Web Services Integration –
It’s All About Standards

Denise Hatzidakis
Perficient, Inc.
denise.hatzidakis@perficient.com
J2EE 1.4 and Web Services

- J2EE 1.4 now contains several Web Services Standards
  - Web Services 1.1
  - JAX-RPC 1.1
  - SAAJ 1.2
  - JAXR 1.0
  - WS-I Basic Profile 1.0

<table>
<thead>
<tr>
<th></th>
<th>Client</th>
<th>Web</th>
<th>EJB</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJB 2.1</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Servlet 2.4</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>JSP 2.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMS 1.1</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>JTA 1.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaMail 1.3</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>JAF 1.0</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>JAXP 1.2</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Connector 1.5</td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Web service-related library support in J2EE 1.4
J2EE 1.4 Web Services

- Web Services are now an integral part of J2EE 1.4
- Support provided via following standards
  - JSR 101 (JAX-RPC)
    - Provides standard programming model for Web Services based on WSDL and SOAP
  - JSR 109 (Enterprise Web Services)
    - Provides standard deployment model for Web Services applications (provider and requestor)
  - WS-I Basic Profile 1.1
    - Promotes Web Services interoperability across J2EE and non-J2EE platforms
  - SAAJ (SOAP with Attachments API for Java) 1.2
    - Allows developers to write applications accessing SOAP messages (JAX-RPC Handlers)
  - JAXR
    - Provides an API for accessing Web Services registries
The Web Service Solution

There are three major parts to a Web Services Solution

- **A Programming model**
  - How you write clients to access Web services
  - How you write service implementations
  - How you handle other parts of the SOAP spec (headers, attachments, *etc.*)

- **A deployment model**
  - Deployment descriptors to map service implementations to SOAP messages
  - Type mapping for parameters (at least in RPC)

- **An Engine**
  - Code to receive SOAP messages and invoke service implementations
  - Code to map Java types to XML
Java Standard Programming and Deployment Model

- Portable Web Services applications (JSR 101) – JAX/RPC
  - Defines the mapping of WSDL to Java and vice versa
  - Defines a client API to invoke a remote Web service
  - Defines a runtime environment for Java Bean as an implementation of a Web service
  - Handler model

- Standard Web Services Deployment Descriptors (JSR 109)
  - JSR109 conceptually enhances JSR101
  - Defines a J2EE compliant deployment/packaging model for Web Services on the server side and for the client
  - New deployment descriptors for Web services
  - Server-side programming model
  - Stateless Session EJB as implementation of a Web service
  - Client-side programming model
  - EJB, Servlet/JSPs, Application Client as client to Web Services
  - J2EE Container required
JAX RPC –

The Programming Model

- Began as V1.0 (JSR101)
- V1.1 added WS-I Basic Profile support and bug fixes
  - Also released under JSR101
- V1.1 specification published October 14<sup>th</sup> 2003
  - Required by J2EE 1.4
- Version 2.0 (JSR224)
  - early draft released 23<sup>rd</sup> July 2004
  - Aims to delegate XML mapping to JAXB 2.0 (JSR222) specification
  - Targets J2SE 1.5
  - The Future...
JAX-RPC Goals

- **Java API for XML based Remote Procedure Call (JAX-RPC)**
  - formalizes the procedure for invoking Web services in an RPC-like manner
- **Defines Client side APIs to access web service**
  - as well as server side requirements
- **Defines mapping model between WSDL, XML and Java**
- **Defines a Handler model to the client and the server side**
  - Allow your custom code to intercept the request and the response
JAX RPC – Features

- Defines
  - mapping of WSDL/XML to Java and vice versa
  - a client API to invoke a remote web service
  - a runtime environment for Java Bean as an implementation of a web service
  - Handler model
- JAX-RPC is designed for portability of code across multiple vendors
  - Any client or service implementation should be portable across any vendor that supports JAX-RPC

Note, JAX-RPC is meant to be protocol-neutral, but requires support for SOAP over HTTP
JAX-RPC

- Mappings for WSDL to XML
  - Standard Java Type <-> XML type mapping
  - Standard Java Type <-> WSDL type mapping
  - Standard SOAP binding mapping

- SOAP Manipulation Library
  - JAX-RPC is built on SAAJ (SOAP Attachments API for Java)
  - SAAJ was formerly part of JAXM
  - Provides pattern for sending and receiving SOAP messages with or without attachments, synchronous or asynchronous

- Servlet Container Service Endpoint
  - Standard way to specify servlets that receive SOAP messages
  - Compatible with JSR 109 style deployment descriptors

- Basic Authentication
  - Handlers
JAX-RPC Working Model

Client Program

Stubs

JAX-RPC Runtime

Soap Message

HTTP

Web Service

Ties

JAX-RPC Runtime
An Architectural View

- JAX-RPC and JSR-109 fit together to form a standard approach for Web service implementation, access and deployment in Java
- Based on a few concepts
  - A **Service Implementation** (either a Java Bean or Stateless Session Bean) implements the methods of a WSDL-Described interface
  - A **Service Endpoint Interface (SEI)** provides a Java "view" of the WSDL-described interface. Clients interact with objects implementing this interface
  - A **Service** mediates access to the **Ports** (acts as a factory for ports)
Analogies to EJBs

- The JAX-RPC approach is analogous to the EJB (or RMI) approach in a number of ways
- A **Service Endpoint Interface** (SEI) is analogous to the Remote interface of an EJB
  - Both the Stub and the Service Implementation implement the methods of the SEI
    - If the service implementation is an EJB, it does not directly implement the SEI interface, just as the EJB does not directly implement the remote interface
  - The Service Implementation is thus analogous to the Bean Implementation of an EJB
- Service Factories (and Services) are analogous to the EJB Homes
  - You obtain stubs from them – they are factories of remote objects
Service Implementation Interface Hierarchy

- javax.rmi.Remote
  - Service Endpoint Interface (SEI)
    - Service Implementation Interface (SIB)
    - ServiceLifeCycle Interface

WSDL generates

ServiceLifeCycle Interface implements

Service Implementation Interface (SIB) implements

javax.rmi.Remote extends
WSDL Document Overview

- Definition
  - The root of the WSDL document
  - Contains the definition of one or more services
  - Usually contains attributes

- Service
  - Defines the service

- Messages and PortTypes
  - Describes the actions available for the service

- Bindings
  - How to communicate with the service
WSDL Review

A WSDL document defines Web Service via:

**Messages**
- Defines a single interaction with the service

**Types**
- Defines data types used in a message

**Operations**
- Description of an action

**Port Types**
- Describes the set of operations supported by the service.

**Bindings**
- A concrete protocol and data format for a particular port type.

**Port**
- Describes the network address where the service is being hosted.

**Service**
- Ties together all the elements of the service.
WSDL and JAX/RPC

Abstract definition of the service

Service

Port
(e.g. http://host/svc)

Binding
(e.g., SOAP)

Port
(e.g. http://host/svc)

Binding
(e.g., SOAP)

PortType

Operation(s)

inMessage

outMessage

part

part

part

part

Service Implementation Bean (SIB)

Client API for invoking services

Service Endpoint Interface (SEI)

Method(s)

Method Parameters

Method Return values

Java Objects

Service Implementation Bean

Client API for invoking services

Service Endpoint Interface (SEI)

Method(s)

Method Parameters

Method Return values

Java Objects
WSDL <-> Java Mapping (JAX-RPC)

