

Practical Lessons Learned in Web Services Design and Implementation



André Tost

Senior Technical Staff Member

IBM Software Group

atost@us.ibm.com





Agenda

- Strongly typed *versus* loosely typed
- Namespaces
- Common XML Schema concerns
 - Wrapped document style
 - Unsupported elements
 - Type mapping
 - Elements *vs.* Attributes
 - Null
- Explicit and implicit SOAP headers
- Stateful Web services / Contextual information
- Asynchronous Web services
- Service Deployment
 - Location
 - Lookup
- Versioning
- (.Net) interoperability

Strongly Typed *versus* Loosely Typed

- As more Web services are used for application integration, they are designed with messaging in mind
 - Asynchronous
 - We'll get to that later
 - "loosely typed"
 - "document style"
- Let's look at what that really means
 - WSDL definitions
 - Resulting SOAP messages



Strongly Typed *versus* Loosely Typed

- We have seen many Web services definitions that use generic String parameters
 - One for input, one for output
 - Often carrying XML information
- Service requester builds String request message and parses the returned response (if any)
- Service provider parses incoming String message and builds returned response String message
- Both must agree on format of what is encoded in that String
 - Information cannot be included in WSDL and/or schema



Strongly Typed *versus* Loosely Typed

A “loosely typed” service definition using Strings:

```
<wsdl:definitions targetNamespace="http://mycomp.com" ...>
  <wsdl:types>
    <schema targetNamespace="http://mycomp.com" ...>
      <element name="executeResponse">
        <complexType>
          <sequence>
            <element name="executeReturn" nillable="true" type="xsd:string"/>
          </sequence>
        </complexType>
      </element>
      <element name="execute">
        <complexType>
          <sequence>
            <element name="message" nillable="true" type="xsd:string"/>
          </sequence>
        </complexType>
      </element>
    </schema>
  </wsdl:types>
</wsdl:definitions>
```



Strongly Typed *versus* Loosely Typed

- **Example:** send the following XML document:

```
<customer>
  <name>Joe Smith</name>
  <address>
    <street>4308 Sunshine Blvd</street>
    <city>Rochester</city>
    <state>MN</state>
    <zip>55901</zip>
  </address>
</customer>
```

- **Resulting SOAP message:**

```
<soapenv:Envelope ...>
  <soapenv:Header/>
  <soapenv:Body>
    <p416:executeResponse xmlns:p416="http://mycomp.com">
      <executeReturn>&lt;customer&gt;          &lt;name&gt;Joe Smith&lt;/name&gt;  &lt;address&gt;
        &lt;street&gt;4308 Sunshine Blvd&lt;/street&gt;          &lt;city&gt;Rochester&lt;/city&gt;
        &lt;state&gt;MN&lt;/state&gt;          &lt;zip&gt;55901&lt;/zip&gt;
        &lt;/address&gt;&lt;/customer&gt;</executeReturn>
      </p416:executeResponse>
    </soapenv:Body>
  </soapenv:Envelope>
```





Strongly Typed *versus* Loosely Typed

- A String-encoded XML document is hard to read/debug, and it must always be parsed by the requester or provider before being usable
- Usually, it is best to avoid this design style, unless the data is already string based
 - Many enterprise applications deal with string typed messages only
 - Could be character-delimited, or fixed length
- So how do I deal with generic XML?

Strongly Typed *versus* Loosely Typed

- One alternative is to use `<xsd:any/>`
 - One element representing arbitrary XML
 - Well-formed
 - No schema description in WSDL
 - JAX-RPC maps it to `javax.xml.soap.SOAPElement`
 - Part of SAAJ API
- Requester and provider can insert XML directly into the SOAP body
 - Can be tedious if using the SAAJ API
- Requester and provider must still agree on the message format, or be coded to deal with unknown formats
 - Schema may exist, but is not included in WSDL



Strongly Typed *versus* Loosely Typed

A “loosely typed” service definition using `<xsd:any/>`:

```
<wsdl:definitions targetNamespace="http://mycomp.com" ...>
  <wsdl:types>
    <schema targetNamespace="http://mycomp.com" ...>
      <element name="executeResponse">
        <complexType>
          <sequence>
            <xsd:any/>
          </sequence>
        </complexType>
      </element>
      <element name="execute">
        <complexType>
          <sequence>
            <xsd:any/>
          </sequence>
        </complexType>
      </element>
    </schema>
  </wsdl:types>
  ...
</wsdl:definitions>
```



