SOAP Encoding

Reference:

Articles at
http://www.ibm.com/developerworks/

SOAP encoding styles

- SOAP uses XML to marshal data
- SOAP defines more than one encoding method to convert data from a software object into XML format and back again.
- The SOAP-encoded data is packaged into the body of a message and sent to a host.
- The host then decodes the XML-formatted data back into a software object.

SOAP encoding styles

- SOAP Remote Procedure Call (RPC) encoding:
  - also known as Section 5 encoding, which is defined by the SOAP 1.1 specification
- SOAP Remote Procedure Call Literal encoding (SOAP RPC-literal):
  - which uses RPC methods to make calls but uses an XML do-it-yourself method for marshalling the data
- SOAP document-style encoding:
  - which is also known as message-style or document-literal encoding.

SOAP RPC encoding style

- SOAP RPC encoding style the simplicity.
- You make a call to a remote object, passing along any necessary parameters.
  - The SOAP stack serializes the parameters into XML, moves the data to the destination using transports such as HTTP and SMTP,
  - receives the response,
  - deserializes the response back into objects, and
  - returns the results to the calling method.
- It handles all the encoding and decoding, even for very complex data types, and binds to the remote object automatically.

SOAP RPC literal encoding

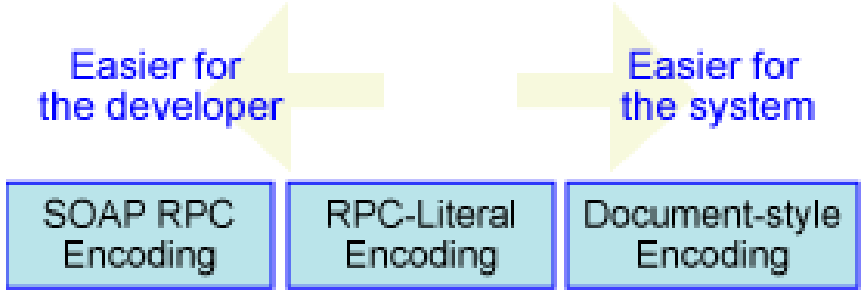
- Assume that you have some data already in XML format.
- SOAP RPC also allows literal encoding of the XML data as a single field that is serialized and sent to the Web service host. This is what's referred to as RPC-literal encoding.
- Since there is only one parameter -- the XML tree -- the SOAP stack only needs to serialize one value.
- The SOAP stack still deals with the transport issues to get the request to the remote object.
- The stack binds the request to the remote object and handles the response.

SOAP document-style call

- SOAP stack sends an entire XML document to a server without even requiring a return value.
- The message can contain any sort of XML data that is appropriate to the remote service.
- The developer handles everything, including
  - determining the transport (e.g., HTTP, MQ, SMTP),
  - marshaling and unmarshaling the body of the SOAP envelope, and
  - parsing the XML in the request and response to find the needed data.

Summary

- SOAP RPC encoding is easiest for the software developer;
  - however, all that ease comes with a scalability and performance penalty.
- In SOAP RPC-literal encoding, you are more involved with handling XML parsing,
  - but it requires there to be overhead for the SOAP stack to deal with.
- SOAP document-literal encoding is most difficult for the software developer,
  - but consequently requires little SOAP overhead.



Why is SOAP RPC easier?

- With this encoding style, you only need to define the public object method in your code once;
- the SOAP stack unmarshals the request parameters into objects and passes them directly into the method call of your object.
- Otherwise, you are stuck with the task of parsing through the XML tree to find the data elements you need before you get to make the call to the public method.

Why is SOAP RPC easier?

- There is an argument for parsing the XML data yourself: since you know the data in the XML tree best, your code will parse that data more efficiently than generalized SOAP stack code.
- You will find this when measuring scalability and performance in SOAP encoding styles.

WSDL SOAP binding

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

Java code

- Java method

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

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 .../>
```

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>
```

RPC/encoded

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.

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.

RPC/literal

- The RPC/literal WSDL for this method looks almost the same as the RPC/encoded WSDL.
- The use in the binding is changed from encoded to literal.

RPC/literal 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>
```

RPC/literal SOAP message for myMethod

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

- The type encodings have been removed.

RPC/literal

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.

Document/encoded

- Nobody follows this style.
- It is not WS-I compliant.

Document/literal

- The WSDL for document/literal changes somewhat from the WSDL for RPC/literal.

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 .../>
```

Document/literal SOAP message for myMethod

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

Document/literal

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

Document/literal

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 the example soap:body has two children.

Document/literal wrapped

```
<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>
```

Document/literal wrapped

```
<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 .../>
```

Document/literal wrapped

- The WSDL schema now has a wrapper around the parameters
- Document/literal wrapped SOAP message for myMethod

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

Document/literal wrapped

- 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 tricky way of putting the operation name back into the SOAP message.

Document/literal wrapped

- 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.

Document/literal wrapped

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.