- Each WSDL `<portType>` element is mapped to a Java interface.
  - Called the “Service Endpoint Interface” or SEI.
  - This interface must extend `java.rmi.Remote`.
  - Its package name can be derived from the namespace of the `<portType>` element and vice versa.
- Each `<operation>` within the `<portType>` is mapped to a Java method in the service endpoint interface.
  - Every method must throw `java.rmi.RemoteException`.
- The `<message>` elements of the `<operation>` are mapped to parameters and return types of the methods of the service endpoint interface.
- The types in the parts map to Java types
  - Complex types must implement `Serializable`
    - JAX-RPC does not define a pass-by-value semantic thus requiring parameters to be serializable
WSDL Interface <-> Java Interfaces and Classes

```xml
<?xml version="1.0"?>
<definitions name="StockQuoteService interface">
    <message name="SymbolRequest">
        <part name="return" type="xsd:string"/>
    </message>
    <portType name="StockQuotePortType">
        <operation name="getQuote">
            <input message="tns:SymbolRequest"/>
            <output message="tns:QuoteResponse"/>
        </operation>
    </portType>
    <service name="StockQuoteService">
        <port name="StockQuotePort" binding="tns:StockQuoteServiceBinding">
            <soap:address location="http://hostname/StockQuote/services/StockQuotePort"/>
        </port>
    </service>
</definitions>
```

```java
public interface StockQuotePortType extends java.rmi.Remote {
    public float getQuote(java.lang.String in0) throws java.rmi.RemoteException;
}
```

```java
public interface StockQuoteService extends javax.xml.rpc.Service {
    public StockQuote getStockQuotePort() throws javax.xml.rpc.ServiceException;
}
```

Service Endpoint Interface

Service Interface

WSDL
Service Interface

- Service Interface
  - A java representation of the Web Service
  - Must implement the javax.xml.rpc.Service interface
  - Can be used to generate
    - A Stub class
    - A Dynamic Proxy
    - A javax.xml.rpc.Call object
Service Interface and SEI

```java
package sample;

public interface StockQuoteService extends javax.xml.rpc.Service {
    public java.lang.String getPurchaseAddress();
    public sample.PurchasePortType getPurchase() throws javax.xml.rpc.ServiceException;
    public sample.PurchasePortType getPurchase(java.net.URL portAddress) throws javax.xml.rpc.ServiceException;
    public java.lang.String getQuoteAddress();
    public sample.GetQuotePortType getQuote() throws javax.xml.rpc.ServiceException;
    public sample.GetQuotePortType getQuote(java.net.URL portAddress) throws javax.xml.rpc.ServiceException;
}
```

```java
package sample;

public interface GetQuotePortType extends javax.rmi.Remote {
    public sample.Price getQuote(sample.Ticker parameter) throws java.rmi.RemoteException;
}
```

Service Interface

Service Endpoint Interface
JAX-RPC

Server Programming Model

- **Port components**
  - Service Endpoint Interface (SEI)
  - Service Implementation Bean (SIB)

- **Service Implementation Bean**
  - Implements the SEI
  - Must be stateless
  - May implement the ServiceLifecycle interface for managing the service's life cycle

- **EJB tier (JSR109 defines)**
  - SIB must also be a Stateless Session Bean
  - Implementing of the SEI *is not required*
    - Discouraged because SEI methods throw RemoteException

- **Web Tier**
  - Java Bean implementation

- **Provider**
Implementing a JAX/RPC service

- A Quick example will probably best show how the pieces fit.
- Imagine that we want an AddressBook Web Service
  - Can get Addresses for a person’s name
- Begin by creating an implementation (either a Stateless Session Bean or a Java Bean) and then creating an SEI for the exposed methods
  - In our example we will call the SEI AddressBook and the Implementation AddressBookImpl
  - The bean implementation must implement (through the implements keyword) the SEI but EJBs don’t have to to actually implement the SEI, but it can (but the method signatures must match – much like with EJB’s)
public interface AddressBook extends java.rmi.Remote {
    public Address getAddressForName(String name);
}

public class AddressBookImpl implements AddressBook {
    public Address getAddressForName(String name)
        throws RemoteException {
        // fill in implementation here
    }
}

public class Address implements java.io.Serializable {
    int name;
    String streetAddress;
    ...
    // getters and setters not shown
}

Provider

Service Endpoint Interface (SEI)

Service Implementation Bean (SIB)

Java classes which are mapped to XML must implement Serializable
There are two options for the style of a message and (at least) two for the type encoding used in that message.

- RPC Style and SOAP Encoding (rpc/encoded)
- RPC Style and Literal Encoding (rpc/literal)
- Document Style and SOAP Encoding (doc/encoded)
- Document Style and Literal Encoding (doc/literal)

<table>
<thead>
<tr>
<th>Style (SOAP message body format)</th>
<th>Mode (parameter encoding)</th>
<th>JAX-RPC related information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document (uses a schema to define body format)</td>
<td>Literal (uses XML Schema for each parameter encoding)</td>
<td>Required</td>
</tr>
<tr>
<td>Document</td>
<td>Encoded (using SOAP Section 5 encoding)</td>
<td>Optional</td>
</tr>
<tr>
<td>RPC (using SOAP Section 7 rules to define body format)</td>
<td>Encoded</td>
<td>Required</td>
</tr>
<tr>
<td>RPC</td>
<td>Literal</td>
<td>Optional</td>
</tr>
</tbody>
</table>
## Type Mapping

### Primitive Types

<table>
<thead>
<tr>
<th>XML Schema Type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:string</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>xsd:integer</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>xsd:int</td>
<td>int</td>
</tr>
<tr>
<td>xsd:long</td>
<td>long</td>
</tr>
<tr>
<td>xsd:short</td>
<td>short</td>
</tr>
<tr>
<td>xsd:decimal</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>xsd:float</td>
<td>float</td>
</tr>
<tr>
<td>xsd:double</td>
<td>double</td>
</tr>
<tr>
<td>xsd:byte</td>
<td>byte</td>
</tr>
<tr>
<td>xsd:QName</td>
<td>javax.xml.namespace.QName</td>
</tr>
<tr>
<td>xsd:dateTime</td>
<td>java.util.Calendar</td>
</tr>
<tr>
<td>xsd:base64Binary</td>
<td>byte[]</td>
</tr>
<tr>
<td>xsd:hexBinary</td>
<td>byte[]</td>
</tr>
</tbody>
</table>

- JAX-RPC does not dictate a specific Java mapping for `xsd:anyType` until JAX-RPC 1.1.
- XML Schema simple types which map to Java base types map to the Java wrapper classes instead when an element is marked as nillable in the XML Schema.

Example:

```xml
<xsd:element name="code" type="xsd:int" nillable="true" />
```

Maps to: `java.lang.Integer`
## JAX-RPC 1.1

<table>
<thead>
<tr>
<th>xsd:unsignedInt</th>
<th>long</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:unsignedShort</td>
<td>int</td>
</tr>
<tr>
<td>xsd:unsignedByte</td>
<td>short</td>
</tr>
<tr>
<td>xsd:date</td>
<td>java.util.Calendar</td>
</tr>
<tr>
<td>xsd:time</td>
<td>java.util.Calendar</td>
</tr>
<tr>
<td>xsd:anyURI</td>
<td>java.net.URI (J2SE 1.4 only) java.lang.String</td>
</tr>
<tr>
<td>xsd:anySimpleType</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>xsd:duration</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>xsd:gYearMonth</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>xsd:gYear</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>xsd:gMonthDay</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>xsd:gDay</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>xsd:gMonth</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>xsd:normalizedString</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>xsd:token</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>xsd:language</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>xsd:Name</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>xsd:NCName</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>xsd:ID</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>xsd:NMToken</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>xsd:NMTokenS</td>
<td>java.util.List</td>
</tr>
<tr>
<td>xsd:nonPositiveInteger</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>xsd:negativeInteger</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>xsd:nonNegativeInteger</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>xsd:positiveInteger</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>xsd:unsignedLong</td>
<td>java.math.BigInteger</td>
</tr>
</tbody>
</table>
# SOAP Encoded Type Mapping