Strongly Typed *versus* Loosely Typed

```
<soapenv:Envelope ...>
  <soapenv:Header/>
  <soapenv:Body>
    <p416:executeResponse xmlns:p416="http://mycomp.com">
      <customer>
        <name>Joe Smith</name>
        <address>
          <street>4308 Sunshine Blvd</street>
          <city>Rochester</city>
          <state>MN</state>
          <zip>55901</zip>
        </address>
      </customer>
    </p416:executeResponse>
  </soapenv:Body>
</soapenv:Envelope>
```





Strongly Typed *versus* Loosely Typed

```

package com.mycomp;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPFactory;
import com.ibm.websphere.webservices.soap.IBMSOAPFactory;
public class GenericAnySoapBindingImpl implements com.mycomp.GenericAny{

    public String customer =
        "<customer>"
        "    <name>Joe Smith</name>"
        "    <address>"
        "        <street>4308 Sunshine Blvd</street>"
        "        <city>Rochester</city>"
        "        <state>MN</state>"
        "        <zip>55901</zip>"
        "    </address>"
        "</customer>";

    public javax.xml.soap.SOAPElement execute(javax.xml.soap.SOAPElement any) throws java.rmi.RemoteException {
        SOAPElement result = null;
        try {
            SOAPFactory factory = SOAPFactory.newInstance();
            result = ((IBMSOAPFactory)factory).createElementFromXMLString(customer);
        } catch (SOAPException x) {}
        return result;
    }
}

```



Strongly Typed *versus* Loosely Typed

- By the way, this has nothing to do with “document style” Web services
 - Most use ‘wrapped document literal’ services
 - Those introduce a wrapper element, which makes the SOAP message look like rpc
- I can define “rpc” style Web services loosely typed or strongly typed
 - *e.g.* start with Java method that has one String input parameter, and returns a String
 - Okay, it’s not that easy for `<xsd:any/>`
- Try to separate discussion about typing from discussion about invocation style



Strongly Typed *versus* Loosely Typed

```

<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://mycomp.com" xmlns:impl="http://mycomp.com" xmlns:intf="http://mycomp.com" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:wsdl:wsi="http://ws-i.org/profiles/basic/1.1/xsd" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
    <schema targetNamespace="http://mycomp.com" xmlns="http://www.w3.org/2001/XMLSchema" xmlns:impl="http://mycomp.com" xmlns:intf="http://mycomp.com"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <element name="getCustomerResponse">
        <complexType>
          <sequence>
            <element name="getCustomerReturn" nillable="true" type="impl:Customer"/>
          </sequence>
        </complexType>
      </element>
      <element name="getCustomer">
        <complexType>
          <sequence>
            <element name="criteria" nillable="true" type="xsd:string"/>
          </sequence>
        </complexType>
      </element>
      <complexType name="Customer">
        <sequence>
          <element name="address" nillable="true" type="impl:Address"/>
          <element name="name" nillable="true" type="xsd:string"/>
        </sequence>
      </complexType>
      <complexType name="Address">
        <sequence>
          <element name="city" nillable="true" type="xsd:string"/>
          <element name="state" nillable="true" type="xsd:string"/>
          <element name="street" nillable="true" type="xsd:string"/>
          <element name="zip" nillable="true" type="xsd:string"/>
        </sequence>
      </complexType>
    </schema>
  </wsdl:types>
</wsdl:definitions>

```



Strongly Typed *versus* Loosely Typed

- And here is how simple the Java looks for this WSDL:

```
package com.mycomp;
public class TypedCustomerRetrieval {
    public Customer getCustomer(String criteria) {
        Customer customer = new Customer();
        Address address = new Address();
        address.setCity("Rochester");
        address.setState("MN");
        address.setStreet("4308 Sunshine Blvd");
        address.setZip("55901");

        customer.setName("Joe Smith");
        customer.setAddress(address);
        return customer;
    }
}
```





Strongly Typed *versus* Loosely Typed

- A mix of both styles is possible
 - Define strongly typed service in WSDL
 - Switch off JAX-RPC type mapping
 - Handle raw XML in implementation *via* SAAJ
- This can be done by the requester or provider or both
 - No dependency between interface and implementation



Strongly Typed *versus* Loosely Typed

- Best practice: always try to define strongly typed WSDL
 - Well defined contract
 - Easier to handle by tools
 - Less error prone
- In some cases, loosely typed services are better:
 - Frequently changing message formats
 - Need to make sure requester and provider implementation is flexible enough to handle changing messages
 - Service is routing XML data without processing it
 - “Message router” service, or intermediary
 - Schema is complex and is not handled by tooling
 - Many industry standard schemas contain elements that are not supported by JAX-RPC or JAXB



