WA2171 Programming Java SOAP and REST Web Services - WebSphere 8.5 - Eclipse

Web Age Solutions Inc.
USA: 1-877-517-6540
Canada: 1-877-812-8887
Web: http://www.webagesolutions.com

For customizations of this book or other sales inquiries, please contact us at:

USA: 1-877-517-6540, email: getinfousa@webagesolutions.com
Canada: 1-877-812-8887 toll free, email: getinfo@webagesolutions.com

Table of Contents

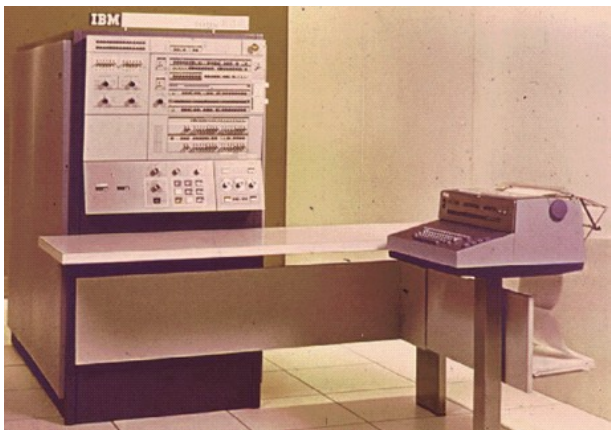
Chapter 1 - Overview of Java Web Services

Objectives

Key objectives of this chapter

- Overview of web services
- Java web service implementation options
- Intro to JAX-WS
- Intro to JAX-RS

1.1 A Conceptual Look at Services



- monolithic/proprietary design
- expensive
- fixed architecture and capabilities
- technology locked
- vendor locked

- standard interfaces (serial, parallel, USB)
- competitive pricing
- component-based design
- adaptable to technology innovation
- vendor-neutral

A Conceptual Look at Services

The evolution that has taken place in the computing industry is analogous to the evolution that service orientation represents for enterprise systems. It is about transitioning from closed, proprietary solutions to open, standards-based solutions. This carries with it a host of benefits with respect to flexibility and cost.

1.2 Defining Services

- The ‘S’ in SOA is embodied as the configurable enterprise assets that comprise the various layers.
- Abstractly, services can be described as follows:
  - Performs one or more well-defined functions.
  - Is loosely coupled and self contained.
  - Is almost always stateless.
- Various enterprise elements can function as services:
  - CICS/IMS mainframe systems
  - Distributed computing components (CORBA, EJB, DCOM, etc.)
  - Messaging systems (MSMQ, IBM MQ, etc.)
  - Packaged applications (ERP, CRM, etc.)
  - Home-grown applications
  - Web services (most popular)

1.3 Benefits of Web Services

- One of the benefits of web services is that the service implementation and the web service client can use different programming languages
  - The standards that might be involved in web services are independent of a particular programming language
- This provides several advantages:
  - Protect investment in IT legacy systems by using Web services to wrap legacy software systems for integration with modern IT systems.
  - Deliver new IT solutions faster and at a lower cost by focusing code development on core business and using Web services applications for non-core business programming.

- ◊ Integrate business processes with customers and partners at lower cost.

1.4 Many Flavors of Services

- ■ Web Services come in all shapes and sizes
  - ◊ XML-based services (SOAP, XML-RPC, RSS / ATOM Feeds)
  - ◊ HTTP services (REST, JSON, standard GET / POST)
  - ◊ Other services (FTP, SMTP)
- ■ While SOAP is the most common style of service, increasingly organizations are utilizing REST for certain scenarios
  - ◊ REpresentational State Transfer (REST), first introduced by Roy Fielding (co-founder of the Apache Software Foundation and co-author of HTTP and URI RFCs)
  - ◊ REST emphasizes the importance of resources, expressed as URIs
  - ◊ Used extensively by Amazon, Google, Yahoo, Flickr, and others

1.5 Java Web Service Implementation Choices

- ■ Java, like most programming languages, provides programming frameworks to make developing web services in Java easier
  - ◊ JAX-WS for Java SOAP web services
    - ■ This replaced JAX-RPC as the choice for Java SOAP web services in Java EE 5
    - ■ Still supports all of the advanced SOAP web service features JAX-RPC did and provides more
    - ■ Code for web service implementations and clients is more portable between servers of different vendors which was a problem with JAX-RPC
  - ◊ JAX-RS for Java REST web services
    - ■ This is relatively new to Java and was only required of Java

  application servers starting with Java EE 6
  - REST relies more on HTTP and there are fewer technical specifications for REST web services compared to SOAP
  - Having JAX-RS definitely helps to put some structure about how to implement REST services in Java
- You can use both types of web services within a single application

1.6 Future of JAX-RPC

- JAX-RPC is a technology that is proposed to be “pruned” from the Java EE standards
- Since JAX-WS is now available and is a better implementation option, future Java EE versions, starting with Java EE 7, may mark JAX-RPC as “optional” and not require that servers support it
  - Some servers may still support it but the idea is servers could be considered “fully compliant” with Java EE even without JAX-RPC support
- This “pruning” process is meant to steer people toward using better technologies and allow servers to maintain only support for the “best of breed” Java technologies
- Any organization currently using JAX-RPC should look to start migrating to JAX-WS.

1.7 Java SOAP Web Services with JAX-WS

- Defines a standard way to expose Java components as SOAP web services
  - Includes support for generating the WSDL of a SOAP web service directly from the Java code or Java code from a WSDL
- The first version of JAX-WS was 2.0 in Java EE 5 as a replacement for JAX-RPC
  - Java EE 6 requires support for JAX-WS 2.2 but there have not been major changes since 2.0

- JAX-WS makes many improvements over the previous API JAX-RPC. They fall mostly under these categories:
    - Much simpler programming model achieved through annotations like @WebService.
    - Much better vendor neutral code for service provider and consumer.
    - More advanced programming API for asynchronous Web Service development, message attachment using MTOM and others.

```
@WebService
public class Quote {
    public double getQuote(String symbol) { ... }
```

1.8 Java REST Web Services with JAX-RS

- Defines a standard way to expose Java components as REST-style web services
- JAX-RS is a separate Java specification and Java EE 6 requires support for JAX-RS 1.1
- Based on a number of annotations
    - @Path
    - @PathParam
    - HTTP Methods
        - @GET, @PUT, @POST, @DELETE

```
@GET
@Path("/thisResource/{resourceId}")
public String getResourceById
     (@PathParam("resourceId") String id) { … }
```

1.9 REST vs SOAP Summary

- REST
    - Ideal for use in Web-centric environments, especially as a part of Web Oriented Architecture (WOA) and Web 2.0

  - Takes advantage of existing HTTP tools, techniques, & skills
  - Little standardization and general lack of support regarding enterprise-grade demands (security, transactions, etc.)
- SOAP
  - Supports robust and standardized security, policy management, addressing, transactions, etc.
  - Tools and industry best practices for SOA and WS assume SOAP as the message protocol

1.10 Java and XML with JAXB

- JAXB applies mapping rules to bind XML types to Java.
- You can customize the default rules using JAXB annotations.
- Improvements to JAXB mapping have been a primary driver between improvements to Java web services and interoperability with other programming languages
- The following entities are mapped (bi-directionally):
  - XML namespace is mapped to Java package.
  - XML complexType is mapped to Java class.
  - Child elements and attributes of a complexType are mapped to JavaBean properties.
  - Java enum types are mapped to constrained XML types.
  - Type collections like java.util.List<T> are mapped to XML.

1.11 Java Web Service Clients

- Java applications can be clients of both types of web services
- JAX-WS provides a way to take the WSDL of a SOAP web service and generate client code that will take care of SOAP/XML programming
  - This means to the client application the SOAP web service is more just like a regular Java component

- Currently JAX-RS 1.1 does not provide a way to generate client code or even a client API to use
  - REST services are much more commonly used from JavaScript/AJAX applications than from Java applications
  - All JAX-RS implementations provide a client API but the code would be specific to that implementation and not portable
  - JAX-RS 2.0 will provide a Java client API but this is currently a weakness of JAX-RS
    - Since JAX-RS is based more on HTTP and requests and responses are “simpler” this is not too bad

1.12 Summary

- Java has two main specifications for web service support, JAX-WS and JAX-RS
- You can implement SOAP and REST web services in Java using code from these standards
- Java EE 6 requires support for both web service styles

Chapter 2 - Basic XML Schemas

Objectives

Key objectives of this chapter

- The goal of XML schema.
- Namespace
- Using schema from an XML document.
- Defining basic data types.
- Defining attribute.

2.1 What is XML Schema ?

- XML Schema defines:
  - ◊ Elements that can appear in a document.
  - ◊ Attributes that can appear in a document.
  - ◊ Which elements are child elements.
  - ◊ The sequence in which the child elements can appear.
  - ◊ The number of child elements.
  - ◊ Whether an element is empty or can include text.
  - ◊ Data types for elements and attributes.
  - ◊ Default values for elements and attributes.

2.2 Goals of Schema

- Special tools should not be required for maintaining schema documents.
- XML and HTML authors should not have to learn a new syntax to express schema information.
- Schemas should be extensible.
- The Schema language should be simple enough to encourage implementation in all XML processors.

- Schemas should meet the needs of Web-based applications that require standard encoding to facilitate document interchange.
- Schemas should meet the needs of Web-based applications that require additional validation beyond the capabilities of DTDs.
- The Schema language must support the ability of individual documents to comprise parts defined in several sources.

Goals of Schema

1. Special tools should not be required for maintaining schema documents.

   One of the complaints about DTDs was that they had a special syntax which introduced a need for a toolset focused directly on them. Because schemas are an XML grammar, no new toolset is required.

2. XML and HTML authors should not have to learn a new syntax to express schema information.

   Once again, schemas are an XML grammar. This means the declarations will all be done in an XML format, and, therefore, will be more familiar to the existing community.

3. Schemas should be extensible.

   This goal is accomplished by the open schema definition syntax which allows future authors to create specialized versions of a schema.

4. The Schema language should be simple enough to encourage implementation in all XML processors.

   Simply put, the standard is useless unless it is accepted and used.

5. Schemas should meet the needs of Web-based applications that require standard encoding to facilitate document interchange.

   To support this interchange, schemas include standard types found in databases and programming languages (e.g. floating point).

6. Schemas should meet the needs of Web-based applications that require additional validation beyond the capabilities of DTDs.

   Schemas give us additional primitive data types and allow us easy control over their ranges. In addition, we can derive new types from the existing base set to better meet our needs. Using regular expressions we can very tightly control validation of content.

7. The Schema language must support the ability of individual documents to comprise parts defined in several sources.

   Because XML documents can comprise other documents, or pieces of other documents, it follows that so should schemas. Schemas accomplish this naturally as they are part of XML itself. The use of namespaces in schemas give us a match to the capabilities of DTDs.

2.3 Goals of Schema

- Reuse of content model definitions should be easier than when using parameter entities.
- Schemas must be upwardly compatible with XML 1.0

2.4 Converting DTDs to Schema

```
<!ELEMENT Student (First, Last) >
<!ELEMENT First (#PCDATA) >
<!ELEMENT Last (#PCDATA) >
```

Converting DTDs to Schema

We will convert the above DTD to schema. For now, we will leave all the types as String.

2.5 Recall: Namespaces

- A tag or attribute can be associated with a namespace:
  - ◊ Declared by the reserved xmlns attribute:

```
<was:profile xmlns:was="http://www.webagesolutions.com">
```

  - ◊ Use a colon to separate the namespace prefix from the local name, e.g.:

```
<was:profile xmlns:was="http://www.webagesolutions.com">
    <was:training>.... </was:training>
    <was:consulting>.... </was:consulting>
</was:profile>
```

Recall: Namespaces

The namespace name should uniquely identify the context in which the tags have meaning. Because URLs are unique across the internet, they are a good choice to use. You can also declare a default namespace for an element and its children, for example:

```
<profile xmlns="http://www.webagesolutions.com">
```

```
   <training>.... </training>
   <consulting>.... </consulting>
</profile>
```

2.6 The equivalent schema

```
<?xml version="1.0"?>
<xsd:schema targetNamespace="http://www.students.org"
            xmlns="http://www.students.org"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified">
   <xsd:element name="Student">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref="First"/>
            <xsd:element ref="Last"/>
      </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="First" type="xsd:string"/>
   <xsd:element name="Last" type="xsd:string"/>
</xsd:schema>
```

The equivalent schema

The equivalent schema defines one new type Student—a type is similar to a class in object-oriented concepts—made up of two string variables: First and Last. Notice that while defining the Student type, we just refer to the variables First and Last. We define them later as String type in the last two lines.

The schema language itself is made up of reserved words such as schema, element, complexType and sequence. The whole schema document has schema as the parent element.

targetNamespace assigns a namespace for the new types defined in this schema document. The new types defined here are Student, First and Last. They belong to the namespace uniquely identified by the URL http://www.students.org.

xmlns and xmlns:xsd refers to two namespaces. All names that belong to the namespace pointed to by xmlns (the default namespace) does not need to be pre-qualified, whereas all names belonging to the xmlns:xsd namespace need to be pre-qualified by the string xsd:. An example of such a non-qualified name is First, while a qualified name will be xsd:complexType.

2.7 Sample instance document

```
<?xml version="1.0" encoding="UTF-8"?>
<Student xmlns="http://www.students.org"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.students.org/student.xsd">
   <First>Tapas</First>
   <Last>Banerjee</Last>
</Student>
```

Sample instance document

Again xmlns refers to the default namespace, whereas everything belonging to xmlns:xsi namespace has to be pre-qualified by xsi. The qualified namespace happens to be the one that defines the reserved words for schema instances such as schemaLocation.

The schemaLocation attribute says that the http://www.students.org namespace is defined by the file student.xsd.

2.8 Documents Needed

- Schema Document:
  - An XML document with predefined elements and attributes describing the structure of another XML document.
  - The root element is schema.
  - Saved with a xsd file extension.
- Instance Document:
  - The XML data document conforming to a schema.
  - Links the schema by using the namespace.

Documents Needed

The Schema Document: student.xsd

```
<?xml version = "1.0" ?>
<schema>
   …. Structure of another xml document
</schema>
```

The Instance Document: student.xml

```
<?xml version = "1.0" ?>
<student>
   ….. Data Structure
</student>
```

2.9 XML Schema Namespaces

- Used in the schema itself.
  - http://www.w3.org/2001/XMLSchema
  - Standard alias or prefix: xsd or xs.
  - Every schema document must use this namespace.
- Used in the instance document.
  - http://www.w3.org/2001/Schema-instance.
  - Standard alias or prefix: xsi.
- The prefix xmlns to declare namespaces.

XML Schema Namespaces

The namespace http://www.w3.org/2001/XMLSchema is the normal, expected namespace for the XML Schema documents.

The Schema Document: student.xsd

```
<?xml version = "1.0" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   …. Structure of another xml document
</xsd:schema>
```

The Instance Document: student.xml

```
<?xml version = "1.0" ?>
<student xmlns:xsi="http://www.org/2001/Schema-instance">
   ….. Data Structure
</student>
```

2.10 Link Documents to Schemas

- Two ways to link an XML document to a schema:
  - Use the **schemaLocation** attribute if the instance document declares its own namespace. The value of the attribute is a space separated list

of:

  - The namespace name for the document.
  - The location URL of the schema document.

- ◊ Use the **noNamespaceSchemaLocation** attribute if the instance document does not declare its own namespace. You need:
  - The location of the schema document.

Link Documents to Schemas

Each of these attributes in only a suggestion as to where a processor could locate the schema. A processor may choose to ignore the suggestion and locate the schema using some other means.

2.11 Link Documents to Schemas

- Using the schemaLocation attribute:

```
<?xml version="1.0" encoding="UTF-8"?>
<addr:addressBook
   xmlns:addr="http://www.webagesolutions.com/partner"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.webagesolutions.com/partner
   http://www.webagesolutions.com/schema/address.xsd">
   <address>…</address>
</addr:addressBook>
```

- For example of **noNamespaceSchemaLocation** see below.

Link Documents to Schemas

```
<?xml version = "1.0" ?>
<student01
   xmlns:xsi="http://www.org/2001/Schema-instance"
   xsi:noNamespaceSchemaLocation="student.xsd" >
</student01>
```

The student.xsd is the name of the schema document.

The student.xsd is on the same directory as of the student01.xml.