<table>
<thead>
<tr>
<th>SOAP Encoded Simple Type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>soapenc:string</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>soapenc:boolean</td>
<td>java.lang.Boolean</td>
</tr>
<tr>
<td>soapenc:float</td>
<td>java.lang.Float</td>
</tr>
<tr>
<td>soapenc:double</td>
<td>java.lang.Double</td>
</tr>
<tr>
<td>soapenc:decimal</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>soapenc:int</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>soapenc:short</td>
<td>java.lang.Short</td>
</tr>
<tr>
<td>soapenc:byte</td>
<td>java.lang.Byte</td>
</tr>
<tr>
<td>soapenc:base64</td>
<td>byte[]</td>
</tr>
</tbody>
</table>

*Note: the SOAP encoded types are all nillable, so they are mapped to the Java wrapper classes*
Complex Type Mapping

- XML Arrays
  - XML Arrays are mapped to Java arrays
  - Types of XML Arrays that are mapped
  - Arrays declared using wsdl:arrayType or soapenc:Array
    - Not recommended by the WS-I Basic Profile
- XML Enumeration
  - Mapped to Java class following enumeration pattern
Complex Type Mapping

- **XML Struct / Complex Type**
  - `xsd:complexType` with sequence of elements with simple/complex type
  - `xsd:complexType` with `xsd:all` of elements of simple/complex type
  - Mapped to Java classes (called JAX-RPC Value types)
    - with getters and setters to access each element in the complex type
    - must have public default constructor
    - must implement `java.io.Serializable`
    - not required to be a JavaBean
    - may not implement `java.rmi.Remote`
  - Notes:
    - no sequencing maintained in Java class
    - if `maxOccurs` on element > 1, maps to an Array for the element
    - XML element attributes not specified, maps to `SOAPElement`
Complex Type Mapping

XMLSchema Fragment

```
<xsd:complexType name="Book">
  <sequence>
    <element name="author" type="xsd:string" maxOccurs="10" />
    <element name="price" type="xsd:float" />
  </sequence>
</xsd:complexType>
```

Java

```
public class Book {
    private float price;
    private String[] author;

    String[] getAuthor() {
        return author;
    }

    public void setAuthor(String[] author) {
        // set author
    }

    public float getPrice() {
        return price;
    }

    public void setPrice(float price) {
        // set price
    }
}
```
Enumeration Mapping

<complexType
    name="EyeColorType">
    <restriction base="xsd:string">
        <enumeration value="green"/>
        <enumeration value="blue"/>
    </restriction>
</complexType>

public class EyeColorType {
    // Constructor
    protected EyeColorType(String value) { ... }

    public static final String _green = "green";
    public static final String _blue = "blue";
    public static final EyeColorType green = new EyeColorType(_green);
    public static final EyeColorType blue = new EyeColorType(_blue);

    public String getValue() { ... }
    public static EyeColorType fromValue(String value) {
        ... }

    public boolean equals(Object obj) { ... }
    public int hashCode() {
        ... }
    // ... Other methods not shown
}

Java
JAX-RPC and Literal Encoding

- If JAX-RPC specifies a mapping for the XML type of a message part
  - That mapping is used
- If there is no JAX-RPC mapping for the XML type of the message part
  - An implementer of javax.xml.soap.SOAPElement is used
    - SOAPElement interface comes from SAAJ and provides a DOM-like API for manipulating SOAP messages
- This is true for document or rpc style
JAX/RPC parameter modes

- **IN type:**
  - An *IN* parameter is passed as a copy. The value of the *IN* parameter is copied before a Web service invocation. The return value is created as a copy and returned to the Web service client.

- **OUT type:**
  - An *OUT* parameter is passed as a copy without any input value to the Web service method. The Web service method fills out the *OUT* parameter and then returns it back to the client.

- **IN OUT type:**
  - An *INOUT* parameter is passed as a copy with an input value to the Web service method. The Web service method uses the input value, process it, fills in the *INOUT* parameter with a new value and returns it back to the client.
Holder Classes

- WSDL allows for “in/out” parameters to operations
  - parts that appear both in the input and output message
  - service client uses the Holder class instance to send the values of either the out or the in/out parameter.
  - The contents of the Holder class are modified by the remote method calls and the service client can use this changed content after the method invocation.

- in/out parameters are mapped to Holder Classes

- Holder classes implement `javax.xml.rpc.Holders.Holder`
Holder Classes

- Holder Classes for primitive types
  - Named according to the following convention:
  - float’s holder is `javax.xml.rpc.holders.FloatHolder`
  - float’s holder wrapper is `javax.xml.rpc.holders.FloatWrapperHolder`

- Holder classes are generated for all other XML types
  - For complex XML data types, the name of the Holder class is constructed by
    - appending Holder to the name of the corresponding Java class
    - example: `com.example.holders.BookHolder`
  - Holder has a field `value` with the type of the mapped Java class
  - Public constructor that sets `value` to the constructor argument
Holder Classes Example – Sample WSDL

```xml
<xsd:complexType name="Authors">
  <xsd:all>
    <xsd:element name="Authors" type="typens:AuthorArray"/>
  </xsd:all>
</xsd:complexType>

<message name="AuthorPresentRequest">
  <part name="Authors" type="typens:Authors"/>
</message>

<message name="AuthorPresentResponse">
  <part name="return" type="xsd:boolean"/>
  <part name="Authors" type="typens:Authors"/>
</message>

<portType name="AcmeAuthorPresentPortType">
  <operation name="IsAuthorPresent">
    <input message="typens:AuthorPresentRequest"/>
    <output message="typens:AuthorPresentResponse"/>
  </operation>
</portType>
```

You can see that the input and output messages contain an Authors type as a parameter. This is the in/out style of parameter passing.
Holder Classes Example – Mapped Java

**Holder class Definition**

```java
public final class AuthorsHolder implements javax.xml.rpc.holders.Holder {
    public com.acme.www.Authors value;
    public AuthorsHolder() {}
    public AuthorsHolder(com.acme.www.Authors value) {
        this.value = value;
    }
}
```

**Java Service Endpoint Interface**

```java
public interface AcmeAuthorPresentPortType extends java.rmi.Remote {
    public boolean isAuthorPresent(AuthorsHolder authors) throws java.rmi.RemoteException;
}
```
## Parameter Mappings

<table>
<thead>
<tr>
<th>Parameter defined in</th>
<th>Parameter style</th>
<th>Possible JAX-RPC mapping classes</th>
<th>Sample JAX-RPC mapping class</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsdl:input</td>
<td>in</td>
<td>Wrapper classes or JavaBeans</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>wsdl:output</td>
<td>out</td>
<td>Holder class</td>
<td>IntHolder, StringHolder, etc.</td>
</tr>
<tr>
<td>wsdl:input and wsdl:output</td>
<td>inout</td>
<td>Holder class</td>
<td>IntHolder, StringHolder, etc.</td>
</tr>
</tbody>
</table>
JAX-RPC Type Mapping Framework

- TypeMappingRegistry
- TypeMapping
- Serializer
- Deserializer
Type Mapping / Custom Serialization

- JAX-RPC defines interfaces which a vendor may provide implementations for to support custom serializers
- JAX-RPC does not specify the XML processing model to be used
- A vendor would extend the Serializer and Deserializer interfaces to support a specific XML processing mechanism (ex. SAX)
Type Mapping / Custom Serialization

- **TypeMappingRegistry** interface is how you lookup type mapping for a Java class or XML type

- **TypeMapping**
  - Maps between
    - JavaType (Class)
    - XML Type (QName)
    - `javax.xml.rpc.encoding.SerializerFactory`
    - `javax.xml.rpc.encoding.DeserializerFactory`

- **Serializer**
  - Implements `javax.xml.rpc.encoding.Serializer`

- **Deserializer**
  - Implements `javax.xml.rpc.encoding.Deserializer`

- Get the registry for a service from
  - `getTypeMappingRegistry()` on `javax.xml.rpc.Service`
SOAP with Attachments

- MIME Types are mapped to Java types according to WSDL MIME binding
- To create an attachment, pass one of the mapped Java types as a parameter or return value
- You may also pass javax.activation.DataHandler to handle types not included in the JAX-RPC mapping (implementation dependent).

