xmleurope.Name. You could use two XML schema types Name and Name2. The problem here is that programmers are usually careful about what they call types to allow the abstraction be known by the class name. Here you are distorting the abstraction by changing the name.

An alternative is to encode the package name into the XML schema type name. The problem here is that XML schema is supposed to be a language-independent type system. By encoding the Java package name you are slightly altering the XML schema ethos. Clients in languages other than Java might not appreciate these long type names. The best compromise is to use the package name encoding algorithm but only for types that are clashing and not all types. In our example, the two names in the XML schema would be XmleuropeName and OrgXmleuropeName.

6. Finishing up

We have taken a very brief look at Web Services and why you may want to use them. Then we looked at how we can easily convert an EJB to a Web Service and invoke it from a Visual Basic application using the Microsoft SOAP Toolkit. From here we had a look at WSDL and how these tools go about mapping EJBs to WSDL. We took a brief look at CORBA to demonstrate how we could deal with more complex issues. Finally, we looked at the problems posed by method overloading in Java. Hopefully the magic that these tools do has become less like magic and more a process that you now understand.

7. Appendix 1

Our example WSDL file.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions
  name="records"
  targetNamespace="http://www.capeclear.com/records.wsdl"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.capeclear.com/records.wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd1="http://www.capeclear.com/records.xsd">
<types>
  <xsd:schema
    targetNamespace="http://www.capeclear.com/records.xsd"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

(Rendered by www.RenderX.com)
<xsd:complexType name="Name">
 <xsd:sequence>
  <xsd:element maxOccurs="1" minOccurs="1"
   name="firstName"
   nillable="true"
   type="xsd:string"/>
  <xsd:element maxOccurs="1" minOccurs="1"
   name="secondName"
   nillable="true"
   type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Address">
 <xsd:sequence>
  <xsd:element maxOccurs="1" minOccurs="1"
   name="street1"
   nillable="true"
   type="xsd:string"/>
  <xsd:element maxOccurs="1" minOccurs="1"
   name="street2"
   nillable="true"
   type="xsd:string"/>
  <xsd:element maxOccurs="1" minOccurs="1"
   name="city"
   nillable="true"
   type="xsd:string"/>
  <xsd:element maxOccurs="1" minOccurs="1"
   name="state"
   nillable="true"
   type="xsd:string"/>
  <xsd:element maxOccurs="1" minOccurs="1"
   name="zipcode"
   nillable="true"
   type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>

</xsd:schema>

<message name="getCustomerAddress">
 <part name="company" type="xsd:string"/>
</message>

<message name="getCustomerAddressResponse">
 <part name="return" type="xsd1:Address"/>
</message>

<message name="getCustomerContact">
 <part name="company" type="xsd:string"/>
</message>

<message name="getCustomerContactResponse">
 <part name="return" type="xsd1:Name"/>
</message>
<message name="getCustomerEmail">
  <part name="company" type="xsd:string"/>
</message>
<message name="getCustomerEmailResponse">
  <part name="return" type="xsd:string"/>
</message>

<portType name="Record">
  <operation name="getCustomerAddress">
    <input message="tns:getCustomerAddress"/>
    <output message="tns:getCustomerAddressResponse"/>
  </operation>
  <operation name="getCustomerContact">
    <input message="tns:getCustomerContact"/>
    <output message="tns:getCustomerContactResponse"/>
  </operation>
  <operation name="getCustomerEmail">
    <input message="tns:getCustomerEmail"/>
    <output message="tns:getCustomerEmailResponse"/>
  </operation>
</portType>

.binding name="RecordBinding" type="tns:Record">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="getCustomerAddress">
    <soap:operation soapAction="capeconnect:records:Stock#getCustomerAddress"/>
    <input>
      <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="capeconnect:records:Stock"
        use="encoded"/>
    </input>
    <output>
      <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="capeconnect:records:Stock"
        use="encoded"/>
    </output>
  </operation>
  <operation name="getCustomerContact">
    <soap:operation soapAction="capeconnect:records:Stock#getCustomerContact"/>
    <input>
      <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="capeconnect:records:Stock"
        use="encoded"/>
    </input>
    <output>
      <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="capeconnect:records:Stock"
        use="encoded"/>
    </output>
  </operation>
  <operation name="getCustomerEmail">
    <soap:operation soapAction="capeconnect:records:Stock#getCustomerEmail"/>
    <input>
      <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="capeconnect:records:Stock"
        use="encoded"/>
    </input>
    <output>
      <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="capeconnect:records:Stock"
        use="encoded"/>
    </output>
  </operation>
</binding>
8. Appendix 2

Here are the example classes for the EJB Example

```java
package xmleurope;

import javax.ejb.*;
import java.rmi.RemoteException;
import java.rmi.Remote;

public interface Record extends EJBObject {
    public String getCustomerEmail( String company ) throws RemoteException;
    public Address getCustomerAddress( String company ) throws RemoteException;
    public Name getCustomerContact( String company ) throws RemoteException;
}

package xmleurope;

public class Name {
    public Name( String firstName, String secondName ) {
        this._firstName = firstName;
        this._secondName = secondName;
    }

    public String getFirstName() {
        return _firstName;
    }

    public String getSecondName() {
        return _secondName;
    }

    private String _firstName;
    private String _secondName;
}

package xmleurope;
```

public class Address
{
    public Address( String street1,
                     String street2,
                     String city,
                     String state,
                     String zipcode)
    {
        this._street1 = street1;
        this._street2 = street2;
        this._city   = city;
        this._state  = state;
        this._zipcode = zipcode;
    }

    public String getStreet1()
    {
        return _street1;
    }

    public String getStreet2()
    {
        return _street2;
    }

    public String getCity()
    {
        return _city;
    }

    public String getState()
    {
        return _state;
    }

    public String getZipCode()
    {
        return _zipcode;
    }

    private String _street1;
    private String _street2;
    private String _city;
    private String _state;
    private String _zipcode;
}

**Biography**

Tuesday, 21 May 16.45

Kevin Curley
Senior Software Engineer
Cape Clear Software Inc
Dublin
Ireland
Email: kevin.curley@capeclear.com

Kevin is a Senior Software Engineer at Cape Clear, the Web Services Software Company, where he has been developing various tools for creating WSDL from back end systems, such as J2EE and CORBA. He previously
worked for IONA Technologies, working on various middleware projects. While in IONA, he worked on the JMX Expert Group and set up JSR 70.

Kevin previously spoke at Java One in 1999 about JMX and will be presenting again this year at Java One at the Web Services Track on Composition of Web Services using WSDL. Kevin is on the XML Ireland Special Interest Group committee.