IBM

Which style of WSDL should I use?

Level: Advanced

Russell Butek (butek@us.ibm.com), SOA and Web services consultant, IBM

31 Oct 2003
Updated 24 May 2005

A Web Services Description Language (WSDL) binding style can be RPC or document. The use can be encoded or literal. How do you determine which combination of style and use to use? The author describes the WSDL and SOAP messages for each combination to help you decide.

Introduction

A WSDL document describes a Web service. A WSDL binding describes how the service is bound to a messaging protocol, particularly the SOAP messaging protocol. A WSDL SOAP binding can be either a Remote Procedure Call (RPC) style binding or a document style binding. A SOAP binding can also have an encoded use or a literal use. This gives you four style/use models:

1. RPC/encoded
2. RPC/literal
3. Document/encoded
4. Document/literal

Add to this collection a pattern which is commonly called the document/literal wrapped pattern and you have five binding styles to choose from when creating a WSDL file. Which one should you choose?

Before I go any further, let me clear up some confusion that many of us have stumbled over. The terminology here is very unfortunate: RPC versus document. These terms imply that the RPC style should be used for RPC programming models and that the document style should be used for document or messaging programming models. That is *not* the case at all. The style has nothing to do with a programming model. It merely dictates how to translate a WSDL binding to a SOAP message. Nothing more. You can use either style with any programming model.

Likewise, the terms *encoded* and *literal* are only meaningful for the WSDL-to-SOAP mapping, though, at least here, the traditional meanings of the words make a bit more sense.

For this discussion, let's start with the Java method in Listing 1 and apply the JAX-RPC Java-to-WSDL rules to it (see Resources for the JAX-RPC 1.1 specification).

Listing 1. Java method

```
public void myMethod(int x, float y);
```

RPC/encoded

Take the method in Listing 1 and run it through your favorite Java-to-WSDL tool, specifying that you want it to generate RPC/encoded WSDL. You should end up with something like the WSDL snippet in Listing 2.

Listing 2. RPC/encoded WSDL for myMethod

```
<message name="myMethodRequest">
    <part name="x" type="xsd:int"/>
    <part name="y" type="xsd:float"/>
</message>
<message name="empty"/>

<portType name="PT">
    <operation name="myMethod">
        <input message="myMethodRequest"/>
        <output message="empty"/>
    </operation>
</portType>

<binding .../>
<!-- I won't bother with the details, just assume it's RPC/encoded. -->
```

Now invoke this method with "5" as the value for parameter `x` and "5.0" for parameter `y`. That sends a SOAP message which looks something like Listing 3.

Listing 3. RPC/encoded SOAP message for myMethod

```
<soap:envelope>
    <soap:body>
        <myMethod>
            <x xsi:type="xsd:int">5</x>
            <y xsi:type="xsd:float">5.0</y>
        </myMethod>
    </soap:body>
</soap:envelope>
```

There are a number of things to notice about the WSDL and SOAP message for this RPC/encoded example:

Strengths

- The WSDL is about as straightforward as it's possible for WSDL to be.
- The operation name appears in the message, so the receiver has an easy time dispatching this message to the implementation of the operation.

A note about prefixes and namespaces

For the most part, for brevity, I ignore namespaces and prefixes in the listings in this article. I do use a few prefixes that you can assume are defined with the following namespaces:

- xmlns:xsd="http://www.w3.org/2001/XMLSchema"
- xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
- xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"

For a discussion about namespaces and the WSDL-to-SOAP mapping, see paper "Handle namespaces in SOAP messages you create by hand" (see Resources).

Weaknesses

- The type encoding info (such as `xsi:type="xsd:int"`) is usually just overhead which degrades throughput performance.
- You cannot easily validate this message since only the `<x ...>5</x>` and `<y ...>5.0</y>` lines contain things defined in a schema; the rest of the `soap:body` contents comes from WSDL definitions.
- Although it is legal WSDL, RPC/encoded is not WS-I compliant.

Is there a way to keep the strengths and remove the weaknesses? Possibly. Let's look at the RPC/literal style.

WS-I compliance

The various Web services specifications are sometimes inconsistent and unclear. The WS-I organization was formed to clear up the issues with the specs. It has defined a number of profiles which dictate how you should write your Web services to be interoperable. For more information on WS-I, see the WS-I links in Resources.

RPC/literal

The RPC/literal WSDL for this method looks almost the same as the RPC/encoded WSDL (see Listing 4). The use in the binding is changed from *encoded* to *literal*. That's it.