<table>
<thead>
<tr>
<th>MIME Type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>image/gif</td>
<td>java.awt.Image</td>
</tr>
<tr>
<td>image/jpeg</td>
<td>java.awt.Image</td>
</tr>
<tr>
<td>text/plain</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>multipart/*</td>
<td>javax.mail.internet.MimeMultipart</td>
</tr>
<tr>
<td>text/xml</td>
<td>javax.xml.transform.source</td>
</tr>
<tr>
<td>application/xml</td>
<td></td>
</tr>
</tbody>
</table>
JAX-B (JSR31)

- Java API for XML Binding
  - Initially part of JAXM
- Java data binding facility that compiles an XML schema into one or more Java classes
- Interface oriented
  - binding compiler that binds components of a *source schema* to schema-derived Java *content classes*
  - establish conventions for annotating classes with the necessary metadata.
    - standard way to customize the binding of existing schema’s components to Java
- Binding framework – APIs for
  - *unmarshalling* of an XML document into a tree of interrelate instances of both existing and schema-derived classes,
  - *marshalling* of such *content trees* back into XML
  - *validation* of content trees against the constraints expressed in the schema.
JAX-RPC and JAXB

- According to the JAX-RPC 1.1 specification draft
  - You may or may not use JAXB for converting Java to XML
    - JAXB does however have some impact on type mapping
  - In order to make it possible to use other binding frameworks in JAX-RPC
    - It must be possible to selectively turn the standard JAX-RPC mapping off on a per part basis
    - JAX-RPC tools are required to provide a facility for specifying metadata for this purpose
SAAJ

SOAP Attachments API for Java

- Used in Handler infrastructure to manipulate XML-based SOAP messages (SOAPMessage/SOAPEnvelope/SOAPElement/...)
- Used to render literal XML snippets (SOAPElement)

- Can be used in a stand-alone environment to create SOAP-based messages
  - Including adding Attachments in contained MIME envelope structure
SAAJ 1.1 -> 1.2

- The SAAJ 1.1 interfaces are still valid.

  For SAAJ 1.2,

    - SAAJ 1.1 interfaces implement DOM.
      - Thus the SAAJ tree can now be manipulated with DOM apis.
    - Interface javax.xml.soap.SOAPElement now extends org.w3c.dom.Element.
    - Abstract class javax.xml.soap.SOAPPart now implements org.w3c.dom.Document.
    - Interface javax.xml.soap.Text now extends org.w3c.dom.Text.
SAAJ 1.2

- SAAJ 1.2 contains new APIs and binds the SAAJ APIs to Document Object Model (DOM) APIs
  - The new APIs focus on ease of use and Java Doc support.
  - The SOAPPart of a SOAP message is now also a level 2 DOM document.
  - An object model is a mechanism for accessing a document

```
org.w3c.dom.Document

SOAPPart

java.xml.soap.Node

javax.xml.soap.Text

javax.xml.soap.Text

SOAPElement
```
Exception Handling

- JAX-RPC specification addresses Web Service run-time (application and system) exceptions
  - Based on the standard approach to service endpoint interface design and its mapping to `wsdl.fault` elements
  - Service-specific exceptions are declared in the `wsdl.fault` element, and these exception types are derived from the `java.lang.Exception` class

```
<wsdl:portType name="Transfer_SEI">
  <wsdl:operation name="transferFunds" parameterOrder="fromAcctId toAcctId amount">
    <wsdl:input name="transferFundsRequest" message="impl:transferFundsRequest"/>
    <wsdl:output name="transferFundsResponse" message="impl:transferFundsResponse"/>
    <wsdl:fault name="InsufficientFundsException" message="impl:InsufficientFundsException"/>
  </wsdl:operation>
</wsdl:portType>
```
JAX-RPC 1.1

- Published October 14, 2003
- Part of J2EE 1.4
- Mainly changes to comply with WS-I BP 1.0
  - e.g. mandatory support for rpc/literal
- Plus many clarifications and minor fixes for 1.0 spec
  - e.g. <xsd:any/> mapped to SOAPEElement
  - e.g. requirement to map any element to SOAPElement
  - Attributes on complex types must be mapped to properties of JavaBean class
JAX-RPC 1.1

- New `<simpleType>` to Java mappings
  - New set of primitives
  - New mapping for simpleTypes derived by restriction
- Required mapping for `<list>`.
  - `<list>` maps to an array.
- Unknown optional types must map to SAAJ SOAPElement
- ServiceFactory.loadService methods
- ServletEndpointContext.isUserInRole method
- WSDL faults can have elements as well as types
- Generated Service.get<Name_of_wsd1:port>
- Additional name collision rules
- One way operations from Java void methods
JAX-RPC Runtime Services

- Provided via property interface on Stub and Call interfaces

- HTTP Basic Authentication
  - `javax.xml.rpc.security.auth.username (String)`
  - `javax.xml.rpc.security.auth.password (String)`
  - No other HTTP authentication forms are required

- HTTP Session Management
  - `javax.xml.rpc.session.maintain (boolean)`
JAX-RPC Handlers
JAX-RPC Handlers

- Provides a mechanism for intercepting the SOAP message and operating on header information
  - Can examine and potentially modify a request before it is processed by a Web Service component.
  - Can examine and potentially modify the response after the component has processed the request
- Conceptually, similar to Servlet 2.3 Filter
- Can be provided for both the client and server
- Handler MUST NOT CHANGE SOAP message, Operation name, number of parts in the message, or types of the message parts
  - SOAP Fault is sent if Handler does this
- Handlers are configured into ‘ordered chains’
  - Client side Handlers run after the Stub/proxy has marshaled the message, but before container services and the transport binding occurs.
  - Server side Handlers run after container services have run including method level authorization, but before demarshalling and dispatching the SOAP message to the endpoint.
- Handler chains (handlers) are service specific
JAX RPC Handler Architecture

HTTP Request

Filter Chain

Filter
Filter
Filter

Servlet

SOAP Message

Handler Chain

Handler
Handler

Endpoint

SOAP messagecontext

SOAP message

Provider
End to End Flow for SOAP/HTTP

1. SOAP/HTTP

2. Servlet

3. Handler

4. Deserializer

5. Web Service Endpoint

6. Handler

7. Serializer

8. Handle Request
   Steps 1-4

   Handle Response
   Steps 5-8
Handler Class Hierarchy

<<interface>>
HandlerChain

* uses
<<interface>>
Handler

uses
<<interface>>
MessageContext

uses
<<interface>>
SOAPMessageContext

0..1
<<interface>>
SOAPMessage

Message Handler
**Handler Configuration**

- Handlers are configured programmatically *via* `HandlerRegistry`
  - The `HandlerRegistry` is available *via* the `getHandlerRegistry` method on `Service`
  - `HandlerRegistry` has methods to
    - `getHandlerChain`
    - `setHandlerChain`

- J2EE Deployment Descriptor for Handlers *(JSR109)*
package javax.xml.rpc.handler;

public interface Handler{
    boolean handleRequest(MessageContext ctx);
    boolean handleResponse(MessageContext ctx);
    boolean handleFault(MessageContext ctx);
}
Header Handling

- The handleRequest() method does the following
  - Return `true` to indicate that the processing of the handler chain should continue
  - Return `false` to indicate that the handler chain is blocked.
    - Processing continues in on the handleResponse() method of this handler instance and the handlers backward in the chain
  - Throw a `JAXRPCException` or other `RuntimeException` for a runtime error (in which case the HandlerChain will generate a SOAP Fault)

- The handleResponse() mechanism is similar, except for the processing false case
  - After blocking (return false), no other handlers will be processed
Handler Chains

A Handler is associated with a SOAP header block using the qualified name of the outermost element of each header block.