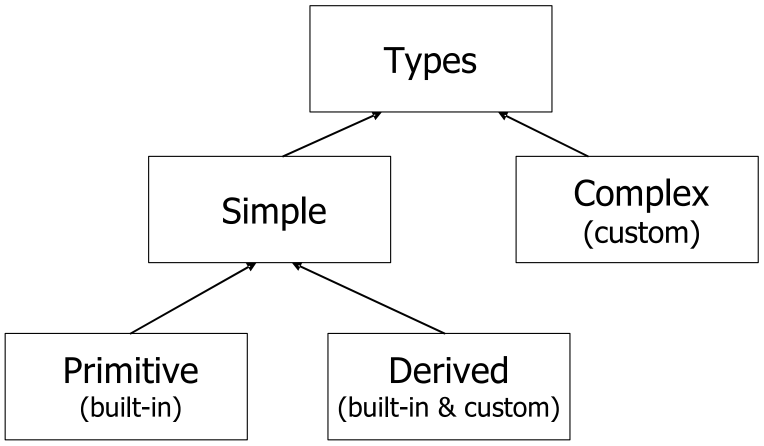
2.12 Inline element declarations

```
<?xml version="1.0"?>
<xsd:schema targetNamespace="http://www.students.org"
   xmlns="http://www.students.org"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified">
   <xsd:element name="Student">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="First" type="xsd:string"/>
          <xsd:element name="Last" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Inline element declarations

If we do not use any ref construct, then the schema that we defined a few pages back will look as above.

2.13 Schema Data Types



Schema Data Types

In an XML document, data is represented into two places:

1. Element content.

2. Attribute values.

Complex types are used when you want to define child elements or attributes. Simple types are made up of built-in types or their refinements.

2.14 Schema Type Definitions

- Simple types:
  - ◊ Contain only text.
  - ◊ Cannot contain attributes.
  - ◊ Cannot have children.
- Complex types:
  - ◊ Can contain sub elements.
  - ◊ Can contain attributes.
  - ◊ Can have children.

2.15 Schema Simple Data Types

- Primitive data types:
  - ◊ Can be used for element or attribute values.
  - ◊ Cannot contain children elements or attributes.
  - ◊ Are always built-in.
- Derived data types:
  - ◊ Contain well-formed XML that is valid.
  - ◊ They may be built-in or user-derived.
  - ◊ Allows the use of custom-created data types (e.g. a phone number field) using Facets

Schema Simple Data Types

The following are the common primitive datatypes:

- string

- boolean
- decimal

The following are some common derived datatypes:

- Integer
- IDREFS
- long

2.16 Primitive Data Types

- List of the common types:
  - xs:string
  - xs:decimal
  - xs:integer
  - xs:boolean
    - true/false or 1/0
  - xs:date
    - In format YYYY-MM-DD
  - xs:time
    - In format hh:mm:ss.sss

Primitive Data Types

- xs:string – A character string as a sequence of Unicode including space, tab, carriage return, and line feed.
- xs:decimal – A decimal number as a sequence of decimal digits separated by a period or by a decimal indicator.
- xs:integer – A whole number.
- xs:boolean – Binary logic as true, false, 1 or 0.
- xs:date – Represents a calendar date. The format is is: YYYY-MM-DD. The representation of February 15, 2002 is 2002-02-15.
- xs:time – The format is hh:mm:ss.sss

2.17 Simple Types

- Can contain only text.
- The text it contains can be:
  - Types included in the XML Schema Definition.
  - Custom types – You define yourself.
- Use the simpleType construct.
- Syntax:

```
<simpleType name="typeName" >
   list|restrictions
</simpleType>
```

Simple Types

- Example where SIN stands for Student Id Number:

```
<xs:simpleType name="SIN">
</xs:simpleType>
```

2.18 Facet – Restrictions on Element Content

- Common facets:
  - enumeration
  - length
  - maxLength
  - minLength
  - maxExclusive
  - minExclusive

Facet – Restrictions on Element Content

- enumeration – Defines an enumeration of values.
- length – Defines the length.
- maxLength – Defines the maximum length.

- minLength – Defines the minimum length.
- maxExclusive – Defines the largest value.
- minExclusive – Defines the smallest value.
- default – To specify the default value for the element.

```
<xs:element name="firstname" type="xs:string" default="yourName" />
```

- fixed – To specify that an element must contain a particular value.

```
<xs:element name="branch" type="xs:string" default="main" />
```

2.19 Using the Facet

- To restrict a type by enumeration.
- Syntax:

```
<simpleType name="typeName">
   <restriction base="baseType"
      <enumeration value="value1" />
      <enumeration value="value2" />
   </restriction>
</simpleType>
```

Using the Facet

Sample:

```
<xs:element name="car">
<xs:simpleType>
   <xs:restriction base="xs:string">
      <xs:enumeration value="Honda" />
      <xs:enumeration value="Toyota" />
      <xs:enumeration value="Volvo" />
   </xs:restriction>
</xs:simpleType>
</xs:element>
```

In the given example, the element called car is defines to be a simple type with a derived restriction based on the XML Schema data type string. It is also limited to the set of applicable values: Honda, Toyota and Volvo.

2.20 Using the Facet

- Syntax:

```
<facet value="value" fixed="true|false" />
```

- Sample:

```
<xs:simpleType name="SIN">
   <xs:restriction base="xs:string">
      <xs:length value="9">
   </xs:restriction>
</xs:simpleType>
```

Using the Facet

In the syntax above:

- facet – Restrictions in the element (previous slide).
- value – The value appropriate to the facet used.
- fixed – Lets you prevent the restriction being changed by a type derived from this one. The default value is false.

2.21 Samples using Regular Expressions

- Simple and built-in types

```
<xsd:simpleType name="ISBNType">
   <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d{1}-\d{5}-\d{3}-\d{1}"/>
      <xsd:pattern value="\d{1}-\d{3}-\d{5}-\d{1}"/>
      <xsd:pattern value="\d{1}-\d{2}-\d{6}-\d{1}"/>
   </xsd:restriction>
</xsd:simpleType>
```

Samples Using Regular Expressions

```
<xsd:simpleType name="NATelephoneNumber">
   <xsd:restriction base="xsd:string">
      <xsd:length value="12"/>
      <xsd:pattern value="\d{3}-\d{3}-\d{4}"/>
   </xsd:restriction>
</xsd:simpleType>
```

2.22 Define Simple Element Type

- Syntax:

```
<xsd:element name="nnnnn" type="ttttt" />
```

- In XML:

```
<lastname>Lynch</lastname>
<age>18</age>
```

- Corresponding element definition:

```
<xs:element name="lastname" type="xs:string" />
<xs:element name="age" type="xs:number" />
```

2.23 Element Declaration

- Global element declaration:
  - Are children of the root schema element.
  - Syntax:

```
<xsd:element name="nameofelement">
</xsd:element>
```

- Local element declaration:
  - Are not direct children of the root schema element.

Element Declaration

Local elements cannot be referenced.

2.24 Element Occurrence Indicators

- Can make use of the following:
  - minOccurs
  - maxOccurs
- Can only be declared on local element declaration.

- Sample:

```
<xs:element name="lastname" type="xs:string" minOccurs="1"
maxOccurs="1" />
```

Element Occurrence Indicators

To specify how many times an element may occur:

- minOccurs – Minimum number of occurrence (default 1).
- maxOccurs – Maximum number of occurrence (default 1).
- Use unbounded for maxOccur to specify that there is unlimited occurrence of this element.

2.25 Complex Type

- To define child elements and/or attributes of an element.
- Use the complexType element.
- Syntax:

```
<xs:element name="name">
   <xs:complexType>
      element content
   </xs:complexType>
</xs:element>
```

Complex Type

```
<xsd:element name="product">
   <xsd:complexType>
      <xsd:element name="make" …/>
   </xsd:complexType>
</xsd:element>
```

2.26 Attribute Declaration

- Cannot contain any child information.
- Attribute values are always simple types.
- They are unordered, cannot specify the order in which attributes should appear on the parent element.
- Are added inside the complex type definition for that element.

- Attributes should always be defined after child elements (e.g. after a sequence declaration)

Attribute Declaration

```
<?xml version="1.0" ?>
<schema>
   <element name="student">
      <complexType>
         <sequence>
            <!--- simple element type declarations --->
         </sequence>
         <!-- attribute declaration here -->
      </complexType>
   </element>
</schema>
```

2.27 Attribute Declarations

- Syntax:

```
<attribute name="" type="" />
```

- Appears nested inside the type or element declaration to which it belongs.
- Some examples of attribute declarations:

```
<xsd:attribute name="InStock"  type="xsd:boolean"
default="false"/>

<xsd:attribute name="Reviewer"  type="xsd:string" default="
"/>
```

Attribute Declarations

```
<xsd:element name="apple">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="variety" type="xsd:string"  use="required"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

2.28 Occurrence of Attributes

- Add the "use" attribute in the attribute declaration.
- The possible values are:
  - required
  - optional
  - prohibited
- Syntax:

```
<attribute name="attributename" use="occurrencevalue" />
```

Occurrence of Attributes

- required – Indicates that an attribute must appear.
- optional – When the attribute can either appear once or not at all (the default value).
- prohibited – Used to explicitly indicate that the attribute must not appear.

2.29 Value Constraints on Attributes

- Can make use of the following attributes:
  - default:
    - Add a default value to the attribute.
    - Note: if you have a default value, then the use value must be set to optional.
  - fixed:
    - Indicate that the value of the attribute is same as the value prescribed in the schema.

2.30 Sequence Element

- Used to list one or more child elements.
- Syntax:

```
<xs:element name="name">
```

```
   <xs:complexType>
      **<xs:sequence>**
         <xs:element declaration1>
         <xs:element declaration2>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Sequence Element

The order of each element within the sequence is the order in which they appear in the resulting document. (Replaces the “,” operator in a DTD.)

```
<xsd:element name=”product">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="make" type="xs:string" />
         <xsd:element name=”model" type="xs:string" />
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

2.31 Element Choices

- Choose only one element from a list of alternatives.
- Syntax:

```
<xs:element name="name">
   <xs:complexType>
      <xs:choice>
         <xs:element declaration1>
         <xs:element declaration2>
      </xs:choice>
   </xs:complexType>
</xs:element>
```

Element Choices

```
<xsd:complexType name="status">
   <xsd:choice>
      <xsd:element name="work" type="xsd:string"/>
      <xsd:element name="play" type="xsd:string"/>\
   </xsd:choice>
</xsd:complexType>
```

In this case, either the work or the play element may appear as the child of any element of status, but not both.

Use this technique to define a choice of attributes as well as elements.

2.32 Express any order

- Must contain all elements, but they may occur in any order:

```
<xs:element name="name">
   <xs:complexType>
      <xs:all>
         <xs:element declaration1>
         <xs:element declaration2>
      </xs:all>
   </xs:complexType>
</xs:element>
```

2.33 Annotations

- The **<annotation>** element is used for documenting the schema, for human users and for applications:
    - <documentation> – For humans.
    - <appinfo> – For programs.
    - The content may be any well-formed XML.

Chapter 3 - The Java Architecture for XML Binding (JAXB)

Objectives

Key objectives of this chapter

- Understand why you need JAXB.
- Learn the basic architecture of JAXB.
- Learn the details XML to Java mapping.
- Learn how to customize mapping using annotations.

3.1 Introduction to JAXB

- JAXB provides an alternative to SAX and DOM for loading XML data into memory
- JAXB is an alternative for DOM for constructing XML document from data in memory.
- JAXB represents XML data in Java types such as classes and enums.
  - ◊ But, unlike SAX, you do not have to write these classes. JAXB will generate them from the XML schema.
- In summary, JAXB has the best of both the worlds:
  - ◊ Strong typing of SAX.
  - ◊ Minimal coding of DOM.
- Starting with Java SE 6, JAXB is now a core part of Java.

Introduction

XML is a language neutral data format. Applications read or write data using this format. Applications themselves represent data in memory using programming language specific constructs. For example, in Java data is constructed in memory as objects. When you read, data from an XML document is stored in one or more Java objects. When you write, data from the Java objects are saved in an XML document.

Traditionally, Java provided two mechanisms for reading data from an XML document into memory – SAX and DOM. The DOM API constructs a tree like data structure in memory. You can easily navigate the data structure and read the data for the XML elements. This data structure is generic, like java.util.HashMap. For example, if there is an XML element called <Customer>, no corresponding

Java class called Customer is created. DOM can not strictly validate the structure of the customer data.

SAX, on the other hand, does not create any data structure. It is the responsibility of the developer to define the data structure and save the XML data there. For example, the developer will need to code a Customer class. The program needs to have code to create a Customer object for each <Customer> element and save the value of the child elements and attributes in the Java object.

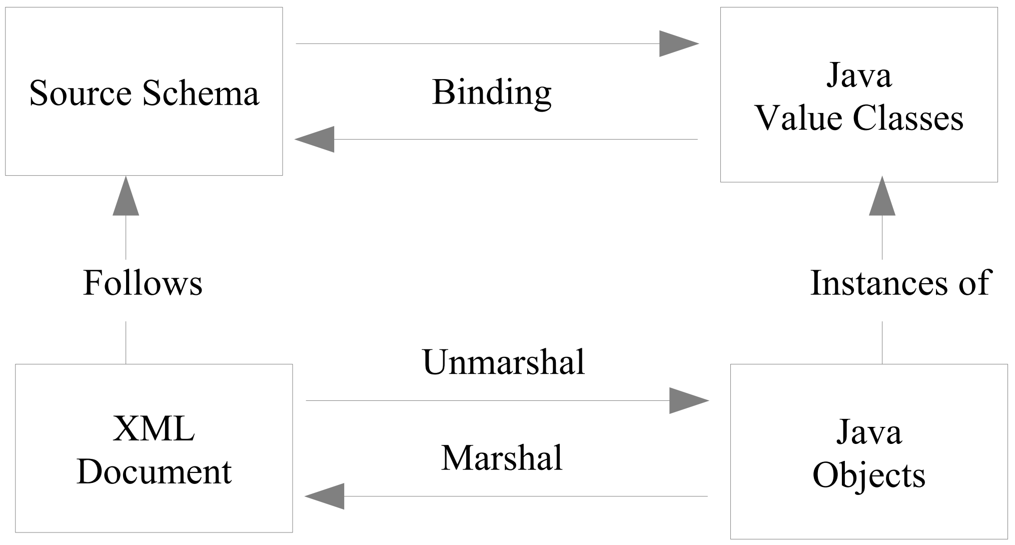
SAX provides more type safety. The resulting application code is properly object oriented. But, SAX requires extensive coding. DOM requires very little coding but has no type safety. JAXB provides a middle ground that borrows the best aspects of both SAX and DOM. JAXB stands for Java Architecture for XML Binding. It is actually JSR 222. The URL for the JSR is: http://jcp.org/en/jsr/detail?id=222.

With JAXB, you get to use a properly typed data structure. But, unlike SAX, you do not have to code the classes. JAXB can generate the classes from XML schema. For example, JAXB can generate the Customer class from the <Customer> element definition in the schema. JAXB can also automate transfer of data from an XML document into Java objects and vice versa. In summary, JAXB allows you to read and write XML data in a strongly typed object oriented manner without having to write a lot of code.

There are two ways you can add JAXB to your application. Java SE 6 comes with JAXB 2.0 as a core part of Java. There is no need for external JAR files. If you must use a later version of JAXB (say 2.1), you will need to download and use the JAXB Reference Implementation. The reference implementation is available from https://jaxb.dev.java.net/. For more information on this issue read https://jaxb.dev.java.net/guide/Migrating_JAXB_2_0_applications_to_JavaSE_6.html.

3.2 Overview of Data Binding

- Data binding is a part of the application design that deals with how XML data will be represented in memory.
- In JAXB, XML data types defined in the source schema are bound to value classes.
- At runtime the following can happen:
  - Data from an XML document is read and stored in value class objects. This is called unmarshalling.
  - Data from value class objects is saved in an XML document. This is called marshalling.



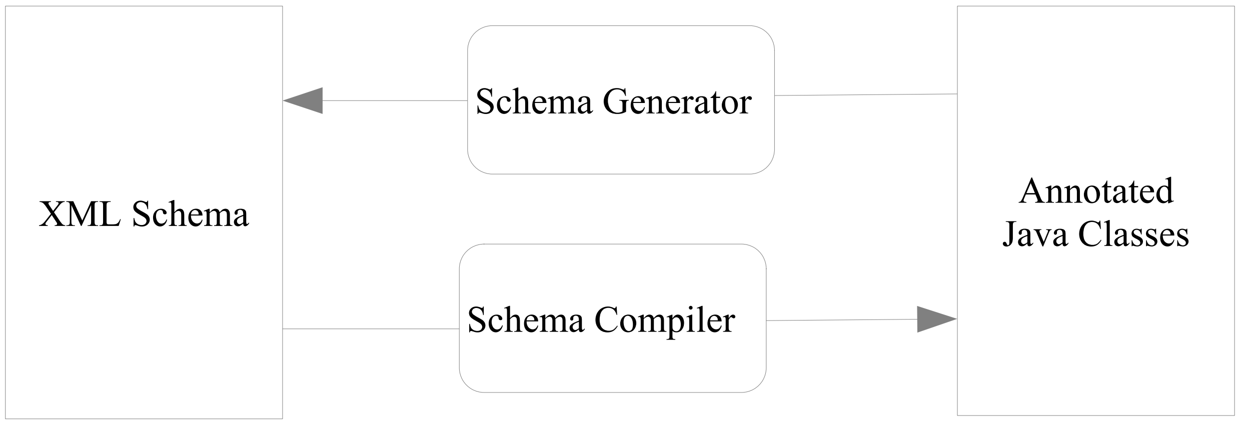
Data Binding

A Java application reads XML data into memory and produces XML documents from data in memory. Data binding is a part of the application design that deals with how XML data will be stored in memory. For example, we can bind the Customer class with the <Customer> element. In that case, data read from the <Customer> elements will be saved in Customer objects. When an XML document needs to be created, Customer objects will be saved as <Customer> elements in the document. The concept is very similar to serialization in Java. Except, in JAXB you can customize and control the binding design.

In JAXB data binding works as follows. The XML schema that describes the structure of an XML document is called the **source schema**. For example, the <Customer> element is defined in that source schema. Elements in the schema are bound to Java classes known as **value classes**. For example, Customer is a value class. Each attribute and child element of the <Customer> element will be bound to a JavaBean property in the Customer class.

This source schema to value class binding happens at coding time. At runtime, data from actual XML documents is bound to actual Java objects. The runtime binding is known as marshalling and unmarshalling. **Unmarshalling** reads data from an XML document and stores the data in value class objects. **Masrhalling** saves data from value class objects into an XML document. JAXB provides an API that you can use to marshal or unmarshal data from the application code.