Namespaces

- Namespaces are your friend!!! 😊
- Make sure you add a namespace to all WSDL and all XML schemas
 - May want to use different namespace for a service and the schema
 - WS-I Basic Profile has quite a few namespace rules, for example:
 - Requires that schema has a targetNamespace
 - Requires that all children of body be namespace qualified
 - Requires that all children of fault are unqualified
 - For document style, disallows namespace definitions for the body in the SOAP binding
 - For rpc style, requires namespace definition for the body in the SOAP binding
- Use `elementFormDefault="qualified"`
 - Avoids unqualified elements
- 'Wrapped Doc/lit' style allows better control over namespaces
 - No implicitly generated wrapper elements, all are defined in schema
- Can use namespaces for versioning
 - There is really no better approach for this

Namespaces

- Example schema:

```
<schema targetNamespace="http://mycomp.com" ... elementFormDefault="unqualified">
  <complexType name="Customer">
```

- SOAP message:

```
<soapenv:Envelope ...>
  <soapenv:Header/>
  <soapenv:Body>
    <p416:storeCustomer xmlns:p416="http://mycomp.com">
      <customer>
        <address>
          <city>Rochester</city>
          <state>MN</state>
          <street>2981 Oakview Drive</street>
          <zip>55906</zip>
        </address>
        <name>Joe Smith</name>
      </customer>
    </p416:storeCustomer>
  </soapenv:Body>
</soapenv:Envelope>
```

This element is in a namespace

These elements are not!

Namespaces

- Example schema:

```
<schema targetNamespace="http://mycomp.com" ... elementFormDefault="qualified">
  <complexType name="Customer">
```

- SOAP message:

```
<soapenv:Envelope ...>
  <soapenv:Header/>
  <soapenv:Body>
    <p416:storeCustomer xmlns:p416="http://mycomp.com">
      <p416:customer>
        <p416:address>
          <p416:city>Rochester</city>
          <p416:state>MN</state>
          <p416:street>2981 Oakview Drive</street>
          <p416:zip>55906</zip>
        </p416:address>
        <p416:name>Joe Smith</name>
      </p416:customer>
    </p416:storeCustomer>
  </soapenv:Body>
</soapenv:Envelope>
```

All elements are part of the namespace.





Namespaces

- Note that instead of...

```
<p416:storeCustomer xmlns:p416="http://mycomp.com">  
  <customer>  
    <address>
```

...

```
</p416:storeCustomer>
```

- you may also see

```
<storeCustomer xmlns="http://mycomp.com">  
  <customer xmlns="">  
    <address xmlns="">
```

...

```
</storeCustomer>
```

- Both are equivalent.



Common XML Schema Concerns

- **Unsupported types**

- Many XML Schema artifacts are not supported by JAX-RPC, or are optional
 - For example, `<xsd:choice/>`, `<xsd:union/>` `<xsd:anyType/>`, abstract types, ...
- Vendors either have proprietary support, or don't support those at all in the tools
 - May generate `SOAPElement` to map unsupported types
- Try to avoid those if you can
 - JAX-WS 2.0 / JAXB 2.0 will support full set of schema elements
 - Next page shows an example for how to avoid `<xsd:choice/>`



Common XML Schema Concerns

- Schema with choice

```
<xsd:element name="Name">
  <xsd:complexType>
    <xsd:choice>
      <xsd:element name="CompanyName" type="tns:CompanyNameType"/>
      <xsd:element name="PersonName" type="tns:PersonNameType"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
<xsd:complexType name="CompanyNameType">
  <xsd:sequence>
    <xsd:element name="FullName" type="xsd:string"/>
    <xsd:element name="LegalForm" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="PersonNameType">
  <xsd:sequence>
    <xsd:element name="FirstName" type="xsd:string"/>
    <xsd:element name="LastName" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```



Common XML Schema Concerns

- Example without choice