A Handler chain performs the following steps during SOAP processing:

- Identify the set of SOAP actor roles in which this handler chain is to act.
- Identify all mandatory header blocks for this role.
- If one or more of the mandatory blocks are not handled by this node then generate a SOAP MustUnderstand fault.
- Otherwise, process all the header blocks by invoking the chain of handlers.
- If processing is unsuccessful, generate exactly one SOAP fault to propagate to the client.
Handler Caveats

- Usually need to be defined in pairs (client, server)
- Client cannot communicate requirements to server handler and vice-versa
  - Not described in WSDL
- Client cannot alter the signature, hence, can’t change the operation or type
- Customers need to consider performance issues
  - Cost associated with each Handler
JSR109 – Enterprise Services Programming Model
JSR-109 – Enterprise Web Services

- Often known as ‘Web Services for J2EE’ or just JSR109
  - Spec actually calls it ‘Web Services for J2EE’!
- Began as V1.0 (JSR109)
  - An addition to J2EE 1.3
- V1.1 added WS-I Basic Profile support and bug fixes (JSR 921)
- V1.1 specification released January 23rd 2004
  - Required by J2EE 1.4
JSR-109 – Enterprise Web Services

- **Key Web Services Objective**
  - Achieve interoperability across heterogeneous platforms and runtimes

- Basically filling the gaps that the JAX-RPC specification left open for use of Web services in a J2EE Application Server

- **JSR 109 – Web Services for J2EE**
  - Facilitates the building of interoperable Web Services in J2EE
  - Standardization of the deployment of Web Services in a J2EE container
  - Specifically addresses
    - Client Access to Web Services
    - Web Service Lifecycle
    - Web Service Deployment
JSR 109 – Features

- Basically filling the gaps that the JAX-RPC specification left open for use of web services in a J2EE Application Server
- Standard Web Services Deployment Descriptors
- Defines a J2EE compliant deployment/packaging model for Web Services on the server side and for the client
  - New deployment descriptors for web services
- Server-side programming model
  - Stateless Session EJB as implementation of a web service
  - Entity and Stateful cannot
- Client-side programming model
  - EJB, Servlet/JSPs, Application Client as client to Web Services
JSR 109 – Features

- J2EE Container required
- The SEI acts like the EJB’s remote interface (or it must contain a subset of the remote interface’s methods)
- The application server (WebApp for HTTP) must route incoming SOAP requests to the appropriate EJB
- New deployment descriptors are defined
- Integrating Handlers into a J2EE container environment
  - Lifecycle
  - Declaration (deployment descriptors)
Overview

- Defines web services support within a J2EE environment
- Leverages JSR 101 work
- Defines a J2EE programming model
- Defines a deployment model

WSDL document

- `<portType>`
- `<SOAP binding>`
- `<service>`
- `<port>`

JAXRPC mapping

- `<portType>`
- `<SOAP binding>`
- `<service>`
- `<port>`

Generated Svc Intfc

- Use JNDI to access svc intf
  - Use Svc Intfc to access endpoint interface
    - `getPort(port,SEI)`
    - `getPort()`
    - `createCall()`

Endpoint Intfc

J2EE DDs

- `<service-ref>`
  - `<service-interface>`
  - `<wsdl-file>`?
  - `<jaxrpc-mapping-file>`?
  - `<service-endpoint-interface>`?
- `<service-ref>`

WSDL to Java emitter

- Use JNDI to access svc intf
  - Use Svc Intfc to access endpoint interface
    - `getPort(port,SEI)`
    - `getPort()`
    - `createCall()`
JSR-109 Server Programming Model

- Attempts to standardize the deployment of Web Services
- Defines deployment of
  - Servlet based implementation bean in a Web Container
  - Stateless Session EJB implementation in an EJB container
- Use of the Service Endpoint Interface (SEI) defined by JAX-RPC
  - Defines the method signatures of the Web Service
  - Must follow the Java and WSDL mapping rules defined by JAX-RPC
Server Side Programming

- Port defined the Server view of Web Service
- Port component services the operations defined in WSDL
- Port component has Service Endpoint Interface and Service Implementation that implements the Interface
- Service Implementation can be:
  - Stateless Session EJB
  - Java Bean (also referred to as JAX-RPC Service Endpoints)
JSR-109 Service Bean Implementation

- The Service Implementation Bean must
  - Implement all the methods defined by the SEI
  - For EJBs
    - SEI methods must be a subset of the remote interface methods
- The application server (WebApp for HTTP) must route incoming SOAP requests to the appropriate Service Bean Implementation
  - New deployment descriptors are defined
- The Service Implementation Bean and state
  - The container may create and pool multiple instances of the bean to optimize request handling
  - Thus carrying state may be questionable and risky
**JSR-109 Service Bean Implementation**

- WSDL is used as the contract for the Web Service
- The WSDL to SEI mapping MUST follow the Java and WSDL mapping rules defined by JAX-RPC
Service LifeCycle

- LifeCycle is controlled by the associated container
- In general, life cycle phases are
  - Instantiation
  - Initialization
  - Execution
  - Removal
Web Service LifeCycle

- **Instantiation**
  - Container invokes the bean’s `newInstance()` method

- **Initialization**
  - Container initializes via the container specific method
    - Servlet based service – `init()`
    - EJB based service – `setSessionContext()`, `ejbCreate()`

- **Execution**

- **Removal**
  - Container removes the service
    - Servlet based service – `destroy()`
    - EJB based service – `ejbRemove()`
Bean Based Implementation

- **Service**
  - Does not exist
    - newInstance()
    - init()
  - Exists and ready
    - destroy()
    - Handle SEI method requests
EJB Based Implementation Bean

Service Does not exist and not referenced
- Container crash or bean timeout

Service exists and not referenced
- Client's methods on reference generates NoSuchObjectException or NoSuchObjectLocalException
- Object.remove(), home.remove(), System exception in bean, Container crash or bean timeout
- handle.getEJBObject()

Service exists and referenced
- Release reference

Service Does not exist and referenced
- Release reference
- Client's methods on reference
Server Side Packaging

- Port components (SEI and Implementation)
  - In their respective modules – Web or EJB

- Web Services DD – webservices.xml
  - For EJB, in the Module META-INF directory
  - For Web, in the Module WEB-INF directory

- WSDL and Mapping files
  - Located relative to the module
    - Typically co-located with the module descriptor
  - Referenced in the Deployment Descriptor
Service Deployment Descriptors

- Specify the set of Web service descriptions to be deployed and their dependencies on container resources and services
- Specify any necessary type mappings
- Specify the Java artifact (JavaBean / EJB) implementing the service
Client Programming Models
Types of Client Access to Service

- **Client-Managed Access** (Standalone Java Client)
  - Lookup of service and stub defined by JAX-RPC (not defined by JSR 109)
  - Does not require a J2EE container
  - The client code must know fully-specified URL of the WSDL and therefore the Web service address/endpoint

- **Container-Managed Access** (Web Services for J2EE Client)
  - Lookup of SEI and stub is defined by JSR 109
  - Runs in a J2EE Container and uses J2EE run-time to access/invoke the methods of a Web Service
  - The client code does not know the URL of the target Web service
  - It only has either the SEI or the Port Type and bindings part of the WSDL
  - Can be
    - J2EE Application client
    - Web component (Servlet/JSP)
    - EJB component
JSR-109 Client Programming Model

- Provides a client view of the Web service in a J2EE environment
- Runtime details are transparent to the client
  - Protocol binding
  - Transport
- A Web Service is invoked much like invoking a method locally
Client Model Invocation Styles