3.3 JAXB Architecture



JAXB Architecture

We will now describe the key components of JAXB and how they work together. A **schema compiler** is provided that reads the XML schema and produces various Java artifacts. For example, for each complexType in the schema a value class is generated. These artifacts are annotated with **JAXB annotations** and are generally known as **portable JAXB annotated classes**. The schema compiler is used for top-down style development where the XML schema is designed first and the Java classes are generated from it. This style of development focuses on the XML schema in a programming language neutral fashion. This leads to excellent schema definition and the best interoperability between programming languages.

A **schema generator** is also provided. It can generate an XML schema file by inspecting a set of Java classes. A schema generator is used in bottom-up style development where the Java classes are developed first and the XML schema is generated from it. This style is good for Java developers who do not wish to learn how to design XML schema. The convenience comes at the expense of good XML schema design. The generated schema may not be the best possible. It may not conform to your corporation's standards. You can customize the schema generation by using the JAXB annotations.

The JAXB programming model is made up of two parts – annotations and programming API. The annotations capture the data binding. They are used at coding or design time in both top-down and bottom-up style development. For top-down, the schema compiler generates Java classes that are already annotated. For bottom-up, the developer is responsible for annotations. At runtime, marshalling and unmarshalling is done using the JAXB API known as the **binding framework**.

In summary, the basic programming steps are as follows.

1. First, you need to create the Java value classes that will hold data. There are two ways you can create them. You can hand code them and annotate them with JAXB annotations (bottom-up). Or, you can generate them from an existing schema (top-down). In that case, the generated classes will be automatically annotated.

2. Optionally, if you do not have a schema, you can generate one from the value classes. This is applicable for bottom-up programming only.
3. Develop application code that marshals and unmarshals data using the JAXB API.

3.4 Binding Example

A complex type in XML schema:

```
<xs:complexType name="AddressType">
    <xs:sequence>
        <xs:element name="city" type="xs:string" />
        <xs:element name="street" type="xs:string" />
        <xs:element name="zip" type="xs:string" />
    </xs:sequence>
</xs:complexType>
```

Is bound to an annotated Java class:

```
@XmlType(name = "AddressType", namespace =
"http://svc.webage.com/data")
public class Address {
    String city;
    String street;
    String zip;
    public String getCity() {
        return city;
    }
    public void setCity(String value) {
        this.city = value;
    }
//Other getters and setters omitted
}
```

Example

In this example we see an XML type (AddressType) and Java class (Address) bound to each other. The key to binding is the XML annotation in the Java class. Here, the @XmlType annotation captures the mapping. Without the annotations, JAXB will apply a set of default mapping rules. By default, the Address class will be mapped to the "address" XML type. With the annotation, the Address class is mapped to the AddressType XML type.

We did not apply any annotation for the getter and setter methods. As a result, JAXB will apply default rules to map the JavaBean properties to the child elements of AddressType. This way, the "city" property of the JavaBean will be mapped to the "city" child element, and so on.

The schema compiler can generate the Java class from the schema. The schema generator can generate the XML type definition from the annotated Java class. No matter what approach you take, you are always left with an XML schema and a set of annotated Java classes.

3.5 Binding Framework Example

This example shows the marshalling API.

```
Address addr = new Address();
addr.setCity("Miami");
//Other setters

JAXBContext ctx =
    JAXBContext.newInstance("com.webage.data");
Marshaller marshaller = ctx.createMarshaller();
marshaller.setProperty(
    Marshaller.JAXB_FORMATTED_OUTPUT, new Boolean(true));

marshaller.marshal(addr,
    new FileOutputStream("Output.xml"));
```

Binding Framework Example

The framework API is used to marshal and unmarshal data. In the example above we observe hos marshalling works. We create an instance of the Address class and initialize the object. Then, we create a marshaller object using the JAXBContext. The marshaller is then used to save the address data in Output.xml.

3.6 Java and XML Mapping Overview

- JAXB applies mapping rules to bind XML types to Java.
- You can customize the default rules using JAXB annotations.
- The following entities are mapped (bi-directionally):
  - XML namespace is mapped to Java package.
  - XML complexType is mapped to Java class.
  - Child elements and attributes of a complexType are mapped to JavaBean properties.

  - Java enum types are mapped to constrained XML types.
  - Type collections like java.util.List<T> are mapped to XML.

Mapping Basics

Mapping is basically the rules used to bind XML types to Java classes. XML complexType and element entities are mapped to Java classes. Child elements and attributes of a complexType are mapped to the properties of a JavaBean class. The namespace of a complexType is mapped to the package name of the Java class. Java enumeration types (enum) are mapped to constrained XML types. JAXB also maps Java collections (like, java.util.List) to collections in XML. We will now learn in more details about these mapping rules.

3.7 Namespace and Package Name Mapping

- In XML schema, namespace is used to make type names unique.
- Namespace is mapped to Java package name.
  - Example – the namespace "http://www.webage.com/order/types.xsd" is mapped to the "com.webage.order.types" package.
- Java package name is mapped to namespace.
  - Example – the package name "com.webage.order.beans" is mapped to "http://beans.order.webage.com/".
- The default mapping can be customized using the @javax.xml.bind.annotation.XmlType annotation.

```
package com.webage.order;
@XmlType(namespace = "http://svc.webage.com/data")
public class Address {...}
```

Mapping Rules

Namespace to Package Mapping

In XML schema, namespace is used to make the type names unique. A namespace is mapped to a Java package. For example, if the Address XML type is defined in "http://www.webage.com/order", the corresponding Address Java class is define in com.webage.order package.

A simplified version of the algorithm used for this mapping is as follows. We will use the example namespace "http://www.webage.com/order/types.xsd" as the namespace.

```
<xs:schema targetNamespace="http://www.webage.com/order/types.xsd">
...
</xs:schema>
```

First, the protocol scheme portion (http) is stripped out.

"www.webage.com/order/types.xsd".

Next, the file name extension, if any is removed.

"www.webage.com/order/types".

Next, the string is split up by the "/" character. The parts will be as follows:

{"www.webage.com", "order", "types"}.

Next, "www" is dropped from the host name.

{"webage.com", "order", "types"}.

The remaining host name is broken up by "." and the parts are reversed.

{"com", "webage", "order", "types"}.

The final package name is: "com.webage.order.types".

The actual algorithm has a few more clauses. But, these basic rules are sufficient in most real life cases.

Package to Namespace Mapping

A simplified version of the algorithm is as follows. We will use the example package name "com.webage.order.beans".

The parts of the package name (separated by ".") are reversed:

"beans.order.webage.com".

The http protocol scheme is added. This gives us the final namespace.

"http://beans.order.webage.com".

Customizing Mapping

The default mapping can be customized using the @XmlType annotation's namespace attribute. For example:

```
package com.webage.order;
@XmlType(namespace = "http://svc.webage.com/data")
public class Address {...}
```

Here, the package name "com.webage.order" is mapped to the namespace "http://svc.webage.com/data".

3.8 Simple Type Mapping

- Java primitive types are mapped to built-in schema types. Example:

Java	**XML**
int	xsd:int
short	xsd:short
long	xsd:long
float	xsd:float

- Java standard class are also mapped to built-in schema types.

Java	**XML**
String	xsd:string
java.util.Calendar java.util.Date	xsd:dateTime
java.lang.Object	xsd:anyType
java.math.BigDecimal	xsd:decimal
java.math.BigInteger	xsd:integer

3.9 Complex Type Mapping

- A JavaBean class is mapped to an XML <complexType> and vice versa.
- Each property of the JavaBean is mapped to a child element of the XML type.
- Example:

```
public class Customer {
    public String getFullName() {...}
    public void setFullName(String fullName) {...}
    public int getId() {...}
    public void setId(int id) {...}
}
```

Maps to:

```
<xs:complexType name="customer">
  <xs:sequence>
    <xs:element name="fullName" type="xs:string"
minOccurs="0"/>
    <xs:element name="id" type="xs:int"/>
  </xs:sequence>
</xs:complexType>
```

Complex Type Mapping

An XML complex type is mapped to a JavaBean class. By default, the name of the complex type is same as the name of the Java class with the first letter converted to lower case. Also, by default, every JavaBean property is represented as a child element of the complex type. The following example will illustrate the default mapping rule.

Let us say that we have a Java class called Customer.

```
package com.webage.order.beans;

public class Customer {
          String fullName;
          int id;

          public String getFullName() {
                    return fullName;
          }
          public void setFullName(String fullName) {
                    this.fullName = fullName;
          }
          public int getId() {
                    return id;
          }
          public void setId(int id) {
                    this.id = id;
          }
}
```

According to the JavaBean standard, the class has two properties – fullName and id. These names are derived from the getter and setter method pair (and not the member variables of the class).

The corresponding complex type will be:

```
<xs:schema version="1.0" targetNamespace="http://beans.order.webage.com/"
xmlns:tns="http://beans.order.webage.com/"
xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:complexType name="customer">
    <xs:sequence>
      <xs:element name="fullName" type="xs:string" minOccurs="0"/>
      <xs:element name="id" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

Note that a complex type called "customer" maps to the Java class Customer. For every getter/setter method pair a child element is created. The order of the child elements is defined by the order of the methods in the class. In this case, the JavaBean properties are of simple types. Notice, how the simple types are mapped to the built-in XML types.

The namespace of the type is derived from the package name using the rule that we have already discussed.

3.10 Customizing Complex Type Mapping

- The @javax.xml.bind.annotation.XmlType annotation is used to customize the name of the complex type, namespace and order of child elements.
- Example:

```
@XmlType(name = "CustomerType",
namespace="http://www.webage.com/types",
    propOrder={"id", "fullName"})
public class Customer {...}
```

Maps to:

```
<xs:complexType name="CustomerType">
  <xs:sequence>
    <xs:element name="id" type="xs:int"/>
    <xs:element name="fullName" type="xs:string"
minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

Customizations

The @XmlType annotation is used to customize complex type mapping. This annotation is applied to the JavaBean class. The key attributes of this annotation are.

Attribute	**Remarks**
name	The name of the complexType.
namespace	The namespace of the complexType.
propOrder	The order of the child elements in complexType.

We can learn the use of @XmlType through an example. First, we will customize the name and namespace of the complex type.

```
@XmlType(name = "CustomerType", namespace="http://www.webage.com/types")
public class Customer {...}
```

This will map the class to the following complex type. The customized portions are highlighted using boldface.

```
<xs:schema version="1.0" targetNamespace="http://www.webage.com/types"
xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:complexType name="CustomerType">
    <xs:sequence>
      <xs:element name="fullName" type="xs:string" minOccurs="0"/>
      <xs:element name="id" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

Next, we will change the order of the child elements using the propOrder attribute.

```
@XmlType(name = "CustomerType", namespace="http://www.webage.com/types",
    propOrder={"id", "fullName"})
public class Customer {...}
```

Now, the complex type will be as follows.

```
<xs:complexType name="CustomerType">
  <xs:sequence>
    <xs:element name="id" type="xs:int"/>
    <xs:element name="fullName" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

3.11 Property Accessor Customization

- By default all public getter/setter pair and every public field will be automatically bound to child elements.
- Alternatively, you can directly map the member fields to child elements without the need for getter/setter method. This is done using the @XmlAccessorType annotation.
- Example:

```
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
       String firstLastName;
       int id;
}
```

Customization

By default, all JavaBean properties and public member variables are mapped to child elements of the complex type. Alternatively, you can directly map the member variables of the Java class. This is done using the @XmlAccessorType annotation. Example:

```
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
          String firstLastName;
          int id;
}
```

This will be mapped to:

```
<xs:complexType name="customer">
  <xs:sequence>
    <xs:element name="firstLastName" type="xs:string" minOccurs="0"/>
    <xs:element name="id" type="xs:int"/>
  </xs:sequence>
</xs:complexType>
```

There is no need for the getter/setter methods in the Java class. Although, they are probably recommended. The member variables do not have to be public.

Possible values of the XmlAccessType are:

Value	**Remark**
PUBLIC_MEMBER	Every public getter/setter pair and every public field will be automatically bound to child elements. This is the default.
PROPERTY	Every getter/setter pair in a JAXB-bound class will be automatically bound to child elements.
FIELD	Every non static, non transient field in a JAXB-bound class will be automatically bound to child elements.

3.12 Property Mapping Customization

- The @XmlElement annotation is used to customize the property mapping.
- Apply it to the accessor of the property – either a getter/setter method or directly to the field.

- Example:

```
@XmlElement(name="CustomerName", required=true)
String fullName;
```

Maps to the child element:

```
<xs:element name="CustomerName" type="xs:string"/>
```

Customization

You can customize how the JavaBean properties will be mapped to child elements of the complex type using the @XmlElement annotation. If the accessor type is PROPERTY or PUBLIC_MEMBER, this annotation is applied to the getter or setter method (but not both). If the accessor type is FIELD, the annotation is applied to the member field.

The annotation has these key attributes.

Attribute	Remarks
name	The name of the child element mapped to the property.
namespace	The namespace of the child element.
nillable	Can the property value be null? true or false.
required	If the value is required. true or false.
type	The name of the Java class being referenced by the field. This is useful when the field's type is an interface or a base class.

Example:

```
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
          @XmlElement(name="CustomerName", required=true)
          String fullName;
          @XmlElement(name="CustomerId")
          int id;
}
```

This will map to:

```
<xs:complexType name="customer">
  <xs:sequence>
    <xs:element name="CustomerName" type="xs:string"/>
    <xs:element name="CustomerId" type="xs:int"/>
  </xs:sequence>
</xs:complexType>
```

Note how the element names have been changed. Also, for CustomerName the minoccurs is 1 which means at least one occurrence of the child element must occur. This happened because we set the required attribute to true.

3.13 XML Element Mapping

- A JavaBean can be designated as an XML element type using the @XmlRootElement annotation.
- For such a class, both <complexType> and <element> will be defined.
- You can use @XmlType in conjunction with @XmlRootElement to customize the mapping.
- Example:

```
@XmlRootElement(name="CustomerElement")
@XmlType(name="CustomerType")
public class Customer {...}
```

Maps to:

```
<xs:element name="CustomerElement" type="tns:CustomerType"/
>

<xs:complexType name="CustomerType">
  <xs:sequence>...</xs:sequence>
</xs:complexType>
```

Element Mapping

You can designate a JavaBean class as an XML element. Recall, by default, a class is only mapped to a complex type. The element designation is done using the @XmlRootElement annotation. For such a class, a complex type and an element is defined. For example:

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
          @XmlElement(name="CustomerName", required=true)
          String fullName;
          @XmlElement(name="CustomerId")
          int id;
}
```

This will map to:

```
<xs:complexType name="customer">
```

```
  <xs:sequence>
    <xs:element name="CustomerName" type="xs:string"/>
    <xs:element name="CustomerId" type="xs:int"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="customer" type="tns:customer"/>
```

You can use @XmlRootElement and @XmlType to customize the mapping. For example:

```
@XmlRootElement(name="CustomerElement", namespace="http://www.webage.com/types")
@XmlType(name="CustomerType", namespace="http://www.webage.com/types")
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
         @XmlElement(name="CustomerName", required=true)
         String fullName;
         @XmlElement(name="CustomerId")
         int id;
}
```

This will be mapped to the following schema. The customized portions are highlighted in boldface.

```
<xs:schema version="1.0" targetNamespace="http://www.webage.com/types"
xmlns:tns="http://www.webage.com/types"
xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="CustomerElement" type="tns:CustomerType"/>

<xs:complexType name="CustomerType">
  <xs:sequence>
    <xs:element name="CustomerName" type="xs:string"/>
    <xs:element name="CustomerId" type="xs:int"/>
  </xs:sequence>
</xs:complexType>

</xs:schema>
```

3.14 Mapping Java Enums

- By default, a Java enum type maps to a restricted simple type.
- Example:

```
public enum CustomerType {
       SILVER, GOLD, PLATINUM;
}
```

This is mapped to:

```
<xs:simpleType name="customerType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="SILVER"/>
    <xs:enumeration value="GOLD"/>
    <xs:enumeration value="PLATINUM"/>
  </xs:restriction>
</xs:simpleType>
```

- You can use @XmlEnum to change the type of the values from default string to any other built in type.

Java Enums

An enum is mapped to a restricted simple type. Example:

```
public enum CustomerType {
         SILVER, GOLD, PLATINUM;
}
```

This is mapped to:

```
<xs:simpleType name="customerType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="SILVER"/>
    <xs:enumeration value="GOLD"/>
    <xs:enumeration value="PLATINUM"/>
  </xs:restriction>
</xs:simpleType>
```

As you can see, the enumeration values are simply the values in the Java enum converted to string (such as "GOLD" and "SILVER").

You can use @XmlType as usual to customize the name and namespace of the simple type. In addition, you can use @XmlEnum to change the data type of the enumerated values. The following uses integer as the value.

```
@XmlType(name="TypeQualifier")
@XmlEnum(Integer.class)
public enum CustomerType {
         @XmlEnumValue("1") SILVER,
         @XmlEnumValue("2") GOLD,
         @XmlEnumValue("3") PLATINUM
}
```

The @XmlEnumValue annotation is used here to assign integer values. No such construct exists in Java. That is, the values in a Java enum do not belong to an integer or String type. The values are of the same type as the enum itself. However, in XML schema <xs:enumeration> needs a simple value. @XmlEnumValue provides that value. The mapped simple type will now look like this. Customized portions are highlighted.

```
<xs:simpleType name="TypeQualifier">
  <xs:restriction base="xs:int">
    <xs:enumeration value="1"/>
    <xs:enumeration value="2"/>
    <xs:enumeration value="3"/>
  </xs:restriction>
</xs:simpleType>
```

You can use the enum in a value class as usual.