Listing 4. RPC/literal WSDL for myMethod

```
<message name="myMethodRequest">
    <part name="x" type="xsd:int"/>
    <part name="y" type="xsd:float"/%gt;
</message>
<message name="empty"/>

<portType name="PT">
    <operation name="myMethod">
        <input message="myMethodRequest"/>
        <output message="empty"/>
    </operation>
</portType>

<binding .../>
<!-- I won't bother with the details, just assume it's RPC/literal. -->
```

What about the SOAP message for RPC/literal (see Listing 5)? Here there is a bit more of a change. The type encodings have been removed.

Listing 5. RPC/literal SOAP message for myMethod

```
<soap:envelope>
    <soap:body>
        <myMethod>
            <x>5</x>
```

```
            <y>5.0</y>
        </myMethod>
    </soap:body>
</soap:envelope>
```

Here are the strengths and weaknesses of this approach:

Strengths

- The WSDL is still about as straightforward as it is possible for WSDL to be.
- The operation name still appears in the message.
- The type encoding info is eliminated.
- RPC/literal is WS-I compliant.

Weaknesses

- You still cannot easily validate this message since only the `<x ...>5</x>` and `<y ...>5.0</y>` lines contain things defined in a schema; the rest of the `soap:body` contents comes from WSDL definitions.

A note about xsi:type and literal use

Although in normal circumstances `xsi:type` does not appear in a literal WSDL's SOAP message, there are still cases when type information is necessary and it will appear -- in polymorphism, for instance. If the API expects a base type and an extension instance is sent, the type of that instance must be provided for proper deserialization of the object.

What about the document styles? Do they help overcome this weakness?

Document/encoded

Nobody follows this style. It is not WS-I compliant. So let's move on.

Document/literal

The WSDL for document/literal changes somewhat from the WSDL for RPC/literal. The differences are highlighted in bold in Listing 6.

Listing 6. Document/literal WSDL for myMethod

```
                                    <types>
    <schema>
        <element name="xElement" type="xsd:int"/>
        <element name="yElement" type="xsd:float"/>
    </schema>
</types>

<message name="myMethodRequest">
    <part name="x" element="xElement"/>
    <part name="y" element="yElement"/>
</message>
<message name="empty"/>

<portType name="PT">
    <operation name="myMethod">
        <input message="myMethodRequest"/>
        <output message="empty"/>
    </operation>
</portType>

<binding .../>
<!-- I won't bother with the details, just assume it's document/literal. -->
```

The SOAP message for this WSDL is in Listing 7:

Listing 7. Document/literal SOAP message for myMethod

```
<soap:envelope>
    <soap:body>
        <xElement>5</xElement>
        <yElement>5.0</yElement>
    </soap:body>
</soap:envelope>
```

Here are the strengths and weaknesses of this approach:

A note about the message part

I could have changed only the binding, as I

did from RPC/encoded to RPC/literal. It would have been legal WSDL. However, the WS-I Basic Profile dictates that document/literal message parts refer to elements rather than types, so I'm complying with WS-I. (And using an element part here leads well into the discussion about the document/literal wrapped pattern.)

Strengths

- There is no type encoding info.
- You can finally validate this message with any XML validator. Everything within the `soap:body` is defined in a schema.
- Document/literal is WS-I compliant, but with restrictions (see weaknesses).

Weaknesses

- The WSDL is getting a bit more complicated. This is a very minor weakness, however, since WSDL is not meant to be read by humans.
- The operation name in the SOAP message is lost. Without the name, dispatching can be difficult, and sometimes impossible.
- WS-I only allows one child of the `soap:body` in a SOAP message. As you can see in Listing 7, this example's `soap:body` has two children.

The document/literal style seems to have merely rearranged the strengths and weaknesses from the RPC/literal model. You can validate the message, but you lose the operation name. Is there anything you can do to improve upon this? Yes. It's called the document/literal wrapped pattern.

Document/literal wrapped

Before I describe the rules for the document/literal wrapped pattern, let me show you the WSDL and the SOAP message in Listing 8 and Listing 9.

Listing 8. Document/literal wrapped WSDL for myMethod

```
<types>
    <schema>
        <element name="myMethod">
            <complexType>
                <sequence>
                    <element name="x" type="xsd:int"/>
                    <element name="y" type="xsd:float"/>
                </sequence>
            </complexType>
        </element>
        <element name="myMethodResponse">
            <complexType/>
        </element>
    </schema>
</types>
<message name="myMethodRequest">
    <part name="parameters" element="myMethod"/>
</message>
<message name="empty">
    <part name="parameters" element="myMethodResponse"/>
</message>

<portType name="PT">
    <operation name="myMethod">
        <input message="myMethodRequest"/>
        <output message="empty"/>
    </operation>
</portType>

<binding .../>
<!-- I won't bother with the details, just assume it's document/literal. -->
```

The WSDL schema now has a wrapper around the parameters (see Listing 9).