- Three invocation styles:
  - **Static (generated) stub invocation**
    - Java class is statically bound to an SEI, WSDL Port and port component
    - Pretty much as with existing Web Services Implementations
  - **Dynamic Proxy invocation**
    - Java Class is not statically bound to the Web Service via a generated Stub
    - A dynamic proxy can be obtained at runtime by providing the SEI
    - Uses the JDK 1.3 Dynamic Proxy feature
  - **Dynamic Call DII invocation**
    - javax.xml.rpc.Call object is instantiated and configured to invoke the Web service
    - Used when a client needs dynamic, non-stub based communication with the Web Service
Client Program Flow

- Get Service
- Get stub from the service
- Instantiate and setup the objects which are parameters to the operation
- Use the stub to invoke the remote service operation
- Process return message or fault
JAX-RPC Client Using a Static Stub

- Uses the Service Factory class to create instance of a javax.xml.rpc.Service class

```java
Service addressBookService = ServiceFactory.newInstance().createService(
    new URL("file", ", " Addressbook.wsdl "),
    new QName("http://addr", "AddressBookService"));
```

- On the Service class, invoke getPort with the Port name, to return an instance of a stub that implements the SEI interface
  - The stub will be generated during deployment and are vendor-specific
  - The stub is the client representation of an instance of the Web Service

```java
AddressBook stub = (AddressBook) addressBookService.getPort(
    new QName("http://addr", "AddressBook"), AddressBook.class);
```

- Client uses the stub to drive the Web Service request to the endpoint that implements the Web service

```java
Address response = stub.getAddress();
```
JAX-RPC Client Running in J2EE Container

- Client uses JNDI lookup that returns a Service object

- Using the Service Object, the client gets the stub (that implements the Service Interface)

  ```java
  InitialContext initCtx = new InitialContext();
  AddressBookService addressBookService = (AddressBookService) initCtx.lookup("java:com/env/service/AddressBook");
  AddressBook stub = addressBookService.getPort(AddressBook.class);
  ```

- Client uses the stub to drive the Web Service request

  ```java
  Address response = stub.getAddress();
  ```
Dynamic Invocation Model

- Implement `javax.xml.rpc.Call`

```java
Service service = ServiceFactory.newInstance().createService(null);

Call call = service.createCall();
URL u = new URL("http://localhost:6080/AddresBookService/services/AddressBook");
call.setTargetEndpointAddress(u);
call.setProperty(Call.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
call.setProperty(Call.SOAPACTION_URI_PROPERTY, "getAddress");
call.setProperty(Call.ENCODINGSTYLE_URI_PROPERTY, "http://schemas.xmlsoap.org/soap/encoding/");
call.setOperationName(new QName("urn:address-book", "getQuote"));
call.addParameter("name", XMLType.XSD_STRING, ParameterMode.IN);
call.setReturnType("urn:address", AddressBook.class);

Object result = call.invoke(new Object[] {name = "John Doe"});

return (Address) result;
```
Choosing the Client API to Use

- **Static**
  - These are generated by the tooling
  - Use when you know the service location is unlikely to change
  - You want compile time type checking of use of stub

- **Dynamic Proxy**
  - Dynamic Proxy allows you to use a service endpoint interface without a generated stub class
  - Program to the static SEI, but can change the location of the service endpoint
    - The SEI allows you to retrieve a port and pass in the endpoint URL

- **DII**
  - Use when you need to figure out the service location / signature dynamically
  - Allows late/dynamic binding
  - No service endpoint interface is required
JSR-109 Deployment Descriptors
JSR-109 Deployment

- Defines
  - Set of web service descriptions to be deployed
    - their dependencies on container resources and services
  - Necessary type mappings
  - Specify the Java artifact (JavaBean / EJB) implementing the service

Diagram:
- Find
- Publish
- Bind
- Deployment
- Tooling
- Runtime
Deployment Descriptors

- **Service Deployment Descriptor**
  - webservices.xml
  - Packaged in same directory as module deployment descriptor
    - META-INF for EJB and Application client modules
    - WEB-INF for WAR modules
  - Stateless Session Bean Service Implementation defined by `session` element in `ejb-jar.xml`
  - Web container Service Implementation defined by `servlet-class` element in `web.xml`

- **Client Deployment Descriptor**
  - webservicesclient.xml
  - Packaged in same directory as module deployment descriptor
    - META-INF for EJB and Application client modules
    - WEB-INF for WAR modules

- **JAX-RPC Mapping Deployment Descriptor**
  - `<service>_mapping.xml`
  - Packaged in same directory as module deployment descriptor
    - META-INF for EJB and Application client modules
    - WEB-INF for WAR modules
JSR-109 Service Deployment Descriptors

- JSR-109 builds on top of JAX-RPC and defines two deployment descriptors
  - webservices.xml
    - Defines the structural information of the deployed Web services
    - Each WSDL defines service is mapped to a description in webservices.xml
  - JAX-RPC mapping file
    - The name of this file is not dictated by the specification
    - Referenced by the webservices.xml
Each description element MUST contain

- **Name**
  - References the service in the WSDL file

- **WSDL file reference**
  - References the WSDL file location in the J2EE module

- **JAX-RPC mapping file reference**
  - References the Mapping file in the J2EE module
<webservices>
  <webservice-description>
    <webservice-description-name>ExchangeRateService</webservice-description-name>
    <wsdl-file>WEB-INF/wsdl/Sample.wsdl</wsdl-file>
    <jaxrpc-mapping-file>WEB-INF/Sample_mapping.xml</jaxrpc-mapping-file>
    <port-component>
      <port-component-name>ExchangeRate</port-component-name>
      <wsdl-port>
        <namespaceURI>http://sample</namespaceURI>
        <localpart>ExchangeRate</localpart>
      </wsdl-port>
      <service-endpoint-interface>sample.ExchangeRatePortType</service-endpoint-interface>
      <service-impl-bean>
        <servlet-link>sample_ExchangeRateBindingImpl</servlet-link>
      </service-impl-bean>
    </port-component>
  </webservice-description>
  ...
</webservices>
WSDL and webservicess.xml

WSDL

```xml
<wSDL:service name="StockQuoteService">
  <wSDL:port binding="intf:GetQuoteBinding" name="GetQuote">
    <wSDLsoap:address location="http://www.example.com/stockquote/services/getquote"/>
  </wSDL:port>
  <wSDL:port binding="intf:PurchaseBinding" name="Purchase">
    <wSDLsoap:address location="http://www.example.com/stockquote/services/purchase"/>
  </wSDL:port>
</wSDL:service>
<wSDL:service name="ExchangeRateService">
  <wSDL:port binding="intf:ExchangeRateBinding" name="ExchangeRate">
    <wSDLsoap:address location="http://www.example.com/exchangerate/services/getrate"/>
  </wSDL:port>
</wSDL:service>
```

webservices.xml

```xml
<webservices>
  <webservice-description name="StockQuoteService">
    <wsdl-file>WEB-INF/wsdl/Sample.wsdl</wsdl-file>
    <jaxrpc-mapping-file>WEB-INF/Sample_mapping.xml</jaxrpc-mapping-file>
  </webservice-description>
  <webservice-description name="ExchangeRateService">
    <wsdl-file>WEB-INF/wsdl/Sample.wsdl</wsdl-file>
    <jaxrpc-mapping-file>WEB-INF/Sample_mapping.xml</jaxrpc-mapping-file>
  </webservice-description>
</webservices>
```
webservices.xml – Ports

- Port
  - Defines the server view of the service

- Contains
  - Name
  - WSDL port reference
    - Namespace URI
    - WSDL local name of the wsdl:port
  - SEI reference
    - Fully qualified class name of the SEI
  - SIB reference
    - Web Service implementation
      - Servlet based service – references a servlet element in the web.xml
      - EJB based service – references an EJB defined in ejb-jar.xml
webservices.xml – Port component

```xml
<?xml version='1.0' encoding='UTF-8'?>
<wsdl:definitions
    targetNamespace='http://sample'*
……
<wsdl:service name="StockQuoteService">
    <wsdl:port binding="intf:GetQuoteBinding" name="GetQuote">
        <wsdlsoap:address location="http://www.example.com/stockquote/services/getquote"/>
    </wsdl:port>
    <wsdl:port binding="intf:Pur...`
Servlet Based Service Port