```
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
         String fullName;
         int id;
         CustomerType memberType;
}
```

The mapped complex type will be:

```
<xs:complexType name="customer">
  <xs:sequence>
    <xs:element name="fullName" type="xs:string" minOccurs="0"/>
    <xs:element name="id" type="xs:int"/>
    <xs:element name="memberType" type="tns:TypeQualifier" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

3.15 Mapping Collections

- An array property is mapped to an element of unbounded cardinality.
- A java.util.List<T> property is mapped to an element of unbounded cardinality.
- A java.util.Map<K, V> property is mapped to an element whose structure is defined within the definition of the complex type for the Java class.
- Always use generics and specify the type held in the collection. Without that the schema will use <xs:anyType> which is hard to validate against.

Mapping Collections

Ordered Collections

In Java an ordered collection can be represented by an array or as java.util.List<T>. Both are mapped to an element of "unbounded" cardinality. For example:

```
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
         int id;
         List<String> shippingAddress;
         String[] billingAddress;
}
```

Will be mapped to:

```
<xs:complexType name="customer">
  <xs:sequence>
    <xs:element name="id" type="xs:int"/>
    <xs:element name="shippingAddress" type="xs:string" nillable="true"
minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="billingAddress" type="xs:string" nillable="true"
minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

If you do not specify the type for a List, then the property will be mapped to <xs:anyType>. Example:

```
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
         List shippingAddress;
...
}
```

Will be mapped to:

```
<xs:complexType name="customer">
  <xs:sequence>
    <xs:element name="shippingAddress" type="xs:anyType" nillable="true"
minOccurs="0" maxOccurs="unbounded"/>
...
  </xs:sequence>
</xs:complexType>
```

This is not recommended and you should always specify the type for a List.

Mapping java.util.Map

A property of type Map<K, V> is mapped to an element whose type is defined inline. That is, the type structure of the element is defined within the type for the Java value class. An example will make this clear.

```
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
         String fullName;
         int id;
         Map<Integer, String> address;
}
```

Will be mapped to:

```
<xs:complexType name="customer">
  <xs:sequence>
    <xs:element name="fullName" type="xs:string" minOccurs="0"/>
    <xs:element name="id" type="xs:int"/>
    <xs:element name="address">
      <xs:complexType>
```

```
        <xs:sequence>
          <xs:element name="entry" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="key" minOccurs="0" type="xs:int"/>
                <xs:element name="value" minOccurs="0" type="xs:string"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

The type structure of the Map is as follows. It has zero or more <entry> elements. Each <entry> element has a <key> and <value> element. An example document for the above schema will look like this:

```
<customer>
    <fullName>John Doe</fullName>
    <id>10001</id>
    <address>
        <entry><key>1</key><value>101 Flora Street</value></entry>
        <entry><key>2</key><value>328 MacLaren Street</value></entry>
    </address>
</customer>
```

3.16 Generating Java Class and Schema

- Annotated Java classes are generated from XML schema using the xjc command.

```
xjc -p com.webage.lib -d src library.xsd
```

- XML schema is generated from annotated Java classes using the schemagen command.

```
schemagen -classpath . com\webage\lib\Library.java
```

Generating Java Class and Schema

JAXB specification calls for a tool called **xjc** that can generate Java classes from XML schema and vice versa. Java SE 6 and up includes the xjc command in the bin directory of the JDK. To generate Java classes from schema, run the command as follows:

```
xjc -p PACKAGE -d DIR SCHEMA_FILE
```

Where:

- *PACKAGE* – The package name of the generated classes.
- *DIR* – The folder where the Java files and packages will be created.
- *SCHEMA_FILE* – The name of XML schema file.

Example:

```
xjc -p com.webage.lib -d src library.xsd
```

This will generate a class for each complexType and element defined in the schema. As an example, we will use a schema for library and books as follows.

```
<complexType name="AuthorType">
    <sequence>
        <element name="FirstName" type="string"></element>
        <element name="LastName" type="string"></element>
    </sequence>
</complexType>

<complexType name="BookType">
    <sequence>
        <element name="Title" type="string"></element>
        <element name="ISBN" type="string"></element>
        <element name="Author" type="tns:AuthorType" minOccurs="1"
            maxOccurs="unbounded" />
    </sequence>
</complexType>

<element name="Library">
    <complexType>
        <sequence>
            <element name="Book" type="tns:BookType" minOccurs="1"
                maxOccurs="unbounded" />
        </sequence>
    </complexType>
</element>
```

This will generate three Java files – Library.java, BookType.java and AuthorType.java.

Schema generation from Java files is done by the **schemagen** command. This command is included in the bin directory of JDK6. Run the command as follows:

```
schemagen -classpath CP JAVA_FILE
```

Where:

- *CP* – The class path. The command must be able to load all the necessary classes from this path.
- *JAVA_FILE* – The Java file name for which the schema should be created. The command will automatically generate the schema for all other classes referenced by this class. Typically, this class represents a top level element (annotated with @XmlRootElement). All referenced classes must be compiled and available in the class path.

The command creates a schema file for each namespace. The file name is schema1.xsd, schema2.xsd and so on. There is no way to change the output schema file names.

Example:

```
schemagen -classpath . com\webage\lib\Library.java
```

This will create the schema1.xsd file.

The JAXB reference implementation (RI) ships with Ant tasks that can be used in place of the commands. Following is an example that defines and uses the xjc task.

```
<taskdef name="xjc" classname="com.sun.tools.xjc.XJCTask">
  <classpath>
    <fileset dir="JAXB_LIB" includes="*.jar"/>
  </classpath>
</taskdef>

<target name="build">
  <xjc destDir="src" package="com.webage.lib" >
    <schema  dir="src" includes="Library.xsd" />
  </xjc>
</target>
```

Where, ***JAXB_LIB*** is the folder where all JAXB RI JAR files are located.

Note, this Ant task is *not* shipped with Java SE 6. As a result, it is not a portable solution. Generally speaking, avoid using the task and use the commands.

3.17 Marshalling and Unmarshalling

- Marshalling serializes a document tree in an XML document.
- Marshalling is achieved using javax.xml.bind.Marshaller.
- Unmarshalling creates a document tree from an XML document.
- Unmarshalling is done using javax.xml.bind.Unmarshaller.
- Optionally, you can validate the source document for schema conformance during unmarshalling.

Marshalling

If you need to create a new XML document, JAXB marshalling is perhaps the simplest way to go about it. DOM API also has a XML serialization facility which you can consider as an alternative.

First, you will need to instantiate the Java value classes and initialize them. The document tree is built by setting up relationship between the objects. For the library schema shown in the previous section, we can instantiate the following objects.

```
Library l = new Library();
BookType b = new BookType();
AuthorType a = new AuthorType();
```

We can initialize the objects.

```
b.setTitle("Rabbit Redux");
b.setISBN("I0394474392");

a.setFirstName("John");
a.setLastName("Updike");
```

We can build the tree by setting up relationship.

```
b.getAuthor().add(a);
l.getBook().add(b);
```

Next, we need to create a javax.xml.bind.JAXBContext object. The constructor takes the name of the package for the Java classes. A context can work with multiple packages. In that case separate the package names with ":".

```
JAXBContext ctx = JAXBContext.newInstance("com.webage.lib");
```

Using the JAXBContext object, create and configure a javax.xml.bind.Marshaller.

```
Marshaller marshaller = ctx.createMarshaller();
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
```

Finally, output the document tree in an output stream.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Library xmlns="http://www.webage.com">
    <Book>
        <Title>Rabbit Redux</Title>
        <ISBN>I0394474392</ISBN>
        <Author>
            <FirstName>John</FirstName>
            <LastName>Updike</LastName>
        </Author>
    </Book>
</Library>
```

Unmarshalling

Unmarshalling is used as an alternative to SAX and DOM for reading data from an XML document. First, we will need to create an javax.xml.bind.Unmarshaller object using the JAXBContext.

```
JAXBContext ctx = JAXBContext.newInstance("com.webage.lib");
Unmarshaller um = ctx.createUnmarshaller();
```

Next, unmarshal an input stream. The unmarshal() method of the unmarshller returns the root level object in the document. In our example, that will be a Library object.

```
Library l = (Library) um.unmarshal(new File("C:/Output.xml"));
```

From the root level object you can traverse the document tree.

```
for (BookType b : l.getBook()) {
         System.out.println(b.getTitle());
}
```

By default, the unmarshaller does not validate the source document for schema validity. You can change that by associating a javax.xml.validation.Schema object with the unmarshaller.

```
Unmarshaller um = ...;
SchemaFactory schemaFactory =
    SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");
Schema schema = schemaFactory.newSchema(new File("C:/library.xsd"));
um.setSchema(schema);

Library l = (Library) um.unmarshal(new File("C:/Output.xml"));
```

If the source document is not valid, the unmarshal() method will throw a javax.xml.bind.UnmarshalException.

3.18 Summary

- JAXB is a programming model to simplify reading and writing XML documents from Java applications.
- JAXB maps XML schema entities like complex type and element to Java classes.
- Annotations are used in the Java classes to define the mapping.
- An API is provided to marshal and unmarshal data to and from Java objects.

Chapter 4 - Introduction to JAX-WS

Objectives

Key objectives of this chapter

- What is JAX-WS?
- Why do we need JAX-WS?
- Creating a basic Web Service provider.
- Creating a basic Web Service consumer.

4.1 What is JAX-WS?

- JAX-WS stands for Java API for XML-based Web Services. It provides the programming model to develop a Web Service provider and consumer using Java.
- JAX-WS is a vendor neutral specification. A service provider or consumer developed using JAX-WS can be deployed in any Java EE platform with minimal change.
- JAX-WS requires at least Java SE 5.
- Although, JAX-WS can be used from J2EE 1.4, most applications would be running on Java EE 5 or later.

JAX-WS Specification

JAX-WS (or JSR 224) is a programming API to develop Web Services for the Java environment. The URL of the JSR is: http://jcp.org/en/jsr/detail?id=224.

JAX-WS is the next iteration of JAX-RPC. JAX-RPC (JSR 101).

Application server vendors are gradually migrating their Web Services stack from JAX-RPC to JAX-WS. In most cases, both APIs are supported. Some features may not have been ported to JAX-WS yet. In this case, one will have to continue to develop JAX-RPC based service provider or consumer. For example, in some platform, to use WS-Security, you may have to use JAX-RPC.

If you will be using JAX-WS, chances are high that you are also using Java EE 5. Although, the connection between the two are not very strong. It is conceivable that one uses JAX-WS from a J2EE 1.4 based application. However, you must use Java SE 5 to use JAX-WS.

4.2 Advantages of JAX-WS

- JAX-WS makes many improvements over the previous API JAX-RPC. They fall mostly under these categories:
  - Much simpler programming model achieved through annotations like @WebService.
  - Much better vendor neutral code for service provider and consumer.
  - More advanced programming API for asynchronous Web Service development, message attachment using MTOM and others.
- Vendor neutrality is achieved by JAX-WS in two ways:
  - Application code developed by the developers or generated by any tool are now vendor neutral. Previously with JAX-RPC, such code, especially the generated code was vendor specific.
  - XML to Java class data mapping is now completely vendor neutral and controlled by the JAXB specification (JSR 222).

Advantages

Java SE has added support for annotations. Annotations are used by the developers to specify qualifying statements about a class, member variable or method. You can think of them as metadata. Annotations are later processed by an external entity, such as a EJB container or Web Services engine to understand the different qualities of a class or its member variables and methods.

Annotations have had a beneficial effect on Java programming. It has made EJB programming simpler. Now, with JAX-WS, annotations have made Web Services development much easier than JAX-RPC. For example, to create a Web Service provider using a POJO, all you have to do is annotate the class with @WebService. We will learn about some of these basic annotations in this chapter. A separate chapter will describe them in more details.

The JAX-RPC did not quite nail down the Java to XML mapping. This lead to development of Web Service providers and consumer that only ran in a certain Java vendor's platform. Similarly, any code that was generated in the server or client side had many vendor specific constructs. It was hard to write a provider application and deploy it in multiple Java vendor's platform. With JAX-WS, these problems are largely resolved. XML to Java mapping is now fully controlled through JAXB. Any hand developed code or generated code will most likely be vendor neutral. This will really help companies who need to run their service in multiple vendor's platform.

4.3 Why Do We Need a Programming Model?

- A Web Services programming model, such as JAX-WS or Microsoft's Windows Communication Foundation (WCF) isolates the programming logic from the low level details of Web Services. For example:
  - ◊ Developers keep developing code using Java classes (or, C# in Microsoft platform) and the programming model takes care of converting data from the class objects to XML and vice versa.
  - ◊ The programming model takes care of formatting the over all SOAP document. Programmer's can control some aspects of the document using API if they wish to do so.
  - ◊ Finally, the Web Services engine deals with all network communication between the service provider and consumer.

Web Services Programming Model

One of the major advantages of having a programming model (or API) for Web Service is that the developers do not have to get involved in the gritty details of Web Services, such as, HTTP, SOAP and XML. In a way this is similar to how Servlet programming isolates the programmers from the details of the HTTP communication protocol.

With JAX-WS, a developer keeps the focus on the business logic in various Java classes. Conversion of Java object data to XML and the SOAP document are taken care of by JAX-WS. Advanced programmers can control many aspects of this XML conversion and SOAP document generation using the JAX-WS and JAXB API. Much of this API is available in the form of easy to use annotations.

4.4 Basic Java to WSDL Mapping

- JAX-WS is responsible for mapping WSDL and XML schema to Java classes. Here we discuss the basics of this mapping.
- An XML data type defined in the WSDL is mapped to a JavaBean class. This mapping is controlled by the JAXB rules.
- A <portType> in the WSDL is mapped to a Java interface known as the Service Endpoint Interface (SEI).
- Each <operation> in the <portType> maps to a method in the SEI.
- The input parameters of a Java method are mapped to an <input> message of the operation. Similarly, the return value of the Java method is

mapped to an <output> of the operation.

- Any fault thrown by an operation is mapped to a Java exception thrown by the corresponding method.

4.5 Developing a Service Provider

- A Web Service implementation is created using a Java class that has been annotated with the **@javax.jws.WebService** annotation.
- The Java class may be a POJO in a web module (WAR file). Or, it can be a stateless session bean class in an EJB module.
- Example:

```
package com.webage.quote;

import javax.jws.*;

@WebService
public class Quote {
    public double getQuote(String symbol) {
        //Retrieve quote from database
        double quotedPrice = ...;
        return quotedPrice;
    }
}
```

This is a complete implementation of a Web Service with one operation – getQuote. We didn't have to do anything with XML, SOAP or WSDL!

Service Provider

Here we discuss the very basics of creating a service provider. Later, we will learn many complex options for service creation. But, at the bottom line is, to implement a service you must have a Java class that has been annotated with @WebService. This Java class can be a regular class in a web module or a stateless session EJB class in a EJB module. Using an EJB for Web Service implementation has a few advantages. For example, the business logic runs within a distributed transaction scope. Also, if you already have an EJB, you can make it available to non-Java client applications by creating a Web Service out of it.

4.6 The Service Implementation Class

- The class must be public and must be annotated with @WebService.
- By default, all public methods will be exposed as operations of the Web Service.
- To selectively expose certain methods as operations, you can take one of two approaches:
  - Use a Service Endpoint Interface (SEI). This is discussed later.
  - Annotate the methods to be excluded using @WebMethod(exclude=true). Example:

```
@WebService
public class Quote {
    public double getQuote(String symbol) {
        //...
    }
    @WebMethod(exclude=true)
    public internalMethod() {
        //...
    }
}
```

- Only the getQuote() method will be exposed as a Web Service operation.
- The class must have a public default (no-argument) constructor.
- Input parameters, return types and exceptions thrown by all exposed methods must be compatible with the JAXB 2.0 Java to XML Schema mapping definition.

Service implementation Class

When you annotate a class with @WebService, all public methods are automatically exposed as the operations of the Web Service. As a result, there is no need to individually annotate each method with @WebMethod. This annotation is used to exclude certain public methods from being exposed. This is particularly handy when you are Web Service enabling an existing class from a legacy application. The class may have certain methods that are not fit for public consumption.

Each exposed method will correspond to a <operation> element in the <portType> of the service. By default, the name of the operation will be same as the Java method name. This may not be ideal or acceptable in some cases. You can specify the name of the operation using the "operationName" attribute of @WebMethod. For example:

```
@WebMethod(operationName="GetStockQuote")
public double getQuote(String symbol) {
    //...
}
```

This will make the name of the operation in the WSDL file to be "GetStockQuote".

4.7 The Service Endpoint Interface (SEI)

- An SEI is a Java interface that is used to specify the methods that will be exposed as Web Services operations.
- SEI directly corresponds to the <portType> in a WSDL file. It is an abstract description of what the service is capable of doing. The implementation class provides the concrete existence of the service.
- The use of SEI is totally optional as you can implement a Web service without an SEI. However, SEI leads to a cleaner description of what the Web Service does.
- To create an SEI, define a Java interface and annotate it with **@WebService**. Example:

```
package com.webage.quote;
//...
@WebService
public interface QuoteSvc {
    public double getQuote(String symbol);
}
```

4.8 The Service Endpoint Interface (SEI)

- The implementation class specifies the SEI using the **endpointInterface** attribute of the @WebService annotation.
  - You should also use the 'implements' Java keyword to implement the interface to get compilation errors if there are mis-matches