```
<xsd:complexType name="Name"/>
<xsd:element name="Name" type="tns:Name"/>
<xsd:complexType name="CompanyNameType">
  <xsd:extension base="Name">
    <xsd:sequence>
      <xsd:element name="FullName" type="xsd:string"/>
      <xsd:element name="LegalForm" type="xsd:string"/>
    </xsd:sequence>
  </xsd:extension>
</xsd:complexType>
<xsd:complexType name="PersonNameType">
  <xsd:extension base="Name">
    <xsd:sequence>
      <xsd:element name="FirstName" type="xsd:string"/>
      <xsd:element name="LastName" type="xsd:string"/>
    </xsd:sequence>
  </xsd:extension>
</xsd:complexType>
```



Common XML Schema Concerns

- Wrapped document style
 - Most common style of Web services
 - Not formally specified
 - Requires single wrapper element for request message of document/literal service
 - Must not use “type” attribute in message part definition
 - WS-I Basic Profile requires this, too
 - Wrapper element named after operation
 - WS-I Basic Profile does not require this
 - But it has become the norm among vendors
 - If reusing externally defined schema types, you may have to declare the wrapper element locally in the `<types>` section and import the schema



Common XML Schema Concerns

- Type mapping

- JAX-RPC assumes that you want to map all types in a schema to Java classes
 - JAX-RPC 1.0 defined its own mapping
 - JAX-RPC 1.1 still had those, but also supported JAXB 1.0
 - JAX-RPC 2.0 (renamed to JAX-WS 2.0) uses JAXB 2.0
 - Vendors may offer ways to roll-your-own mapping
- As mentioned earlier, you may not be able to completely map any schema
 - At least not until JAX-WS 2.0 arrives
 - Forces you to do your own parsing
- Moreover, you may not want to map a schema into a collection of Java classes
 - May result in a very large number of classes
 - You may want to deal with plain XML in your requester or provider



Common XML Schema Concerns

- Elements *versus* attributes
 - This has been a discussion since XML Schema first came out
 - Somewhat of a philosophical debate over where and when to use which one
 - Attributes can only be simple types
 - Elements are more common
 - Attributes use up slightly less space
 - JAX-RPC defines that both elements and attributes are turned into class properties when generating Java from WSDL
 - It does not define how to tell a Java2WSDL tool which properties to turn into elements and which into attributes
 - Default behavior is all elements
 - Maybe this another reason why attributes are rarely used in Web services schemas
 - The best practice? There really isn't any. You are somewhat limited by what the tools support, otherwise the usual discussion about attributes versus elements applies



Common XML Schema Concerns

- Null

- To represent null in an XML document you have several choices in the schema
 - use="optional" for attributes
 - nillable="true" for elements
 - minOccurs="0" for elements
- They lead to different XML instances
 - use="optional" -> attribute isn't there
 - nillable="true" -> element contains xsi:nil="true"
 - minOccurs="0" -> element isn't there
- Typically, minOccurs or optional are better, because they make the instance smaller
- However, in arrays, you need to use nillable
 - Otherwise, a member of an array with value null cannot be represented
- .Net cannot handle nillable primitive types



Explicit and Implicit SOAP Headers

- SOAP header fields are often used to transfer contextual information that is not part of the payload of a message
 - Security credentials
 - Transactional information
 - Message handshake data
 - “cookies”
- There are two ways to describe SOAP header fields in WSDL
 - Implicit and explicit headers
 - (Actually, there are three, because one option is not to describe them at all and send them anyway)



Explicit and Implicit SOAP Headers

- “Explicit” means that the message part that is transported in the header is also used in the portType of the service
- “Implicit” means that the header refers to a standalone message part that is not part of the portType
- Implicit headers are not mapped to the JAX-RPC Service Endpoint Interface!!
 - Your JAX-RPC tools may ignore them
 - One solution is to write a JAX-RPC handler for insertion and processing of implicit headers



Explicit and Implicit SOAP Headers

- An example – both styles share the same type definition:

```
<wsdl:definitions targetNamespace="http://soapheader.ibm.com" xmlns:impl="http://soapheader.ibm.com"
  xmlns:intf="http://soapheader.ibm.com" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
    <schema elementFormDefault="qualified" targetNamespace="http://soapheader.ibm.com"
      xmlns="http://www.w3.org/2001/XMLSchema" xmlns:impl="http://soapheader.ibm.com"
      xmlns:intf="http://soapheader.ibm.com" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <element name="getLastSellPrice">
        <complexType>
          <sequence>
            <element name="ticker" nillable="true" type="xsd:string"/>
          </sequence>
        </complexType>
      </element>
      <element name="getLastSellPriceResponse">
        <complexType>
          <sequence>
            <element name="getLastSellPriceReturn" type="xsd:float"/>
          </sequence>
        </complexType>
      </element>
      <element name="quote_timestamp" type="xsd:dateTime" />
    </schema>
  </wsdl:types>
```

Explicit and Implicit SOAP Headers

- Explicit header:

```

<wsdl:message name="getLastSellPriceRequest">
  <wsdl:part element="intf:getLastSellPrice" name="parameters"/>
  <wsdl:part name="request_header" element="intf:quote_timestamp"/>
</wsdl:message>
<wsdl:message name="getLastSellPriceResponse">
  <wsdl:part element="intf:getLastSellPriceResponse" name="parameters"/>
</wsdl:message>
<wsdl:portType name="StockServiceExplicit">
  <wsdl:operation name="getLastSellPrice">
    <wsdl:input message="intf:getLastSellPriceRequest" name="getLastSellPriceRequest"/>
    <wsdl:output message="intf:getLastSellPriceResponse" name="getLastSellPriceResponse"/>
  </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="StockServiceSoapExplicitBinding" type="intf:StockServiceExplicit">
  <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="getLastSellPrice">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="getLastSellPriceRequest">
      <wsdlsoap:header message="intf:getLastSellPriceRequest" part="request_header" use="literal"/>
      <wsdlsoap:body parts="parameters" use="literal"/>
    </wsdl:input>
    <wsdl:output name="getLastSellPriceResponse">
      <wsdlsoap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>

```

The message part "request_header" is defined in the message that is used in the portType

The SOAP binding defines that this part goes into the SOAP header

Explicit and Implicit SOAP Headers

- Implicit header:

```

<wsdl:message name="getLastSellPriceRequest">
  <wsdl:part element="intf:getLastSellPrice" name="parameters"/>
</wsdl:message>
<wsdl:message name="getLastSellPriceResponse">
  <wsdl:part element="intf:getLastSellPriceResponse" name="parameters"/>
</wsdl:message>
<wsdl:message name="getLastSellPriceRequestHeader">
  <wsdl:part name="request_header" element="intf:quote_timestamp"/>
</wsdl:message>
<wsdl:portType name="StockServiceImplicit">
  <wsdl:operation name="getLastSellPrice">
    <wsdl:input message="intf:getLastSellPriceRequest" name="getLastSellPriceRequest"/>
    <wsdl:output message="intf:getLastSellPriceResponse" name="getLastSellPriceResponse"/>
  </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="StockServiceSoapImplicitBinding" type="intf:StockServiceImplicit">
  <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="getLastSellPrice">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="getLastSellPriceRequest">
      <wsdlsoap:header message="intf:getLastSellPriceRequestHeader" part="request_header" use="literal"/>
      <wsdlsoap:body parts="parameters" use="literal"/>
    </wsdl:input>
    <wsdl:output name="getLastSellPriceResponse">
      <wsdlsoap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>

```

The message part "request_header" is defined in its own message, which is not part of the portType

The SOAP binding defines that this part goes into the SOAP header, as before



Explicit and Implicit SOAP Headers

- In both cases, the actual SOAP message is identical!
 - It contains the header field as defined in both WSDL files
- However, if you generate a JAX-RPC service endpoint (SEI) interface for both, they are different!
 - One includes the header field as an additional parameter (*i.e.* in the explicit case), the other one doesn't (*i.e.* in the implicit case)
 - The reason is that JAX-RPC only maps the portType into the SEI
- Hence, explicit header definitions are easier to handle in JAX-RPC implementations



Explicit and Implicit SOAP Headers

- Handling implicit headers is different between the requester and the provider
- Provider:
 - SOAP message can be passed into service implementation *via* the `javax.xml.rpc.server.ServiceLifecycle` interface
 - Implement the interface and get access to the SOAP message *via* `SOAPMessageContext`
 - Requires parsing of message with SAAJ
 - Only works for request message
 - Header fields can be retrieved and processed in a JAX-RPC handler
 - Works for both request and response
- Requester
 - Must use JAX-RPC handlers for both request and response message
 - Not always simple to deploy handlers on client
 - Client can communicate with handler by setting property on stub



Stateful Web Services

- By definition, Web services interactions are stateless
 - Each request is handled independently from another
 - Service provider has no way to determine that two requests came from the same client/user
- Some implementations provide settings that let you attach an interaction to an HTTP session
 - JAX-RPC defines that implementations must be able to participate in a session, but does not provide you with a programming model to take advantage of that
 - Via `javax.xml.rpc.server.ServiceLifecycle`, the implementation has access to the `MessageContext`
 - Axis and others store the `HTTPSession` there, making it available to the service implementation



Stateful Web Services

- WS-Addressing can also help
 - Defines “endpoint reference (EPR)”
 - EPR is extensible to include additional information to identify a specific instance of a service
 - The first request is followed by a response message that has an EPR in the “From” header
 - Subsequent requests are sent to this EPR
- WS-Resource Framework addresses complete set of state requirements for services
 - Uses EPRs to address “WS-Resource”
 - A WS-Resource is a Web service and it has identity and state
 - Public Review Draft of specs at OASIS



Asynchronous Web Services

- My definition of an asynchronous Web service is one where there is either no response to a request, or where the response is received independently (*i.e.* later) from the request
- No response at all means it is a “oneway” Web service
 - No response message defined in WSDL
 - If starting with Java, mapped from a method that returns void
 - If sent over HTTP, no SOAP envelope comes back from the HTTP POST
- JAX-RPC currently does not define a programming model for asynchronous request-response services
 - Will come in JAX-WS 2.0
 - No other standardized way of doing this
- You have two choices
 - Use your own (or a vendor-provided) API
 - Or, establish two oneway Web services

Asynchronous Web Services

■ Asynchronous API