```xml
<webservices>
  <webservice-description>
    ...
    <port-component>
      <port-component-name>GetQuote</port-component-name>
      <wsdl-port>
        <namespaceURI>http://sample</namespaceURI>
        <localpart>GetQuote</localpart>
      </wsdl-port>
      <service-endpoint-interface>sample.GetQuotePortType</service-endpoint-interface>
      <service-impl-bean>
        <servlet-link>sample_GetQuoteBindingImpl</servlet-link>
      </service-impl-bean>
    </port-component>
    ...
  </webservice-description>
</webservices>
```

```xml
<servlet>
  <servlet-name>sample_GetQuoteBindingImpl</servlet-name>
  <servlet-class>sample.GetQuoteBindingImpl</servlet-class>
</servlet>
```

 webservices.xml

 web.xml
EJB Based Service Port

```xml
<enterprise-beans>
  <session>
    <ejb-name>GetQuoteBindingImpl</ejb-name>
    <home>sample.GetQuotePortTypeHome</home>
    <remote>sample.GetQuotePortType_RI</remote>
    <ejb-class>sample.GetQuoteBindingImpl</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
  </session>
</enterprise-beans>

<webservice-description>
  ...
  <port-component>
    <port-component-name>GetQuote</port-component-name>
    <wsdl-port>
      <namespaceURI>http://sample</namespaceURI>
      <localpart>GetQuote</localpart>
    </wsdl-port>
    <service-endpoint-interface>sample.GetQuotePortType</service-endpoint-interface>
    <service-impl-bean>
      <ejb-link>GetQuoteBindingImpl</ejb-link>
    </service-impl-bean>
  </port-component>
  ...
</webservice-description>
```

```
ejb-jar.xml

webservices.xml
```
Handlers

- JSR-109 also standardizes the support for JAX-RPC handlers

- Handlers can
  - Pre-process a request before it is dispatched to a Web Service endpoint
  - Post-process a response before it is sent to the client
  - Access and modify the SOAP Header and content if the SOAP binding protocol is used
  - Handler MUST NOT CHANGE SOAP message structure, Operation name, number of parts in the message, or types of the message parts

  SOAP Fault is send if Handler does this
Handlers

```
<port-component>
  <port-component-name>GetQuote</port-component-name>
  <wsdl-port>
    <namespaceURI>http://sample</namespaceURI>
    <localpart>GetQuote</localpart>
  </wsdl-port>
  <service-endpoint-interface>sample.GetQuotePortType</service-endpoint-interface>
  <service-impl-bean>
    <servlet-link>sample_GetQuoteBindingImpl</servlet-link>
  </service-impl-bean>
  <handler>
    <handler-name>sample.ValidationHandler</handler-name>
    <handler-class>sample.ValidationHandler</handler-class>
  </handler>
  <handler>
    <handler-name>sample.LoggingHandler</handler-name>
    <handler-class>sample.LoggingHandler</handler-class>
    <init-param>
      <param-name>level</param-name>
      <param-value>warning</param-value>
      <description>Logging level</description>
    </init-param>
    <soap-header>
      <namespaceURI>http://sample</namespaceURI>
      <localpart>GetQuote</localpart>
    </soap-header>
    <soap-role>LoggingHandler</soap-role>
  </handler>
</port-component>
```
JSR-109 Client Deployment Descriptors

- Similar to service descriptors – two deployment descriptors
  - webservicesclient.xml
    - Defines each Web Service referenced by the client
  - jaxrpcmapping.xml
    - Defines the JAX-RPC mapping used by the client to access the Web Services
Each service referenced by the client is defined by a service-ref which contains

- **Name**
  - References the service in the WSDL file

- **Service Interface**
  - Fully qualified class name of the Service Interface

- **WSDL file reference**
  - References the WSDL file location in the J2EE module

- **JAX-RPC mapping file reference**
  - References the Mapping file in the J2EE module
<webservicesclient>
   <service-ref>
      <description>WSDL Service ExchangeRateService</description>
      <service-ref-name>service/ExchangeRateService</service-ref-name>
      <service-interface>sample.ExchangeRateService</service-interface>
      <wsdl-file>WEB-INF/wsdl/Sample.wsdl</wsdl-file>
      <jaxrpc-mapping-file>WEB-INF/Sample_mapping.xml</jaxrpc-mapping-file>
      <service-qname>
         <namespaceURI>http://sample</namespaceURI>
         <localpart>ExchangeRateService</localpart>
      </service-qname>
      <port-component-ref>
         <service-endpoint-interface>sample.ExchangeRatePortType</service-endpoint-interface>
      </port-component-ref>
   </service-ref>
   ...
</webservicesclient>
Client Side Handlers

- Client side handlers are associated with service references
  - Server side handlers are associated with port component references
- They can
  - Process a request before it is sent to the service endpoint
  - Process a response before it is returned to the client
- Defined the same way as Service Handlers
  - Additional the port name parameter
  - Port names are used to associate handlers with WSDL ports
Client Side Handlers

<webservicesclient>
  <service-ref>
    ...
    <port-component-ref>
      <service-endpoint-interface>sample.PurchasePortType</service-endpoint-interface>
    </port-component-ref>
    <port-component-ref>
      <service-endpoint-interface>sample.GetQuotePortType</service-endpoint-interface>
    </port-component-ref>
  </service-ref>
  <handler>
    <handler-name>sample.LoggingHandler</handler-name>
    <handler-class>sample.LoggingHandler</handler-class>
    <init-param>
      <param-name>level</param-name>
      <param-value>information</param-value>
      <description>Logging level</description>
    </init-param>
    <soap-header>
      <namespaceURI>http://sample</namespaceURI>
      <localpart>GetQuote</localpart>
    </soap-header>
    <soap-role>LoggingHandler</soap-role>
    <port-name>GetQuote</port-name>
  </handler>
  ...
</webservicesclient>
**JAX-PRC Mapping**

- Mechanism for standardizing the Java ↔ WSDL mappings
- JAX-RPC provides the rules for the mappings
- JSR-109 provides the XML-based deployment descriptor standardized representation
- No standardized name
  - Name determined by the jaxrpc-mapping-file element in webservices.xml or webservicesclient.xml
- File structure matches closely to the WSDL file structure
JAX-RPC Mapping

- `<package-mapping>` element
  - Defines Java package and namespace mapping

```xml
<java-wsdl-mapping>
  <package-mapping>
    <package-type>sample</package-type>
    <namespaceURI>http://sample</namespaceURI>
  </package-mapping>
  ...
</java-wsdl-mapping>
```
WSDL Service ↔ Service Interface Mapping

```
<wSDL:service name="StockQuoteService">  
  <wSDL:port binding="intf:GetQuoteBinding" name="GetQuote">  
    <wsdlsoap:address  
      location="http://www.example.com/stockquote/services/getquote"/>  
  </wSDL:port>  
  <wSDL:port binding="intf:PurchaseBinding" name="Purchase">  
    <wsdlsoap:address  
      location="http://www.example.com/stockquote/services/purchase"/>  
  </wSDL:port>  
</wSDL:service>

<service-interface-mapping>  
  <service-interface>sample.StockQuoteService</service-interface>  
  <wSDL-service-name>  
    <namespaceURI>http://sample</namespaceURI>  
    <localpart>StockQuoteService</localpart>  
  </wSDL-service-name>  
  <port-mapping>  
    <port-name>Purchase</port-name>  
    <java-port-name>Purchase</java-port-name>  
  </port-mapping>  
  <port-mapping>  
    <port-name>GetQuote</port-name>  
    <java-port-name>GetQuote</java-port-name>  
  </port-mapping>  
</service-interface-mapping>
```

WSDL

JAX-RPC Mapping file
WSDL Mappings