Listing 9. Document/literal wrapped SOAP message for myMethod

```
<soap:envelope>
    <soap:body>
        <myMethod>
            <x>5</x>
            <y>5.0</y>
        </myMethod>
    </soap:body>
</soap:envelope>
```

Notice that this SOAP message looks remarkably like the RPC/literal SOAP message in Listing 5. You might say it looks exactly like the RPC/literal SOAP message, but there's a subtle difference. In the RPC/literal SOAP message, the `<myMethod>` child of `<soap:body>` was the name of the operation. In the document/literal wrapped SOAP message, the

<myMethod> clause is the name of the wrapper element which the single input message's part refers to. It just so happens that one of the characteristics of the wrapped pattern is that the name of the input element is the same as the name of the operation. This pattern is a sly way of putting the operation name back into the SOAP message.

These are the basic characteristics of the document/literal wrapped pattern:

- The input message has a single part.
- The part is an element.
- The element has the same name as the operation.
- The element's complex type has no attributes.

Here are the strengths and weaknesses of this approach:

Strengths

- There is no type encoding info.
- Everything that appears in the `soap:body` is defined by the schema, so you can easily validate this message.
- Once again, you have the method name in the SOAP message.
- Document/literal is WS-I compliant, *and* the wrapped pattern meets the WS-I restriction that the SOAP message's `soap:body` has only one child.

Weaknesses

- The WSDL is even more complicated.

As you can see, there is still a weakness with the document/literal wrapped pattern, but it's minor and far outweighed by the strengths.

Where is doc/lit wrapped defined?

This wrapped style originates from Microsoft®. There is no specification that defines this style; so while this style is a good thing, unfortunately, the only choice right now, in order to interoperate with Microsoft's and other's implementations, is to make educated guesses as to how it works based on the output of the Microsoft WSDL generator. This pattern has been around for awhile, and the industry has done a good job of understanding it, but while the pattern is fairly obvious in this example, there are corner cases in which the proper thing to do is not particularly clear. Our best hope is that an independent group like the WS-I organization will help stabilize and standardize this in the future.

RPC/literal wrapped?

From a WSDL point of view, there's no reason the wrapped pattern is tied only to document/literal bindings. It could just as easily be applied to an RPC/literal binding. But this would be rather silly. The SOAP message would contain a `myMethod` element for the operation and a child `myMethod` element for the element name. Also, even though it's legal WSDL, an RPC/literal part should be a type; an element part is not WS-I compliant.

Why not use document/literal wrapped all the time?

So far, this article has given the impression that the document/literal wrapped style is the best approach. Very often that's true. But there are still cases where you'd be better off using another style.

Reasons to use document/literal non-wrapped

If you have overloaded operations, you cannot use the document/literal wrapped style.

Imagine that, along with the method you've been using all along, you have the additional method in Listing 10.

Listing 10. Problem methods for document/literal wrapped

```
public void myMethod(int x, float y);
public void myMethod(int x);
```

WSDL allows overloaded operations. But when you add the wrapped pattern to WSDL, you require an element to have the same name as the operation, and you cannot have two elements with the same name in XML. So you must use the document/literal, non-wrapped style or one of the RPC styles.

A note about overloaded operations

WSDL 2.0 will not allow overloaded operations. This is unfortunate for languages like the Java language which *do* allow them. Specs like JAX-RPC will have to define a name mangling scheme to map overloaded methods to WSDL. WSDL 2.0 merely moves the problem from the WSDL-to-SOAP mapping to the WSDL-to-language mapping.

Reasons to use RPC/literal

Since the document/literal, non-wrapped style doesn't provide the operation name, there are cases where you'll need to use one of the RPC styles. For instance, say you have the set of methods in Listing 11.

Listing 11. Problem methods for document/literal non-wrapped

```
public void myMethod(int x, float y);
public void myMethod(int x);
public void someOtherMethod(int x, float y);
```

Now assume that your server receives the document/literal SOAP message that you saw back in Listing 7. Which method should the server dispatch to? All you know for sure is that it's not `myMethod(int x)` because the message has two parameters and this method requires one. It could be either of the other two methods. With the document/literal style, you have no way to know which one.

Instead of the document/literal message, assume that the server receives an RPC/literal message such as the one in Listing 5. With this message it's fairly easy for a server to decide which method to dispatch to. You know the operation name is *myMethod*, and you know you have two parameters, so it must be `myMethod(int x, float y)`.

Reasons to use RPC/encoded

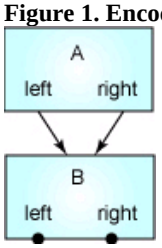
The primary reason to prefer the RPC/encoded style is for data graphs. Imagine that you have a binary tree whose nodes are defined in Listing 12.