```
...  
Ticket ticket = stockQuoteService.stockQuoteRequest("IBM");  
// ... do something else ...  
float quote = stockQuoteService.stockquoteResponse(ticket);  
...  
  
...  
java.lang.reflect.Method stockQuoteServiceResponse = ...;  
stockQuoteService.stockQuoteRequest("IBM", stockQuoteServiceResponse);  
...
```

■ Two oneway services

- Two separate WSDL
- Must ensure correlation between request and response
- WS-Addressing can help
 - Send EPR of the response service with each request
 - Goes into the "ReplyTo" header





Service Deployment

- Location

- SOAP/HTTP Web services have a URL as their endpoint address
 - Defined in WSDL
- JSR109/921 defines deployment of Web service in J2EE application server
 - Updates endpoint address if necessary
 - Provides JNDI binding to stub
- Write clients to use dynamic endpoint
 - Can set endpoint address on stub



Service Deployment

- Lookup

- Given that the endpoint of a Web service can change over time, we need to have a way to retrieve its location at runtime
 - Move from development to test to production
 - Deployment across multiple servers
 - Updates to existing servers
- There are multiple ways to do that
 - UDDI
 - JNDI
 - Roll-your-own
 - ✓ Files
 - ✓ LDAP



Versioning

- To make a long story short: there isn't any!
- You can make changes backward compatible
 - Add new operations
 - Add new optional parameters
 - Make backward compatible changes to parameters
 - e.g. turn boolean into String
 - Implementation changes should never have an impact
- Encode versioning information in service namespace
 - But essentially this means you are creating a whole new service
 - Existing clients must be updated
- Provide placeholder fields or make service generically typed services
 - xsd:any and xsd:anyType
 - Probably require more handcoding
- You can also dynamically transform and route 'old' requests to 'new' service version
 - Vendor products offer some of this
 - Cannot use JAX-RPC handlers for the transformation, since they cannot change the structure of the message

(.Net) Interoperability

- Rules of interoperability:
 1. Follow the WS-I Profiles
 2. Follow the WS-I Profiles
 3. Follow the WS-I Profiles!!
- Exception to this rule is binary attachments
 - Microsoft has decided not to support the Attachments Profile
 - We are waiting for MTOM implementations to fix this
- Advanced standards have not yet been around long enough
 - WS-Security is getting there
- Some simple things (these are things I have heard and seen, but they may not all apply anymore since implementations are constantly changing)
 - Use wrapped doc/literal style when possible
 - Use exceptions with care
 - .Net has no mapping for SOAP Faults into exceptions
 - Don't make simple types nillable
 - Don't use relative namespaces, and use elementFormDefault="qualified"
 - Avoid CDATA
 - .Net maps xsd:decimal to a String



Summary

- Each subject covered in this presentation probably deserves a longer discussion
 - Plenty of resources on each available on the Net
 - Many times, you will have to decide between sticking with standards or taking advantage of proprietary vendor offerings
- Keep exploring your options!
 - Next round of standardization (including their productization) is happening
 - WS-Addressing
 - JAX-WS 2.0



Web Resources

- IBM developerWorks web services domain
 - <http://www-106.ibm.com/developerworks/webservices/?loc=dwmain>
- OASIS
 - <http://www.oasis-open.org/home/index.php>
- World Wide Web Consortium (W3C)
 - <http://www.w3.org/>
- Java Community Process (JCP)
 - <http://www.jcp.org>
- Apache
 - <http://xml.apache.org>
- WS-I
 - <http://ws-i.org>