```xml
<wSDL:message name="getQuoteRequest">
  <wSDL:part element="intf:ticker" name="parameter"/>
</wSDL:message>

<wSDL:message name="getQuoteResponse">
  <wSDL:part element="intf:price" name="parameter"/>
</wSDL:message>

<wSDL:portType name="GetQuotePortType">
  <wSDL:operation name="getQuote">
    <wSDL:input message="intf:getQuoteRequest" name="getQuoteRequest"/>
    <wSDL:output message="intf:getQuoteResponse" name="getQuoteResponse"/>
  </wSDL:operation>
</wSDL:portType>

<wSDL:binding name="GetQuoteBinding" type="intf:GetQuotePortType">
  <wSDLsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <wSDL:operation name="getQuote">
    <wSDLsoap:operation soapAction="/"/>
    <wSDL:input name="getQuoteRequest">
      <wSDLsoap:body use="literal"/>
    </wSDL:input>
    <wSDL:output name="getQuoteResponse">
      <wSDLsoap:body use="literal"/>
    </wSDL:output>
  </wSDL:operation>
</wSDL:binding>
```
WSDL Bindings Mappings

```xml
<service-endpoint-interface-mapping>
  <service-endpoint-interface>sample.GetQuotePortType</service-endpoint-interface>
  <wsdl-port-type>
    <namespaceURI>http://sample</namespaceURI>
    <localpart>GetQuotePortType</localpart>
  </wsdl-port-type>
  <wsdl-binding>
    <namespaceURI>http://sample</namespaceURI>
    <localpart>GetQuoteBinding</localpart>
  </wsdl-binding>
  <service-endpoint-method-mapping>
    <java-method-name>getQuote</java-method-name>
    <wsdl-operation>getQuote</wsdl-operation>
    <method-param-parts-mapping>
      <param-position>0</param-position>
      <param-type>sample.Ticker</param-type>
    </method-param-parts-mapping>
    <wsdl-message-mapping>
      <wsdl-message>
        <namespaceURI>http://sample</namespaceURI>
        <localpart>getQuoteRequest</localpart>
      </wsdl-message>
      <wsdl-message-part-name>parameter</wsdl-message-part-name>
      <parameter-mode>IN</parameter-mode>
    </wsdl-message-mapping>
    <wsdl-return-value-mapping>
      <method-return-value>sample.Price</method-return-value>
    </wsdl-return-value-mapping>
  </service-endpoint-method-mapping>
</service-endpoint-interface-mapping>
```

JAX-RPC Mapping file
Java ↔ XML Mapping

```xml
<wsdl:types>
    <schema
        elementFormDefault="qualified"
        targetNamespace="http://sample"
        xmlns="http://www.w3.org/2001/XMLSchema">
        <element name="ticker">
            <complexType>
                <sequence>
                    <element name="item" type="xsd:string"/>
                </sequence>
            </complexType>
        </element>
    </schema>
</wsdl:types>

<java-xml-type-mapping>
    <class-type>sample.Ticker</class-type>
    <root-type-qname
        namespaceURI="http://sample"
        localpart="ticker"/>
</java-xml-type-mapping>

<java-xml-type-mapping>
    <class-type>java.lang.String</class-type>
    <root-type-qname
        namespaceURI="http://www.w3.org/2001/XMLSchema"
        localpart="string"/>
</java-xml-type-mapping>

JAX-RPC Mapping file

WSDL
V1.1 – JSR 921

- Maintenance release for JSR109
  - Is part of J2EE 1.4
- No dependency between EJB Remote Interface (RI) and Service Endpoint Interface (SEI) for EJBs exposed as web services
  - JSR109 mandated SEI to include subset of RI
  - SEI and RI both act as remote interfaces to access a J2EE component
  - EJB 2.1 deployment descriptor contains reference to SEI
  - “SEI is a peer to Remote Interface/Local Interface”
V1.1 – JSR 921

- Replaced DTD deployment descriptors with XML Schema deployment descriptors
- J2EE 1.4 changes
- Updated to support WS-I Basic Profile 1.0
- Clarified interoperability requirements
- Support for JAX-RPC handlers for EJB web services now required
  - Was optional in JSR109
- MessageContext accessible in EJB container
- Client DDs merged with normal J2EE DDs
What about Service Discovery?

- Remote Service Discovery must be done by developer
  - Not defined by JSR-109 or JAX-RPC
  - Locate and retrieve WSDL *via*:
    - URL
    - UDDI
    - UDDI4J
    - JAX-R

- From the WSDL pick one of:
  - Service and Port
  - Service and PortType

- Can then use the JAX-RPC client-managed programming model to connect to a particular Web service implementation
JAXR

- JAX-RPC and JSR-109 do not define how to achieve Remote Service Discovery
- Developer can locate and retrieve WSDL via URL, UDDI, UDDI4J, JAX-R
- Java API for XML Registries (JAX-R)
  - Uniform Standard API for accessing XML registries (including UDDI)
  - J2EE-based standards access to UDDI
    - Organization -> businessEntity
    - Service -> businessService
    - ServiceBinding -> bindingTemplate
    - ExternalLink -> discoveryURL
    - Concept/Classification Scheme -> tModel
Interoperability  WS-I

- Organization focused on promoting interoperability between Web Services
- Main goal is to provide guidance in the standardization of Web Services between different platforms, applications, and programming languages
- Defines profiles, which are a set of different specifications
  - WS-I Basic 1.0 Profile currently available
  - WS-I 1.1 Profile adds support for attachments
WS-I Basic Profile

- **HTTP V1.1**
  - Specific on HTTP errors and response codes
  - must not require cookie support

- **XML 1.0 and XML Schema 1.0**
  - May use any construct from Schema 1.0

- **SOAP V1.1**
  - Use of SOAP encoding disallowed
  - Specific on structure of fault and when to generate faults
  - “Trailers” (element content after soap-env:Body) disallowed
  - Use of SOAPAction, soap-env:actor clarified

- **WSDL V1.1 with SOAP Encoding = Literal**
  - Exclude use of wsdl:import for XSD files
  - Numerous spec clarifications
  - Only one element in the Body of the element

- **UDDI V2.0**
  - Established category to identify WS-I conformant entities
  - Models must use WSDL as descriptive language
Summary

- With J2EE 1.4 Web Services is becoming a foundation technology with the addition of
  - Web Services 1.1
  - JAX-RPC 1.1
  - SAAJ 1.2
  - JAXR 1.0
  - WS-I Basic Profile 1.0
Resources

- **JSR 101 (JAX-RPC)**

- **JSR 109**
  - [http://jcp.org/jsr/detail/109.jsp](http://jcp.org/jsr/detail/109.jsp)

- **Introduction to Web Services**

- **WS-I Basic Profile**
  - [http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html](http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html)
Resources – Standards Bodies

- http://www.w3.org/TR/SOAP/
- http://www.w3.org/TR/wsdl
- http://www.uddi.org
- http://www.WS-I.org
- http://xml.apache.org/soap
- http://www.xmethods.com
WSDL ↔ Java Mapping

- **JSR 101** defines a standardized mapping model from WSDL to Java artifacts
  - WSDL Port Type maps to Java Service Endpoint Interface (SEI)
  - WSDL Service maps to a Java Service Interface
  - WSDL complex elements/parts maps to a Java Bean

```
<wSDL:portType name="Warehouse">
  <wSDL:Operation name="Order">
    ….
  </wSDL:Operation>
</wSDL:portType>

<wSDL:service name="WHService">
  <wSDL:port binding="…..">
    <wSDLsoap:address location="…..">
    </wSDL:port>
  </wSDL:port>
</wSDL:service>
```