```
@WebService(endpointInterface="com.webage.quote.QuoteSvc")
public class Quote implements QuoteSvc {
    public double getQuote(String symbol) {
        //...
    }
    public internalMethod() {
            //...
        }
}
```

- Note the following about the example above:
  - Only the getQuote() method will be exposed an operation.
  - Both SEI and implementation class must be annotated with @WebService.
- All methods in the SEI will always be exposed as operations. You can not exclude them using the @WebMethod annotation.
- Input parameters and return types of all methods of the SEI must be compatible with the JAXB 2.0 Java to XML Schema mapping definition.

SEI

Technically the Java class which implements the web service does not have to specify 'implements QuoteSvc' in the example above because the 'endpointInterface' property is the only thing JAX-WS will look at. It is highly suggested to implement the Java interface though because then the Java compiler can catch when method signatures don't match up instead of getting odd runtime errors.

Although SEI is not strictly required, it is generally recommended that you use them. They tend to lead to better design. There are a few main uses of SEI. When a service is being created from ground up, the designer should define the SEI to capture the essence of the service. The SEI will describe in accurate details the contract between the service provider and the consumer. Once the SEI is defined, the developers can implement the business logic for the methods.

SEI can also help you Web Service enable an existing (legacy) Java class. Create the SEI and add to it all the methods of the class that should be exposed. Use the @WebMethod annotation in the SEI to further decorate and qualify the methods. Then implement the SEI from the class. This way, you don't have to use the @WebMethod annotation extensively in the class itself making it a clean solution.

4.9 Service Implementation Options

- You can create the service implementation class in one of two ways:
  - **Top-down:** You generate the implementation class and SEI from a WSDL file.
  - **Bottom-up:** Develop the SEI and implementation class and annotate them with @WebService, @WebMethod etc. JAX-WS will

generate the WSDL for you.

- The top-down approach (also known as design first or contract first) defines the service interface in a WSDL file. Then the SEI, implementation class and JavaBeans corresponding to the XML data types are generated. The generated SEI and implementation classes are automatically annotated with @WebService.
    - This is the recommended approach since the wire level XML data format can be precisely controlled through the XML schema definition.
    - A vendor may provide its own tool for code generation. JAX-WS reference implementation provides the wsimport command line tool.
- The bottom-up approach is useful when you already have the service implementation class. This is common when a legacy application must exposed as a Web Service.
    - The problem with this approach is that the wire level data XML format will be controlled by the default mapping rules of JAXB. You can alter the mapping but that will require extensive and time consuming use of various JAX-WS and JAXB annotations.

Options

In Service Oriented Architecture (SOA) the service contract is of paramount importance. The contract is defined in a WSDL file. It captures the service operation names and the input and output XML data format. The XML data format is defined using XML schema. The schema is very carefully defined using corporate standards. A data type may be already defined as a part of another service. In this case, the data type is re-used. The top-down approach lends itself well with this contract oriented design. First the XML schema and WSDL are defined. Then the SEI, service implementation class and the JavaBeans corresponding to the XML data types are generated. These classes will be heavily annotated with JAXB and JAX-WS annotations. Also, the generated implementation class will have skeletal method definitions. The developer will need to fill in the actual business logic in these methods.

4.10 Developing a Consumer

- JAX-WS provides the programming model to develop a Web Service consumer.
- The Web Service itself does not have to be developed using JAX-WS or Java.
  - For example, the service may have been developed using C# and .NET. You should be able to develop a client using JAX-WS.
- There are mainly two approaches to building a JAX-WS consumer:
  - **Dynamic client:** This type of a consumer program can invoke any web service. For example, a generic Web Service testing tool and performance measurement tool will fall under this category.
  - **Static client:** This type of a client can invoke operations of a specific Web Service. Developing such a client is very easy using JAX-WS.

4.11 Static Client Development

- The core components of a static client are:
  - The Service Endpoint Interface (SEI). This represents the <portType> section of the WSDL. This is same as the SEI we have discussed for service implementation.
  - The JavaBean classes that correspond to the XML data types.
  - The service class. This class represents the <service> element of the WSDL.

- These are generated from the WSDL of the Web Service.
  - Every vendor will have their own tool to generate the client side code.
- The consumer application takes these basic steps:
  1. Instantiates an object of the service class.
  2. From the service object, the consumer retrieves a **stub** or **proxy**. The stub object implements the SEI.
  3. Using the stub object, the consumer can easily call the available methods in the SEI. This translates into a SOAP operation call.
- Example:

```
QuoteService svc = new QuoteService();
QuoteSvc quotePort = svc.getQuotePort(); //Get the stub
double val = quotePort.getQuote("WAS"); //SOAP call
```

Static Client

The client uses the service class as a factory to get a hold of the stub object (also known as proxy). A service class object represents a <service> element within the WSDL file. A <service> element contains one or more <port> elements. If a service is accessible by multiple communication protocol, such as HTTP and messaging, there will be a separate <port> element for each protocol. The stub represents a <port> element within the service.

The client first creates an instance of the service class. It then gets a stub object from it. The stub object implements the SEI. That means, the client can start calling the operations exposed by the Web service by calling the corresponding method in the stub object. Each such method call will internally cause the Web Services runtime to submit a SOAP request using the communication protocol specified for the <port>.

4.12 The Service Class

- The service class is generated for you. Still, it's good to know about it in some level of detail.
- The service class extends the **javax.xml.ws.Service** class.
- The service class class is annotated using @WebServiceClient. Example:

```
@WebServiceClient(name = "QuoteService", targetNamespace =
"http://quote.webage.com/", wsdlLocation =
"http://host:8080/QuoteProject/QuoteService?wsdl")
```

```
public class QuoteService extends Service {
```

- It will have a constructor that is used to supply the location of the WSDL file and the name of the service as specified in the <service> element. Example:

```
public QuoteService(URL wsdlLocation, QName serviceName) {
    super(wsdlLocation, serviceName);
}
```

4.13 The Service Class

- A default constructor will also be available that supplies the default values of the WSDL location and service name:

```
public QuoteService() {
  super(
    new URL("http://host:8080/QuoteProject/QuoteService?
wsdl"),
    new Qname(
      "http://quote.webage.com/", "QuoteService"));
}
```

- The service class will contain a method that returns a stub object.
  - A stub represents a <port> element inside the <service>. A stub implements the SEI.
  - This method simply delegates to the Service.getPort() method.

```
@WebEndpoint(name = "QuotePort")
public QuoteSvc getQuotePort() {
    return super.getPort(
        new Qname(
            "http://quote.webage.com/", "QuotePort"),
        QuoteSvc.class);
}
```

The Service Class

The service class is a part of the code generated for the client. By default, the name of the service class is same as the value of the "name" attribute of the <service> element in the WSDL.

A client is interested in the service class only for two reasons. It should be able to instantiate an object of the service class. After that, it should be able to get a stub object from the service object.

The constructor of the service class allows us to override the location of the WSDL file. At the time of code generation, the WSDL location will most likely point to the local development machine. This will be the default WSDL location. Needless to say that this location will not work in the production machine. As a result, you must know how to override the URL of the WSDL file. Generally speaking, avoid using the default constructor.

The constructor also takes as input the name of the service as it appears in the <service> element of the WSDL. This name generally doesn't change from development to production. Notice, though, how a fully qualified name (namespace plus local name) is supplied using the Qname object.

In summary, to create a service object that overrides the WSDL URL, you will need to write code like this in the consumer application.

```
QuoteService svc = new QuoteService(
    new URL("http://production_hostname/QuoteProject/QuoteService?wsdl"),
    new Qname("http://quote.webage.com/", "QuoteService"));
```

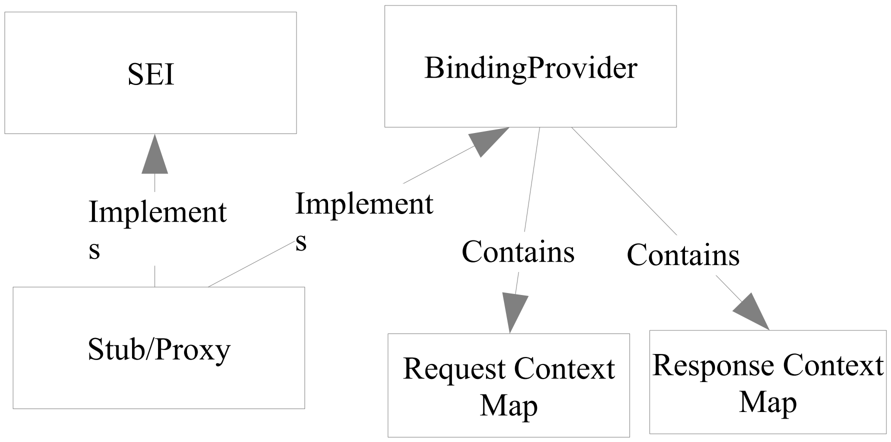
The service class will also have a get*PortName*() method that returns the stub object. Where *PortName* must be same as the "name" attribute of the <port> element within the <service> element in question. A developer need to simply know the expected name of the method and use it. The actual details of the implementation of the method is of no consequence. For the sake of completeness, you should know that this method is annotated with @WebEndpoint. The "name" attribute of the annotation points to the name of the <port>. The method implementation simply calls the Service.getPort() method.

4.14 The BindingProvider Interface

- The communication protocol specific configuration properties of a stub can be controlled through the **javax.xml.ws.BindingProvder** interface.
  - ◊ For example, if a stub represents a <port> that uses the SOAP over HTTP protocol, you will be able to set the SOAP endpoint URL using the BindingProvider interface.
- Every stub object implements the BindingProvider interface.
- The binding provider object maintains a java.util.Map to hold various SOAP request specific configurations. This is called the request context.
  - ◊ A consumer saves configuration values here prior to making a SOAP call.

4.15 The BindingProvider Interface

- Similarly, a Map is used to hold response specific configuration. This is called the response context.



The stub implements the SEI as well as the BindingProvider interface. The binding provider contains a map for the request configuration and another map for the response configuration.

4.16 The BindingProvider Interface

- The following example shows how to supply the SOAP endpoint URL:

```
QuoteService svc = new QuoteService();
QuoteSvc quotePort = svc.getQuotePort(); //Get the stub

BindingProvider bp = (BindingProvider) quotePort;
Map<String, Object> context = bp.getRequestContext();
context.put(
    BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
    "http://production_host/QuoteProject/QuoteService");

double val = quotePort.getQuote("WAS"); //SOAP call
```

The BindingProvider Interface

Binding provider plays a crucial role in the client side. It allows a consumer application to configure various properties of the SOAP communication. At minimum, you should know how to dynamically set the SOAP endpoint URL. By default, when the client side code is generated, the endpoint will most likely be set for the development environment. Typically, you will see "localhost" or "127.0.0.1" being used in the URL. This will work fine in the development machine but will invariably fail in production.

The example above shows how to override the default endpoint URL. First, we typecast the stub into a BindingProvider object. Then, we call the getRequestContext() to obtain the request context map. Within the map we store the endpoint URL property.

Useful properties you can set in the request context are as follows:

Name	**Value**
BindingProvider.ENDPOINT_ADDRESS_PROPERTY	The SOAP endpoint URL string.
BindingProvider.USERNAME_PROPERTY and BindingProvider.PASSWORD_PROPERTY	If the Web Service is protected by basic HTTP authentication, you can supply the user ID and password strings.
BindingProvider.SESSION_MAINTAIN_PROPERTY	Set to Boolean.TRUE or Boolean.FALSE. If set to true, session information, such as cookies will be preserved between multiple operation invocations.

In addition to these standard properties, your vendor may define properties to control more advanced aspects of communication. For example, if you wish to use WS-Security or WS-Addressing, your vendor may require you to configure additional vendor specific properties in the request or response context.

4.17 Summary

- JAX-WS is a major improvement over JAX-RPC as a programming model for Java based Web Service provider and consumer development.
- JAX-WS uses the JAXB standard for Java to XML data mapping.
- A service provider is a Java class annotated using @WebService.
- The service provider class can be a POJO in a web module or a stateless session EJB class in an EJB module.
- In top-down development, you generate a provider class from a WSDL file.
- In bottom-up development an existing Java class is Web Service enabled. System generates the WSDL file.
- Although optional, it is recommended that you develop the SEI to capture the contract of the service.

- Using JAX-WS, you can build a client for a service that may have been developed using any programming language.
- A client uses the service class and stub to make operation calls.
- A client uses the BindingProvider interface to configure various properties of the communication with the service provider.

Chapter 5 - Web Services Description Language (WSDL)

Objectives

Key objectives of this chapter

- Describe basic WSDL features
- Describe the various WSDL Document Tags
- Read and understand most WSDL files

5.1 WSDL Overview

- WSDL provides an XML format to describe the nature of a Web Service.
- A WSDL file has two sections.
  - The abstract part explains what the service can do for its consumers and the logical data format.
  - The concrete part describes the wire level (physical) data format and how a consumer can connect to the service provider.
- Originally proposed by IBM, Microsoft, and Ariba, WSDL is now maintained by W3C.

Overview

Web Services Description Language (WSDL) provides an XML format to describe the nature of a Web Service. In Service Oriented Architecture (SOA), almost any kind of service can be described using WSDL. As a result, WSDL, despite its name, is not restricted to Web Services only.

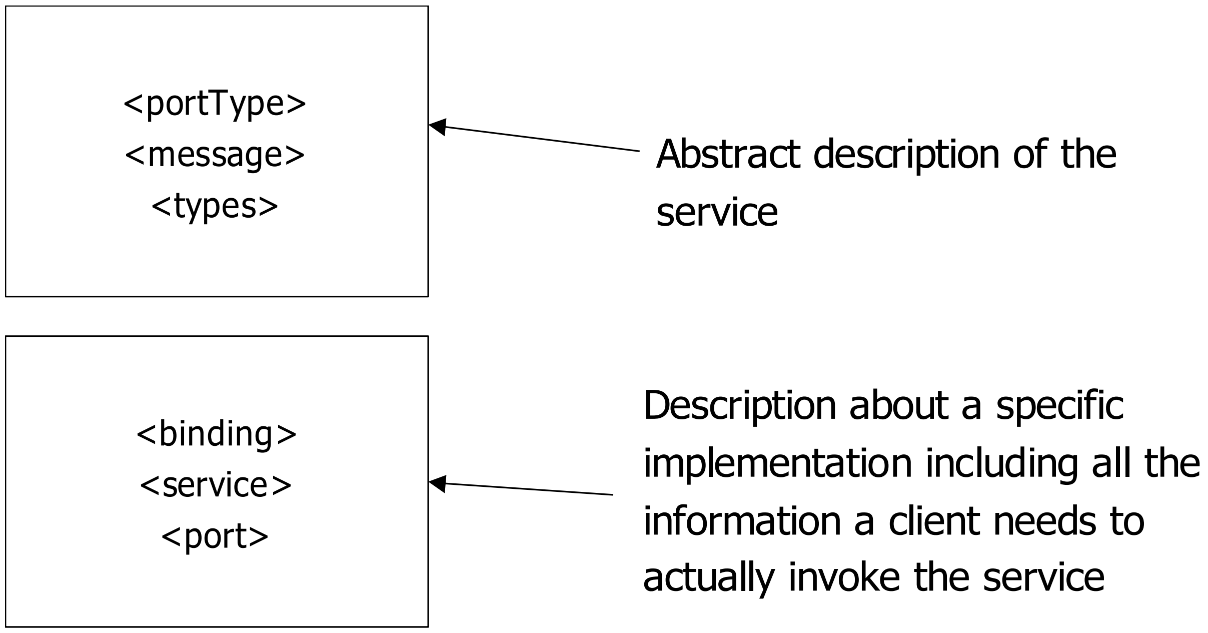
Description of a service has two parts. The **abstract** part explains what the service can do for its consumers. For example, an Accounting service can let a consumer open an account, query account balance and do fund transfer. Each task supported by a service is called an **operation**. An operation can take an input **message**, return an output message and throw a fault. The abstract part also describes the logical data structure used by these messages. For example, to open a new account, a consumer must provide detail information about the customer. After the account is opened, the service will produce the account number.

The **concrete** part describes the wire level (physical) data format used by the service. For example, it will describe exactly how the customer information must be formatted by the consumer before it is transferred over the network. The concrete part also contains the endpoint address. This address is used by the consumer to make contact with the service provider. For example, for SOAP over HTTP protocol, a HTTP URL address will constitute the endpoint.

The abstract description is a specification or blueprint for a service. A specific service provider can conform to that. There may be multiple service providers that conform to the same specification. To model this type of scenario, you need to create a WSDL containing the abstract portion only. A service provider should supply its own WSDL that imports the abstract WSDL and adds the concrete description that is specific to that provider.

WSDL was originally proposed by IBM, Microsoft, and Ariba. It is currently being maintained by W3C In this chapter, we will focus mostly on WSDL 1.1 since this is the most widely implemented standard. You can view the full specification at http://www.w3.org/TR/wsdl. The latest version of WSDL is 2.0. The specification is available at http://www.w3.org/TR/wsdl20/.

5.2 WSDL Syntax Overview


<portType>
<message>
<types>
Abstract description of the service
<binding>
<service>
<port>
Description about a specific implementation including all the information a client needs to actually invoke the service

WSDL Syntax Overview

A WSDL document is made up of various XML elements. The core elements used in a WSDL are defined in the WSDL and XML Schema specification. In addition, a WSDL may contain extension elements defined in other specifications (for example, SOAP). In this chapter, we will focus mostly on the core elements.

The namespace for the elements defined in the WSDL specification is "http://schemas.xmlsoap.org/wsdl/". The full schema for WSDL is available within the specification page. Now, we will take a quick look at a few key WSDL elements.

The elements used in the abstract portion of the WSDL are:

<portType>: A named collection of operations performed by the Web service.

<operation>: Represents an operation and is an abstract definition of an action supported by a Web Service.

<message>: The input/output messages used by Web Service.

<types>: Language neutral data types used by the Web Service.

The elements used in the concrete portion are:

<binding>: Describes the communication protocol and wire level data format used by the Web service.

<port>: Represents a single endpoint address (or access point) for the Web service.

<service>: A collection of ports. If a Web Service can be accessed by multiple communication protocols, the <service> will contain a <port> for each such protocol.

In addition these tags, the following tags need to be mentioned:

<definitions>: The root element of any WSDL file. Also used to set the namespace for all the entities defined in the WSDL (such as <portType>, <service> etc.).

<import>: Used to include an external WSDL file.

Example WSDL

We will use the following example WSDL throughout this chapter. After completing this chapter, you should be able to read and understand this WSDL.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="AccountingSvc" targetNamespace="http://beans.order.webage.com/"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:tns="http://beans.order.webage.com/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
  <types>
    <xsd:schema>
      <xs:element name="CreateAccount" type="tns:CreateAccount"/>
      <xs:element name="CreateAccountResponse" type="tns:CreateAccountResponse"/>
      <xs:element name="GetBalance" type="tns:GetBalance"/>
      <xs:element name="GetBalanceResponse" type="tns:GetBalanceResponse"/>
      <xs:complexType name="GetBalance">
        <xs:sequence>
          <xs:element name="AccountID" type="xs:int"/>
        </xs:sequence>
      </xs:complexType>

      <xs:complexType name="GetBalanceResponse">
        <xs:sequence>
          <xs:element name="Balance" type="xs:double"/>
        </xs:sequence>
      </xs:complexType>
      <xs:complexType name="CreateAccount">
        <xs:sequence>
          <xs:element name="Customer" type="tns:customer" minOccurs="0"/>
        </xs:sequence>
      </xs:complexType>

      <xs:complexType name="customer">
        <xs:sequence>
          <xs:element name="FirstName" type="xs:string" minOccurs="0"/>
          <xs:element name="LastName" type="xs:string" minOccurs="0"/>
          <xs:element name="id" type="xs:int"/>
        </xs:sequence>
      </xs:complexType>

      <xs:complexType name="CreateAccountResponse">
        <xs:sequence>
          <xs:element name="AccountID" type="xs:int"/>
        </xs:sequence>
      </xs:complexType>
    </xsd:schema>
  </types>

  <message name="GetBalance">
    <part name="parameters" element="tns:GetBalance"></part>
  </message>
  <message name="GetBalanceResponse">
    <part name="parameters" element="tns:GetBalanceResponse"></part>
  </message>
  <message name="CreateAccount">
    <part name="parameters" element="tns:CreateAccount"></part>
  </message>
  <message name="CreateAccountResponse">
    <part name="parameters" element="tns:CreateAccountResponse"></part>
  </message>

  <portType name="AccountingPortType">
    <operation name="CreateAccount">
      <input message="tns:CreateAccount"></input>
      <output message="tns:CreateAccountResponse"></output>
    </operation>
    <operation name="GetBalance">
      <input message="tns:GetBalance"></input>
      <output message="tns:GetBalanceResponse"></output>
    </operation>
  </portType>

  <binding name="AccountingPortBinding" type="tns:AccountingPortType">
    <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="CreateAccount">
      <soap:operation soapAction=""/>
      <input><soap:body use="literal"/></input>
      <output><soap:body use="literal"/></output>
    </operation>
    <operation name="GetBalance">
      <soap:operation soapAction=""/>
      <input><soap:body use="literal"/></input>
      <output><soap:body use="literal"/></output>
    </operation>
  </binding>

  <service name="AccountingSvc">
    <port name="AccountingPort" binding="tns:AccountingPortBinding">
      <soap:address location="http://localhost:9080/AccountingSvc"/>
    </port>
  </service>
</definitions>
```