Listing 12. Binary tree node schema

```
<complexType name="Node">
    <sequence>
        <element name="name" type="xsd:string"/>
        <element name="left" type="Node" xsd:nillable="true"/>
        <element name="right" type="Node" xsd:nillable="true"/>
    </sequence>
</complexType>
```

With this node definition, you could construct a tree whose root node -- A -- points to node B through both its left and right links (see Figure 1).

Figure 1. Encoded tree.



The standard way to send data graphs is to use the href tag, which is part of the RPC/encoded style (Listing 13).

Listing 13. The RPC/encoded binary tree

```
<A>
    <name>A</name>
    <left href="12345"/>
    <right href="12345"/>
</A>
<B id="12345">
    <name>B</name>
    <left xsi:nil="true"/>
    <right xsi:nil="true"/>
</B>
```

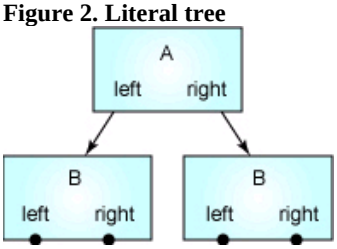
Under any literal style, the href attribute is not available, so the graph linkage is lost (see Listing 14 and Figure 2). You still have a root node, A, which points to a node B to the left and another node B to the right. These B nodes are equal, but they are not the same node. Data have been duplicated instead of being referenced twice.

Listing 14. The literal binary tree

```
<A>
    <name>A</name>
    <left>
        <name>B</name>
        <left xsi:nil="true"/>
        <right xsi:nil="true"/>
    </left>
```

```
    <right>
        <name>B</name>
        <left xsi:nil="true"/>
        <right xsi:nil="true"/>
    </right>
</A>
```

Figure 2. Literal tree



There are various ways you can do graphs in literal styles, but there are no standard ways; so anything you might do would probably not interoperate with the service on the other end of the wire.

SOAP response messages

So far I have been discussing request messages. But what about response messages? What do they look like? By now it should be clear to you what the response message looks like for a document/literal message. The contents of the `soap:body` are fully defined by a schema, so all you have to do is look at the schema to know what the response message looks like. For instance, see Listing 15 for the response for the WSDL in Listing 8.

Listing 15. document/literal wrapped response SOAP message for myMethod

```
<soap:envelope>
    <soap:body>
        <myMethodResponse/>
    </soap:body>
</soap:envelope>
```

But what is the child of the `soap:body` for the RPC style responses? The WSDL 1.1 specification is not clear. But WS-I comes to the rescue. WS-I's Basic Profile dictates that in the RPC/literal response message, the name of the child of `soap:body` is "... the corresponding `wsdl:operation` name suffixed with the string 'Response'." Surprise! That's exactly what the conventional wrapped pattern's response element is called. So Listing 15 applies to the RPC/literal message as well as the document/literal wrapped message. (Since RPC/encoded is not WS-I compliant, the WS-I Basic Profile doesn't mention what an RPC/encoded response looks like, but you can assume the same convention applies here that applies everywhere else.) So the contents of response messages are not so mysterious after all.

Summary

There are four binding styles (there are really five, but document/encoded is meaningless). While each style has its place, under most situations the best style is document/literal wrapped.

Resources

- Be prepared for those times when you have to create your SOAP message by hand and deal with namespace issues without the help of a tool by reading "Handle namespaces in SOAP messages you create by hand" (developerWorks, May 2005).
- Read the Web Services Description Language (WSDL) 1.1, the specification of WSDL.
- For a preview of where WSDL is going, read the WSDL 2.0 draft specifications.
- Try the SOAP 1.2 Primer, an introduction to the SOAP 1.2 specification.
- Java API for XML-Based RPC (JAX-RPC) Downloads & Specifications provides links to the JAX-RPC 1.1 specification itself, as well as javadocs, class files, and Sun's JAX-RPC reference implementation.

- Browse the WS-I (Web Services Interoperability) organization's Web pages.
- Read the WS-I's profiles. Particularly relevant is the basic profile.
- Browse for books on these and other technical topics.
- Get involved in the developerWorks community by participating in developerWorks blogs.
- Want more? The developerWorks SOA and Web services zone hosts hundreds of informative articles and introductory, intermediate, and advanced tutorials on how to develop Web services applications.

About the author

Russell Butek is an SOA and Web services consultant for IBM. He has been one of the developers of IBM WebSphere Web services engine. He has also been a member the JAX-RPC Java Specification Request (JSR) expert group. He was involved in the implementation of Apache's AXIS SOAP engine, driving AXIS 1.0 to comply with JAX-RPC 1.0.