5.3 <definitions>

- Root element of a WSDL document.
- Used to declare the namespace of the entities defined in the document.
- Example:

```
<definitions name="AccountingService"
    targetNamespace="http://beans.order.webage.com/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
....
</definitions>
```

<definitions>[1]

This must be the root element of a WSDL file, even if it is always meant to be imported within a parent WSDL file. The element has two attributes, both optional.

- name – A simple description of the service. It is just a lightweight description and does not have any other serious implication.
- targetNamespace – The URI is the namespace of all entities (such as <portTupe>) defined in the WSDL file.

The xmlns attribute is used here like in any other XML document to declare namespace prefix. It is a common practice to set the default namespace to "http://schemas.xmlsoap.org/wsdl/". This is done in the example above. Doing so simplifies the file since the WSDL elements like <portType> do not have to be qualified with a prefix.

1 <definitions> has been replaced by <description> in WSDL 2.0.

5.4 <import>

- A WSDL file uses this element to include an external WSDL file.
- Most common reasons for separating WSDL elements in different files:
  - Separate abstract and concrete portions of a service description.
  - Group WSDL elements with different namespaces in separate files.
- Example:

```
  <import
    namespace="http://www.xyz.com/abstract"
location="AccountingPortType.wsdl">
  </import>
```

<import>

A WSDL file uses this element to include an external WSDL file. It is quite common to separate the abstract and concrete portions of a WSDL file in two separate files. In that case, the concrete file needs to import the abstract file. Another use of <import> is to deal with difference in namespace for the WSDL entities. For example, the <portType> and <binding> can belong to two different namespaces. To model this, you will need to group entities with the same namespace in their own WSDL file.

A WSDL can have many <import> elements. But, they must appear as the first children of <definitions>, before any other child element.

The element has two attributes, both mandatory:

- namespace – The target namespace of the imported WSDL file.
- location – The relative URL of the imported WSDL file.

Example:

```
<definitions name="AccountingSvc"
  targetNamespace="http://www.xyz.com/concrete" ...>

  <import
    namespace="http://www.xyz.com/abstract" location="AccountingPortType.wsdl">
  </import>
....
</definitions>
```

In this example, the namespace of the imported WSDL is different from the main WSDL.

5.5 <types>

- Used to define the data types used by the Web service in a generic language neutral fashion.
- WSDL uses XML Schema syntax to define data types.
- Usage:

```
<definitions .... >
  <types>
    <xsd:schema>
....
    </xsd:schema>
  </types>
</definitions>
```

<types>

The <types> element is used to embed an XML schema document within the WSDL file. In WSDL schema is used to describe the logical data structure used by the input and output messages. The physical (wire level) data structure may or may not be same as this. The physical data format may not even be XML based. The logical data model is there to conceptually describe what type of information is supplied as input to the operations and what kind output is produced by the operations.

A WSDL can have up to one <types> element. A <types> element can contain any number of <xsd:schema> elements. Example:

```
<definitions targetNamespace="http://abc" ...
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <types>
    <xsd:schema>
      <xsd:import namespace="http://mno"
        schemaLocation="AccountingSchema.xsd"/>
    </xsd:schema>

    <xsd:schema>
      <xs:complexType name="Customer">
        <xs:sequence>
          <xs:element name="FirstName" type="xs:string"/>
          <xs:element name="LastName" type="xs:string"/>
          <xs:element name="ID" type="xs:int"/>
        </xs:sequence>
      </xs:complexType>
    </xsd:schema>
  </types>
...
</definitions>
```

Each <xsd:schema> element represents a XML schema document. It is quite common for the schema to be separated out in a different schema file. This allows the data types defined in the schema to be re-used from multiple WSDL files. Separation of schema files is also necessary when the data types are defined in a different namespace from the target namespaces of the WSDL. In the example above, we have two schema documents in the <types>. The first one is imported from AccountingSchema.xsd. The types in this file are defined in a different namespace from the target namespace of the WSDL. The second schema document directly defines the types within the WSDL file. The downside of this approach is that the types defined within the WSDL (such as the Customer complexType) can not be re-used by other services.

5.6 <message>

- The <message> element defines the input or output data for an operation.
- Each message contains one or more <part> tags.
  - ◊ Each part has an XML data type.
- The parts collectively define the data structure of a message.

<message>

A message represents a data supplied as input to an operation or output by the operation. A message has one or more <part> entities. Each part has an XML data type specified using the "element" or "type" attribute of the <part> element. Below is an example message.

```
<message name="GetBalance">
   <part name="parameters" element="tns:GetBalance"></part>
</message>
```

The name of the message is "GetBalance". It has only one part. The data type of the part is defined by the XML element <GetBalance>. This element must have been defined in the <types> section of then WSDL (see the complete example).

A two-way operation accepts input and returns output. For such an operation two separate messages need to be defined. A one-way operation takes input only. Only one message needs to be defined. **Note:** Normally, an operation must have an input message. An operation without an input message is considered a notification. This is an unusual situation and normally not supported by most services. We will discuss this in more details shortly.

5.7 <portType>

- It is a named collection of operations. Essentially defines a list of tasks a Web Service is capable of performing.

- Also known as the service interface.
- Each operation is declared in an <operation> tag.

<portType>[2]

A port type is a collection of operations. It forms the core of the abstract description of the service. It is also known as the service interface. A WSDL can have many port types. Although, generally speaking, you should create a separate WSDL for each port type. Each port type contains zero or more <operation> elements. Example:

```
  <portType name="AccountingPortType">
    <operation name="CreateAccount">
...
    </operation>
    <operation name="GetBalance">
...
    </operation>
  </portType>
```

The name of the port type must be unique in the namespace. Since, all port types defined in a WSDL file belong to the same namespace, the name must be unique in the WSDL file.

Tip: When studying a WSDL file, it helps to first focus in on the <portType>. This will quickly tell you the capabilities of the service.

5.8 <operation>

- An operation represents a task that a consumer can ask a service provider to perform.
- An operation can contain the <input>, <output> and <fault> elements.
- A one-way operation takes input but has no output or fault. Example:

```
<operation name="CloseAccount">
  <input message="tns:CloseAccount"></input>
</operation>
```

- A two-way operation must have an input and an output. It can have zero or more faults. Example:

```
<operation name="CreateAccount">
  <input message="tns:CreateAccount"></input>
```

2 <portType> has been replaced by <interface> in WSDL 2.0.

```
    <output message="tns:CreateAccountResponse"></output>
  </operation>
```

<operation>

An operation represents a task that a consumer can ask a service provider to perform. It is a capability of the service. An operation is analogous to a class method in a programming language. But, there is a key difference. An operation is abstract notion of a capability. It may or may not directly represent a class method. For example, a service provider may a human being. In that case the operation performed by a person and not by a class.

An operation can contain the <input>, <output> and <fault> elements. These elements specify the messages used for input, output and exception handling respectively. Example:

```
<operation name="CreateAccount">
  <input message="tns:CreateAccount"></input>
  <output message="tns:CreateAccountResponse"></output>
</operation>
```

The "message" attribute of <input>, <output> and <fault> refer to a <message> defined elsewhere in the WSDL.

An operation can be one of these four types:

- One-way.
- Two-way or request-response.
- Solicit-response
- Notification.

Out of these, one-way and two-way are most commonly used. The other two options are outside the scope of this course.

A one-way operation takes an input message but does not have any output or fault. Example:

```
<operation name="CloseAccount">
  <input message="tns:CloseAccount"></input>
</operation>
```

This is similar to methods that do not return any data. **Note:**

1. A method may neither take any input nor return any output. We still need to model it as a one-way operation. That is, the operation must have an input message.
2. A one-way operation is an ideal candidate for asynchronous invocation. Since no output is expected, a consumer can simply send the input message and not wait for the reply.

A two-way operation takes an input and returns an output or throws a fault. In other words, it must have both an input and an output. Optionally, it can have one or more faults. Example:

```
<operation name="CreateAccount">
```

```
      <input message="tns:CreateAccount"></input>
      <output message="tns:CreateAccountResponse"></output>
    </operation>
```

5.9 <binding>

- Binding describes the communication protocol that should be used to interact with the service provider.
  - SOAP over HTTP is the most common protocol.
- Binding also describes the wire level physical data format.
  - In SOAP, the style decides this format.

```
<binding name="AccountingPortBinding"
type="tns:AccountingPortType">
    <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="CreateAccount">
      <soap:operation soapAction=""/>
      <input><soap:body use="literal"/></input>
      <output><soap:body use="literal"/></output>
    </operation>
...
</binding>
```

<binding>

The <binding> and <service> element describe a specific service provider. A service provider is a living and breathing software or some other implementation of a service (such as a human being).

The "type" attribute of a binding refers to a port type indicating the interface of the service provider. At this point, a provider can conform to a single interface only.

The communication protocol that should be used to interact with a service provider is described in the binding. The most common protocol these days is SOAP over HTTP. The existence of an extension element called <binding> within the WSDL's <binding> element describes this protocol. For example:

```
<binding name="AccountingPortBinding" type="tns:AccountingPortType">
    <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
...
</binding>
```

Here, the <soap:binding> extension element describes that SOAP over HTTP protocol should be used. **Note:** This scheme for specifying the protocol is not stipulated by WSDL. It has become a convention that a protocol specific extension element called <binding> should appear as the first child of the WSDL <binding> element to uniquely specify the protocol.

The next responsibility of binding is to specify the wire level format of the messages. SOAP uses XML but has many dialects called styles. The style will dictate how the wire level physical data format will look like. Document literal and RPC literal are the two most common styles. Out of which, document literal, or to be precise, document literal wrapped style is highly recommended. The SOAP chapter will describe each style in details. In our example, we use the document literal style as shown in bold face below.

```
    <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>

    <operation name="CreateAccount">
      <soap:operation soapAction=""/>
      <input><soap:body use="literal"/></input>
      <output><soap:body use="literal"/></output>
    </operation>
```

A service may be accessible using multiple communication protocols. In that case, a separate binding section must appear for each protocol.

5.10 <service>

- <service> is there to describe the endpoint address for a service provider.
- For each communication protocol, <service> contains a <port>.
  - The endpoint is specified inside the <port> using protocol specific extension elements.

```
  <service name="AccountingSvc">
    <port name="AccountingPort"
binding="tns:AccountingPortBinding">
      <soap:address
location="http://localhost:9080/AccountingSvc"/>
    </port>
  </service>
```

<service>

The service section is used to specify the endpoint address of a service provider. Endpoint address is very much network protocol specific. For example, for SOAP over HTTP, a URL describes the endpoint. For SOAP over messaging, the name of the messaging queue and queue manager will constitute the address. We have already said that a service provider may be accessible by multiple protocols. For each protocol a binding is defined. Similarly, for each protocol a separate endpoint is specified using a <port> element within <service>. Example:

```
<service name="AccountingSvc">
  <port name="AccountingPort" binding="tns:AccountingPortBinding">
    <soap:address location="http://localhost:9080/AccountingSvc"/>
  </port>
</service>
```

The "binding" attribute of a <port> refers to a <binding>. Within a <port> protocol specific extension elements are used to set the endpoint information. For example, in SOAP, <soap:address> is used to set the URL.

Reading and Understanding a WSDL

WSDL can be intimidating at first. After following the tips mentioned here, you should be able to get a good understanding about a service described in a WSDL. Use the example given earlier in the chapter to practice reading a WSDL.

To get a quick understanding of a WSDL, go straight to the <portType>. The list of <operation> elements will tell you what the service can do.

To get a more detailed understanding about a specific operation, find out the logical data model for the input and output messages required by the operation. To do this, note the name of the message in the <input> or <output> element. Then lookup the <message> element by that name. The <part> element(s) within the <message> will tell you the XML data type of the message. You can delve into the XML schema and learn more about the data type.

To find out about the communication protocol supported by the service, check the <binding> elements. Finally, to get the endpoint address, check the <port> elements.

5.11 Summary

- WSDL stands for Web services Description Language.
- WSDL is an XML-based language used to describe Web services and how to locate them.
- A WSDL file has two sections:
  - An abstract description of operations and their input/output data format

    - ◊ Concrete implementations of the service

- The abstract description contains programming language and invocation protocol neutral description of operation names and message format.
- Information about a concrete implementation includes enough description for a consumer to actually invoke the service.

Chapter 6 - Simple Object Access Protocol (SOAP)

Objectives

Key objectives of this chapter

- Understand the basic concepts of SOAP.
- Understand various SOAP components, such as:
  - ◊ Envelope
  - ◊ Header
  - ◊ Body
- Describe SOAP communication styles using:
  - ◊ RPC literal method
  - ◊ Document literal method

6.1 SOAP Overview

- Simple Object Access Protocol (SOAP) is an XML-based data format for exchanging information.
- SOAP is a great data format for interoperability.
- Information is exchanged in a request and response pattern.
  - ◊ Data for the requests and responses are composed as XML document.
- SOAP can be used with any communication protocol, such as HTTP and SMTP.

SOAP Overview

SOAP is an XML-based data format for exchanging information. SOAP is not tied to any programming language, operating system, or specific technology. SOAP inherits this interoperability attribute from XML. This makes SOAP a versatile data format for communication between applications written in diverse programming languages and operating systems.

Information is exchanged in a request and response pattern. SOAP defines the request and response data format. A service consumer initiates interaction by sending a request document to the service provider. The provider processes the request and sends a reply document back.

Originally invented by Microsoft but currently controlled by W3C. Web site: http://www.w3.org/TR/soap/. The latest version of SOAP is 1.2 but the most widely used version is

1.1. Both specifications can be viewed from the URL given here. This chapter will focus on SOAP 1.1.

SOAP can be used with multiple communication protocols, such as HTTP, SMTP, FTP, JMS, etc. A service provider that supports SOAP over HTTP is traditionally known as a Web service. HTTP is the only protocol directly covered by the SOAP specification.

SOAP in Protocol Stack

SOAP covers the application level data format. The format is XML based. Requests and responses are actually XML documents. These documents are transferred from one application to another using a communication protocol. HTTP is the most prevalent protocol. SOAP does not exclude the possibility of using other protocols. For example, SOAP messages can be placed in a messaging queue for ensured delivery. They can be exported into the file system as a simple file. Or they can be posted in an FTP site.

The communication protocols themselves sit on top low level TCP/IP or UDP protocols.

6.2 SOAP Document Components

- A SOAP request or response document is made up of elements defined in the SOAP schema as well as application defined elements.

SOAP Document Components

A request or response XML document is made up of elements defined by the SOAP specification as well as application defined elements. In this chapter we will learn about the SOAP elements in details.

The SOAP elements are defined in "http://schemas.xmlsoap.org/soap/envelope/" namespace. The root element of a SOAP document is <Envelope>. The Envelope contains zero or one <Header> element. This is meant to hold infrastructural information such as security, encryption and transaction. The Header element can contain arbitrary child elements. This allows future extensions to be added to SOAP.

<Envelope> must contain one <Body> element. This element contains application defined XML elements. These elements contain business data known as **payload**.

You can download the schema for SOAP from "http://schemas.xmlsoap.org/soap/envelope/".

6.3 Example SOAP Request Document

```
POST /TestWebBottomUp/AccountingSvc HTTP/1.1
Host: localhost:9080
Content-Type: text/xml; charset=utf-8
Content-Length: 354
SOAPAction: ""

<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:q0="http://beans.order.webage.com/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <q0:GetBalance>
      <AccountID>10001</AccountID>
    </q0:GetBalance>
  </soapenv:Body>
</soapenv:Envelope>
```

Example

Here, we see an actual wire level intercept of a request sent by a service consumer. In this case, the consumer is interested in finding out the account balance of a bank account.

The document was sent by performing an HTTP POST. Consequently, the intercept starts with the HTTP headers. These are protocol specific headers and are not of concern to SOAP. Different protocols will use different meta information.

What follows the HTTP headers is the actual SOAP request document. As you can see, the root element is <soapenv:Envelope>. Here, soapenv is a prefix for the "http://schemas.xmlsoap.org/soap/envelope/" namespace. In this example, there is no <Header> element. You will see the header element for more advanced types of communications. For example, when the document is encrypted. Here, the Envelope directly contains a <Body> element. The <Body> element contains the application defined elements. These elements form the payload data. In this case, <q0:GetBalance> is the root of the payload data. Where, q0 is a prefix for the application defined namespace "http://beans.order.webage.com/".

Note: The payload root element usually represents an operation. Essentially, the consumer is asking the provider to perform that operation. The root element contains any information that the provider needs to carry out that operation. In this example, the <AccountID> is such input information required by the GetBalance operation.

6.4 Example SOAP Response Document

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Language: en-US
Content-Length: 312

<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <ns2:GetBalanceResponse
xmlns:ns2="http://beans.order.webage.com/">
        <Balance>500.0</Balance>
    </ns2:GetBalanceResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Example

The response starts with the HTTP reply header. Once again, this is not a part of the SOAP specification. The actual reply document starts with the <Envelope> element. There is no <Header> element in this example. The <Body> element contains the payload data. The root element for the payload is <ns2:GetBalanceResponse>. Where, ns2 is the prefix for the "http://beans.order.webage.com/" namespace.

Note: Usually, the name of the payload root element is made up of the operation name in the request with "Response" added at the end.

6.5 The <Envelope> Element

- Envelope is the root element of a SOAP document.
- It is a mandatory component.
- May have arbitrary attributes which can be used for extensibility purposes.

The <Envelope> Element

<Envelope> must be the root element of a SOAP document. This element is used to declare namespace prefixes. The element itself must be defined for the "http://schemas.xmlsoap.org/soap/envelope/" namespace. For SOAP 1.2, the namespace will be "http://www.w3.org/2003/05/soap-envelope". A recipient of a SOAP document uses the namespace of <Envelope> to tell the version of SOAP in use. SOAP provides no other mechanism to determine the version of SOAP used in a document.

SOAP schema doesn't define any attribute for the <Envelope> element. However, it leaves the door open for a SOAP document to specify any arbitrary attribute. For example:

```
<Envelope xmlns:me="http://my_namespace" me:myAttribute="value" ...>
...
</Envelope>
```

6.6 The <Header> Element

- Contains infrastructural information such as for transaction and security.
- If present, must be the first child element in the Envelope.
- Can contain any arbitrary extensibility elements.
- Extensibility elements can have two SOAP defined attributes:
  - actor
  - mustUnderstand

```
<soapenv:Header>
    <w:TransactionNumber
xmlns:w="http://www.webage.com/transaction/"
        soapenv:actor="http://www.webage.com/"
soapenv:mustUnderstand="1">
XXYZP1234TYR
    </w:TransactionNumber>
</soapenv:Header>
```

The <Header> Element

The header element contains non-business oriented information, such as authentication data and transaction context identifier. Normally, application developers do not need to know much about the header. Web Services consumer and provider runtime automatically add data in the header as needed.

If present, the <Header> element must be the first child element in the Envelope element. It can have any arbitrary attribute. The Header element can have any number of valid XML elements of any kind as children. These can be extensibility elements that contain special processing instructions for the receiver of the message. Example extensibility element:

```
<soapenv:Header>
   <t:MyElement
      xmlns:t="my-URI">Some value</t:MyElement>
</soapenv:Header>
```

A SOAP message may hop through several applications before reaching the final destination. Not all extensibility elements are meant for all of these recipient applications. To avoid any misunderstanding, SOAP provides a mechanism to specify the target for an extensibility element. SOAP specifies two special attributes – actor and mustUnderstand – that can be used with any extensibility element. Example:

```
<soapenv:Header>
    <w:TransactionNumber xmlns:w="http://www.webage.com/transaction/"
        soapenv:actor="http://www.webage.com/" soapenv:mustUnderstand="1">
XXYZP1234TYR
    </w:TransactionNumber>
</soapenv:Header>
```

The actor attribute contains a URI that uniquely identifies a recipient application that the extensibility element is meant for. If mustUnderstand is 1, the recipient must make sense of that header element. Otherwise, an error should be raised.

An actor application should remove the extensibility element before forwarding the SOAP document to the next recipient.

6.7 The <Body> Element

- Must define an element called <Body> within the <Envelope>.
- <Body> follows the <Header> element, if present.
- The body contains the payload request or response data.
- In a SOAP response document, <Body> may contain a <Fault>.

The <Body> Element

The body includes the request or response data. The exact format of this data depends on the communication style–-RPC or Document—and will be discussed later. **Note:** To be WS-I compliant, the <Body> element must contain only one child element as shown above.

The Body must be an immediate child element of a SOAP Envelope element. It must directly follow the SOAP Header element if present. Otherwise, it must be the first immediate child element of the SOAP Envelope element.

The element may contain a set of body entries, each being an immediate child element of the SOAP Body element. The body entries are usually business payload elements. In a SOAP response document, the <Body> may contain a <Fault> element. This happens when the service provider fails to process the request and wishes to indicate an error condition.

A <Fault> element can include four child elements that contain information about the error:

- faultcode – An error code.
- faultstring – A description of the error.
- faultactor – Optional. Uniquely identifies the component or application where the error took place.
- detail – Optional. Can contain application defined children elements for more details about the error.

Example:

```
<SOAP-ENV:Body>
    <SOAP-ENV:Fault>
        <faultcode>ERROR_INVENTORY</faultcode>
        <faultstring>The product is currently not available</faultstring>
    </SOAP-ENV:Fault>
</SOAP-ENV:Body>
```

6.8 SOAP Communication Style

- Style determines how the payload data is formatted.
- SOAP provides multiple style options:
  - RPC/encoded
  - RPC/literal
  - Document/encoded – Not used anymore.
  - Document/literal
  - Document/literal wrapped
- The style is set in the <binding> section of a WSDL.

Communication Style

SOAP provides a very general mechanism for how payload data should be included in the <Body>. WSDL, on the other hand, has a very specific notion for how a service should behave. A service has a set of operations. Each operation works with input and output messages and throws faults. A request from the consumer must identify what operation should be performed by the server and what input data should be used. The response should contain the output message or fault. SOAP communication style deals with how the operation name, input and output messages should be encoded in a SOAP document. Possible style options are:

- RPC/encoded
- RPC/literal
- Document/encoded – Not used anymore. Will not be discussed here.
- Document/literal
- Document/literal wrapped. A variation of document/literal.

Your choice of style is specified in the <binding> section of WSDL. Different style will format the payload data differently within the <Body>. It is essential that the consumer and the provider follow the same style. As long as the WSDL file in the server and client is the same, there should not be a problem.

Each style has its own advantages and disadvantages. We will now discuss them in details.

6.9 Communication Style Example

- Consider an example operation with the following basic specification:

```
double getPriceQuote(String productId, double amountWanted)
```

- The WSDL will be:

```
<message name="quoteRequest">
    <part name="productId" type="xsd:string"/>
    <part name="amountWanted" type="xsd:double"/>
</message>
<message name="quoteResponse">
    <part name="price" type="xsd:double"/>
</message>

<portType name="QuoteService">
    <operation name="getPriceQuote">
        <input message="tns:quoteRequest"/>
        <output message="tns:quoteResponse"/>
    </operation>
</portType>
```

6.10 Setting the Style in WSDL

```
<wsdl:binding name="QuoteServiceSoapBinding"
type="impl:QuoteService">
   <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
   <wsdl:operation name="getPriceQuote">
      <soap:operation soapAction=""/>
      <wsdl:input>
         <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
         <soap:body use="literal"/>
      </wsdl:output>
   </wsdl:operation>
</wsdl:binding>
```

Setting the Style

SOAP communication style is specified in the <binding> section of the WSDL. For the <soap:binding> element, the "style" attribute is set to "rpc" or "document". For each input and output message for an operation, in the <soap:body> element, set the "use" attribute to either "literal" or "encoded".

In the example above, we set the document/literal style.

6.11 RPC/Encoded Style

- The payload root element has the same name as the operation.

```
<soap:Body>
  <getPriceQuote>
    <productId xsi:type="xsd:string">BRUSH001</productId>
    <amountWanted xsi:type="xsd:double">10</amountWanted>
  </getPriceQuote>
</soap:Body>
```

- The problem is, the WSDL defines no element by that name.

RPC/Encoded Style

If this style is used, the SOAP Body will contain a child element with a name which is same as the operation name. Input data will be in child elements of the operation element.

Advantages:

1. The operation name appears in the body. This makes it easy for the server to invoke the appropriate method that implements the operation.
2. The body contains only one child element. This is WS-I compliant. However, WS-I does not recognize RPC/encoded as a valid encoding scheme.

Disadvantages:

1. The element containing the operation name (<getPriceQuote>) is not valid to any type defined in the WSDL file. This makes it difficult for the server to validate the body.
2. The type information (such as xsi:type="xsd:string") is usually just overhead and increases the size of the body.

Generally speaking, avoid this style if you can.

6.12 RPC/Literal Style

- This is similar but without type information
  - There is no type information in the request or in a WSDL file

```
<soap:Body>
  <getPriceQuote>
    <productId>BRUSH001</productId>
    <amountWanted>10</amountWanted>
  </getPriceQuote>
</soap:Body>
```

RPC/Literal Style

This style yields almost the same SOAP document as RPC/encoded, except the type information does not appear for the operation parameters.

Advantages:

1. The operation name appears in the body. This makes it easy for the server to invoke the appropriate method that implements the operation.
2. The encoding scheme is recognized by WS-I compliance.
3. Since the type information is omitted, the size of the body is compact.

Disadvantages:

1. The element containing the operation name (<getPriceQuote>) is not valid to any type defined in the WSDL file. This makes it difficult for the server to validate the body.

RPC/literal is the second most popular style after document literal (wrapped). It is the only style, other than document literal that is approved by WS-I.

6.13 Document/Literal Style

- Document/literal adds schema definitions for elements in the message
  - There are other changes to the WSDL to support this but the addition of schema is most important

```
<types>
    <schema>
        <element name="productId" type="xsd:string"/>
        <element name="amountWanted" type="xsd:double"/>
        <element name="price" type="xsd:double"/>
    </schema>
</types>
```

- Now the type of data in the message is declared
  - We still don't have the name of the operation which was a strength of RPC

```
<soap:Body>
  <productId>BRUSH001</productId>
  <amountWanted>10</amountWanted>
</soap:Body>
```

Document/Literal Style

The document literal style attempts to be fully schema compliant. The problem with the RPC styles was that the the root element for the payload was never defined in the XML schema. With document/literal, all the elements used in the payload are defined in the XML schema. This makes the SOAP document fully schema compliant.

In our on going example, we will have to make some changes to the WSDL. First, we will define a bunch of new elements.

```
<types>
    <schema>
        <element name="productId" type="xsd:string"/>
        <element name="amountWanted" type="xsd:double"/>
        <element name="price" type="xsd:double"/>
    </schema>
</types>
```

Then, we define the input and output messages.

```
<message name="quoteRequest">
  <part name="input1" element="tns:productId"/>
  <part name="input2" element="tns:amountWanted"/>
</message>
<message name="quoteResponse">
  <part name="out" element="tns:price"/>
</message>
```

Finally, we define the portType.

```
<portType name="QuoteService">
  <operation name="getPriceQuote">
    <input message="tns:quoteRequest"/>
    <output message="tns:quoteResponse"/>
  </operation>
</portType>
```

Now, when document/literal style is set in the binding section, the SOAP document will look like this:

```
<soap:Body>
  <productId>BRUSH001</productId>
  <amountWanted>10</amountWanted>
</soap:Body>
```

Advantages:

1. Each input element can now be validated against the types defined in the WSDL file.
2. WS-I recognizes document/literal. However, it also stipulates that the <Body> should have only one child element. As shown in our example, document/literal can easily lead to multiple child elements for <Body> making a message non-compliant.

Disadvantages:

1. The operation name has gone missing. Without it, it is difficult if not impossible for the server to invoke the corresponding method that implements the operation.

Problems with document/literal are solved by adopting a specific variation of the style. This is called document/literal wrapped. We will discuss that next.

6.14 Document/Literal Wrapped Style

- Declares an extra element in the schema that "wraps" all other elements in the message

```
<element name="getPriceQuote">
    <complexType>
        <sequence>
            <element name="productId" type="xsd:string"/>
            <element name="amountWanted"
                    type="xsd:double"/>
        </sequence>
    </complexType>
</element>
```

- Now all messages will have only one child part which is the "wrapper"

```
<soap:Body>
    <getPriceQuote>
        <productId>BRUSH001</productId>
        <amountWanted>10</amountWanted>
   </getPriceQuote>
</soap:Body>
```

- Most standard compliant and popular style.
- The default style for almost all major vendors.

Document/Literal Wrapped Style

Document/literal wrapped is not really a new style. It is a specific variation of document/literal. The style starts out by defining a wrapper element for the request and the response messages respectively. The <Body> contains this wrapper element as the only child. All input data goes inside the request wrapper element. Similarly, all response data goes inside the response wrapper element.

The name of the request wrapper element must be same as the operation name. In our case, this element will be defined as follows.

```
<element name="getPriceQuote">
    <complexType>
        <sequence>
             <element name="productId" type="xsd:string"/>
             <element name="amountWanted" type="xsd:double"/>
        </sequence>
    </complexType>
</element>
```

Note, how all the input data, namely, productId and amountWanted, are added as children of the request wrapper.

The response wrapper, by convention, has the same name as the operation with "Response" added at the end. It too must include all the output data as child elements.

```
<element name="getPriceQuoteResponse">
    <complexType>
        <sequence>
            <element name="price" type="xsd:double"/>
        </sequence>
    </complexType>
</element>
```

Now, a message need have only one part. The request message will have a part which is the request wrapper element.

```
<message name="quoteRequest">
         <part name="input" element="tns:getPriceQuote"/>
</message>
```

The response message will have a part which is the response wrapper element.

```
<message name="quoteResponse">
         <part name="out" element="tns:getPriceQuoteResponse"/>
</message>
```

The portType will remain unchanged.

```
<portType name="QuoteService">
  <operation name="getPriceQuote">
    <input message="tns:quoteRequest"/>
    <output message="tns:quoteResponse"/>
  </operation>
</portType>
```

Now, when document/literal style is set in the binding, the following SOAP request will be created.

```
<soap:Body>
    <getPriceQuote>
        <productId>BRUSH001</productId>
        <amountWanted>10</amountWanted>
   </getPriceQuote>
</soap:Body>
```

On the surface, this looks exactly same as the message produced by RPC/literal. The key difference is that we have a valid element defined called <getPriceQuote>. The advantages of document/literal wrapped are:

1. Existence of the operation name in the request makes it easy for the server to dispatch the request to the corresponding method that implements the operation.
2. The element within the body (<getPriceQuote>) is now valid to a data type defined in the WSDL file. The server can validate the request.
3. This will always be WS-I compliant as there is only one child element to the body.

Document/literal wrapped style is the ad-hoc standard today. All major vendors create a Web Service that by default uses this style.

6.15 Summary

- SOAP stands for Simple Object Access Protocol.
- SOAP is for communication between applications.
- SOAP is a format for sending messages.
- SOAP is designed to communicate via the Internet.
- SOAP is platform independent & language independent.
- SOAP is based on XML.
- SOAP is simple and extensible.
- SOAP allows you to navigate through firewalls.
- SOAP is being developed as a W3C standard.

Chapter 7 - Web Services Interoperability (WS-I)

Objectives

Key objectives of this chapter

- Understand the problems with web services interoperability
- What is WS-I?
- WS-I profiles.
- .NET to Java interoperability guidelines.

7.1 Introduction

- Developing web service provider and consumer requires following many separate standards:
  - XML schema, SOAP, WSDL, security etc.
- These standards have evolved separately.
  - Each attempts to be as flexible as it can be.
- When you combine them all together, the permutation of possible but valid cases are enormous.
  - Vendors began to write tools and servers that behave differently but each rightly claimed to be following specifications.
- Additionally, some of the specifications leave areas not properly explained.
- An organization called Web Services Interoperability was formed to resolve these problems.
  - www.ws-i.org
- The mandate was to come up with clear cut guidelines and sub-sets of specifications that each vendor must implement at minimum.

Introduction

Consider an operation of the form String placeOrder(Order o, User u). There are many different ways to model the input.

1. We can have a single element called placeOrderInput that contains Order and User child

elements.

2. We can have two global elements – Order and User.

Throw into that mix the RPC and document styles.

There are many similar instances where the same problem can be solved in many, each valid, ways.

WS-I is a meta specification that attempts to clarify practical problems that one would face when implementing a solution that must inter-operate between vendors.

7.2 Goal

- Web services gained popularity due to its capacity and promise of interoperability between vendors and programming languages.
  - Yet, very soon that promise was very seriously threatened.
- WS-I attempts to resolve commonly occurring interoperability problems in practical use of web services.
  - It strikes compromises, suggests a small set of options where many are possible and clarifies where the specifications are confusing.
- Also has these goals:
  - Provide best practices advise.
  - Provide WS-I compliance validation tools.
  - Promote consistent vendor implementation.
- As a best practice, use a development and runtime that is WS-I compliant.

Goal

What should vendors look out for?

Understand WS-I. Make sure that implementations satisfy the WS-I requirements. Embed validation tools wherever possible so that the end clients can follow WS-I as well.

What should end users and developers look out for?

Choose to use a WS-I compliant development tool and runtime. This will ensure easier interoperability. Understand key aspects of WS-I so that your code does not deviate from WS-I.

This is meant for the end users and discusses topics that will affect them.

7.3 What Comes out of WS-I?

- Profiles. Each profile is a collection of specifications at fixed version level.
    - A profile also restricts possible valid scenarios to a handful.
    - As long as the vendors satisfy these minimum requirements, interoperability will be ensured.
- Sample applications.
- Test and validation tools.
- Go to: http://www.ws-i.org/deliverables/ to download these.
- WS-I is very much practical problems oriented and constantly gathering information on problems people encounter in the field.
    - The Requirements Gathering Working Group is dedicated to this task. They influence future WS-I initiatives.
- In this chapter we discuss only the recommendations of WS-I that apply to you as an end developer.

What Comes out of WS-I?

A profile specifies what version level of a particular specification must be followed. This helps a lot. As the potential combinations of many versions of different specifications can cause nightmares in interoperability.

7.4 WS-I Tools

- WS-I provides conformance tools. When creating web services, the developer should take the time to run these conformance tests to ensure WS-I adherence
- Many development environments (e.g. IBM's RAD) have WS-I compliance tests built-in
    - Validate web service code right at development time
- WS-I compliance testing is especially important when going across platforms (e.g. .net to J2EE)

7.5 Profiles

- Current "final" version is **Basic Profile 1.2** (Working Group Approval Draft - October 2007).
  - Specifies the version level of SOAP, WSDL etc that must be followed.
  - Resolves over 200 common interoperability conflicts.
  - Basic Profile Version 2.0 is currently in "working draft"
- Split into several categories
  - Messaging
  - Service Description
  - Service Publication and Discovery
  - Security
- This chapter will discuss **Basic Profile 1.2**

7.6 WS-I Messaging

- WS-I incorporates the following messaging specifications:
  - SOAP 1.1
  - HTTP 1.1
  - WS-Addressing
    - Core 1.0, SOAP Binding 1.0, Metadata 1.0
  - SOAP 1.1 Request Optional Response HTTP Binding
  - SOAP Message Transmission Optimization
  - XML-Binary Optimized Packaging
  - SOAP 1.1 Binding for MTOM 1.0

7.7 Messaging Highlights

- Some messaging requirements specified by WS-I

- ◊ An envelope must be serialized as XML 1.0
- ◊ The children of the soap:Body element in an envelope must be namespace qualified
- ◊ An envelope must not contain a DTD declaration
- ◊ When receiving an envelope, a receiver must process headers of a message before any actual body processing
- ◊ A message should be sent using HTTP 1.1, but 1.0 is acceptable
- ◊ SOAP sent over HTTP must use POST (not GET)

7.8 Messaging Highlights

- The value of the **SOAPAction** HTTP header field in an HTTP request message must be a quoted string
  - ◊ SOAPAction: "myAction"
- A service should return a 2xx HTTP status code for a response message that has a successful HTTP outcome
  - ◊ 200 OK should be returned for a response message that has an envelope and is not a fault
- A service should return a 4xx status code in the event of a malformed request
- "500 Internal Server Error" must be returned if the response envelope is a fault

7.9 Service Description

- WS-I dictates use of WSDL as the means of service descriptions
- Makes use of the following specifications:
  - ◊ XML 1.0
  - ◊ Namespace in XML 1.0
  - ◊ XML Schema
  - ◊ WSDL v1.1

7.10 Service Description Highlights

- A service's WSDL 1.1 description, its UDDI binding template, or both must be available upon authorized consumer request
- Every WSDL description must be valid according to the schema [posted at the WS-I site]
- Only the **import** statement can be used to import another WSDL description
- A description must specify a non-empty attribute called **location** on the **wsdl:import** element
- A description containing WSDL extensions must not use them to contradict any other requirements set out in the profile
- When describing types, a declaration must not extend or restrict the soapenc:Array type
- Declarations must not use wsdl:arrayType in the type declaration

7.11 Service Publication/Discovery

- WS-I outlines specification guidelines for service publication and discovery requirements (e.g. UDDI)
  - UDDI v2.04 API Specification
  - UDDI v2.03 Data Structure Reference
  - UDDI Version 2 XML Schema

7.12 Security

- WS-I recognizes that security is an important issue, but also recognizes that security needs will vary on a service-by-service basis
- WS-I primarily discusses security in the context of encryption
  - Issues such as authentication and authorization are not discussed
- Relies primarily on HTTPS
- WS-I makes use of the following specifications for security:

  - RFC2818 HTTP Over TLS
  - RFC2246 The TLS Protocol Version 1.0
  - SSL Version 3.0
  - RFC 2459: X509 Public Key Infrastructure Certificate and CRL Profile

7.13 .NET Interoperability

- .NET supports document wrapped literal style only.
- .NET and Java can send and receive attachments using MTOM.
  - Prior to MTOM, there was no universally accepted standard for dealing with attachments.
- .NET tools showed problems with <wsdl:import> and <xsd:import>.
  - Avoid them or test the WSDL file early on.
- .NET tools do not support UTF-16 encoded WSDL file.
  - Convert to UTF-8.
- If a primitive type (such as xsd:int) is nillable, JAX-WS maps it to java.lang.Integer instead of int and handles null values well.
  - .NET always maps primitive XML types to primitive language types and can not handle null values.
  - .NET will throw error if a date time value is null.
- Java does not have unsigned integer types. Do not use it in schema.

7.14 .NET Interoperability

- Problem with arrays exist when:
  - The array is empty.
  - The array is null.
- Empty zero length array is serialized to XML as:
  - .NET: An empty element by .NET. <Customer/>. Java will deserialize it as a 1 length array.

    - Java: The entire element is omitted. Both Java and .NET will deserialize it as a null.
- A null array is serialized to XML as:
    - .NET and Java: The entire element is omitted. Both deserialize it as NULL.

Chapter 8 - JAX-WS Mapping Details

Objectives

Key objectives of this chapter

- Default WSDL to Java Mapping.
- Customizing WSDL to Java Mapping using WSDL annotations.
- Default Java to WSDL Mapping.
- Customizing Java to WSDL Mapping.
- Learn about JAX-WS and JAXB annotations in details.

8.1 Introduction to Mapping in JAX-WS

- JAX-WS provides rules to map the following WSDL entities to Java:
  - The <portType> to Service Endpoint Interface (SEI)
  - The <operation> elements to Java methods.
  - The <input> and <output> messages of an operation to input and output parameters of a Java method.
  - SOAP <binding> information to Java class.
- JAXB provides the rules to map XML schema data types to JavaBean classes.
- As a JAX-WS programmer, you need to know what the default mapping rules are.
- You will also need to know how to customize the default mapping rules.
- In this chapter, we will discuss customization through these techniques:
  - JAX-WS annotations (JSR 181).
  - JAXB annotations.
  - WSDL annotations.

Mapping

A Web Service uses standards like WSDL, SOAP and XML schema. JAX-WS on the other hand uses

Java to implement a service provider and consumer. JAX-WS provides the mechanism to map these two different worlds. For example, if you have defined a XML data type (element) called Customer, a JavaBean called Customer can represent that type in the Java world. According to the default mapping rule, the name of the Java class is same as the XML element. The Java package name for the class is by default derived from the namespace of the XML data type. XML data types are one of many things that need to be mapped. Here are a few examples:

- The <portType> element.
- The <service> element.
- The operations of a service (the <operation> elements).
- The input and output element names for the operations.
- The SOAP binding information.

JAX-WS provides extensive set of rules that govern these mappings. Java to XML data type mapping is a special case, because it is controlled by the JAXB specification. For basic Web services development, the default rules may be sufficient. In more advanced cases, you will need to customize the default. Customization is performed using these techniques:

- JAX-WS annotations in Java classes.
- JAXB annotations in Java classes.
- WSDL annotations in WSDL files.

JAXB and JAX-WS also provide an alternative to Java annotations. This approach uses external mapping files that specify the mapping rules. Usually, these mapping files are less intuitive to work with. Vendors do not uniformly support external mapping files. It is recommended that you use Java annotations as much as possible. In this chapter will focus mainly on the Java annotations.

8.2 Top-down and Bottom-up Mapping

- We will discuss the mapping rules and their customization separately for top-down and bottom-up development.
- Bottom-up uses **Java to WSDL** mapping.
- Top-down uses **WSDL to Java** mapping.
- Irrespective of the how the service implementation was created, the final result always depends on JAX-WS and JAXB annotations for mapping.
  - In top-down, the annotations are generated for you.
  - In bottom-up, you have to hand code the annotations.

Top-down and Bottom-up

JAX-WS provides two different approaches for a service provider implementation[3]. In top-down approach, the WSDL is designed first and the Java classes are generated from it. In the bottom-up approach, the Java classes are created first and the WSDL (including XML schema) is generated from it.

Despite the differences in the two approaches, one thing is certain. At the end of the day, the Java annotations control the mapping between WSDL and Java. For example, an implementation class is annotated with @WebService no matter which one of the two approaches you have taken to develop it. For top-down, these annotations will be generated for you. For bottom-up, you will have to hand code the annotations. At execution time, there is no difference between how a service implementation was created. Bottom-up and top-down are strictly development time approaches and have no impact on the actual runtime of the service.

Having said that at execution time, system solely depends on the annotations for mapping rules, we should discuss the mapping rules separately for top-down and bottom-up. This is best understood through an example. The package name of a Java class is mapped to the namespace of the corresponding element in WSDL. For example, the package name of the SEI is mapped to the namespace of the <portType>. If you are doing top-down, the package name will be derived from the namespace. If you doing bottom-up, the namespace will be derived from the package name. If you are doing top-down development, you will need to know how to set the namespace of a <portType> and how to customize the default namespace to Java package mapping rule. If you are doing bottom up development, you will need to know how to customize the package to namespace mapping rule.

8.3 WSDL to Java Mapping

- In the next few slides we will discuss the details of the WSDL to Java mapping.
- The mapping of the following entities are discussed:
  - The XML data types in the schema.
  - The <portType>
  - The <operation> elements and their <input> and <output> messages.
  - The SOAP <binding>
- The goal is to understand the default mapping rules. We will also discuss how to customize the mapping using WSDL file annotations.
  - Under no circumstances you should modify generated code (except for the business logic in the service implementation class) for

3 A consumer, on the other hand, is always generated in a top-down (WSDL first) manner.

customization.

- Throughout our discussion, we will use an example WSDL file.

WSDL to Java Mapping

In this approach, the WSDL file and the XML schema used by it are designed first. The Java classes required to implement the service are generated from it. The generator applies a default set of rules to create the Java classes. In the next few pages, we will learn about the default rules. We will also learn how to annotate the WSDL file to customize the mapping rules. We will cover the mapping rules for:

- The XML data types. These map to JavaBean classes.
- The <portType>. This mapped to the SEI.
- The SOAP <binding>. This maps to the implementation class.
- The <operation> elements and their <input> and <output> messages map to the methods in the SEI and the implementation class.

Example WSDL File

The mapping rules can be quite complex since the specification needs to cover many different possibilities. We will keep our discussion focused on the most common scenarios. We will use an example WSDL file and see how the different entities in it are mapped to Java. The example service is used to register a new user in an E-commerce web site. It has only one operation:

String RegisterUser(User user, Address shippingAddress, Address billingAddress)

The operation takes as input three parameters:

- User – The user object to add. The user ID and password are included in the object.
- Address – The default shipping address.
- Address – The default billing address.

The operation returns a status string. If the user is added successfully, "OK" is returned. Otherwise, an error code is returned. The complete WSDL is as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:ns="http://svc.webage.com"
    xmlns:data="http://svc.webage.com/data"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    targetNamespace="http://svc.webage.com">
    <wsdl:documentation>
        Please Type your service description here
    </wsdl:documentation>

    <wsdl:types>
```

```
        <xs:schema attributeFormDefault="qualified"
            elementFormDefault="qualified"
            targetNamespace="http://svc.webage.com/data"
            xmlns:data="http://svc.webage.com/data">

            <xs:complexType name="User">
                <xs:sequence>
                    <xs:element minOccurs="0" name="Password" type="xs:string"/>
                    <xs:element minOccurs="0" name="UserId" type="xs:string"/>
                </xs:sequence>
            </xs:complexType>
            <xs:complexType name="Address">
                <xs:sequence>
                    <xs:element minOccurs="0" name="City" type="xs:string"/>
                    <xs:element minOccurs="0" name="Street" type="xs:string"/>
                    <xs:element minOccurs="0" name="Zip" type="xs:string"/>
                </xs:sequence>
            </xs:complexType>

            <xs:element name="RegisterUser">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element minOccurs="0" name="UserToAdd"
                            type="data:User" />
                        <xs:element minOccurs="0" name="ShippingAddress"
                            type="data:Address" />
                        <xs:element minOccurs="0" name="BillingAddress"
                            type="data:Address" />
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="RegisterUserResponse">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element minOccurs="0" name="ResultStatus"
                            type="xs:string" />
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:schema>
    </wsdl:types>

    <wsdl:message name="RegisterUserRequest">
        <wsdl:part name="RegisterUserRequestPart"
            element="data:RegisterUser" />
    </wsdl:message>
    <wsdl:message name="RegisterUserResponse">
        <wsdl:part name="RegisterUserResponsePart"
            element="data:RegisterUserResponse" />
    </wsdl:message>
    <wsdl:portType name="AccountingAppPortType">
        <wsdl:operation name="RegisterUser">
            <wsdl:input message="ns:RegisterUserRequest" />
                <wsdl:output message="ns:RegisterUserResponse"/>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="AccountingAppSoap11Binding"
        type="ns:AccountingAppPortType">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
            style="document" />
        <wsdl:operation name="RegisterUser">
            <soap:operation soapAction="urn:RegisterUser"
                style="document" />
            <wsdl:input>
                <soap:body use="literal" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="AccountingAppService">
        <wsdl:port name="AccountingAppServiceHttpSoap11Endpoint"
            binding="ns:AccountingAppSoap11Binding">
            <soap:address
                location="http://host:8080/SvcWeb/services/AccountingAppService"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

8.4 XML Data Type to JavaBean Mapping

- JAXB rules are used for this mapping.
- Each <complexType> or <element> is mapped to a JavaBean class by the same name.
- The package name of a JavaBean class is derived from the namespace of the complex type or element.
  - Example: The namespace "http://svc.webage.com/data" maps to the package name "com.webage.svc.data".
- Each child element of a complex type maps to a field of the JavaBean.
- Example:

The XML type User is defined as follows:

```
<xs:complexType name="User">
    <xs:sequence>
        <xs:element minOccurs="0" name="Password"
type="xs:string"/>
```

```
        <xs:element minOccurs="0" name="UserId"
type="xs:string"/>
    </xs:sequence>
</xs:complexType>
```

8.5 XML Data Type to JavaBean Mapping

- It maps the the JavaBean class as follows:

```
package com.webage.svc.data;
import javax.xml.bind.annotation.*;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "User", namespace = "http://svc.webage.com/
data", propOrder = {
    "password",
    "userId"
})
public class User {

    @XmlElement(name = "Password")
    protected String password;
    @XmlElement(name = "UserId")
    protected String userId;
....//Getter and setter methods omitted
}
```

XML Data Type to JavaBean Mapping

This mapping is done according to the JAXB specification. Each <complexType> and <element> in the schema is mapped to a JavaBean class. The name of the class is same as the name of the complex type or the element. For example, the User and Address complex type will be mapped to the User and Address JavaBean class.

The package name of a JavaBean class is derived from the namespace of the XML data type. The namespace is set using the targetNamespace of the <xs:schema> element. In our example, the namespace is "http://svc.webage.com/data". According to JAXB, this will translate into the Java package name "com.webage.svc.data".

Each child element of a complex type is mapped to a field of the JavaBean. For example, the <UserId> child element of the User complex type will be mapped to the userId field of the User JavBean class.

The first letter of the child element is converted to lower case to derive the name of the filed. This presents a problem. Looking at the JavaBean property, we can not reverse engineer the XML element name with the correct case. For example, we can not say with certainty if the "userId" field should map to the "UserId" or "userId" element. This is why, the code generator annotates every field with the @XmlElement JAXB annotation. Example:

```
    @XmlElement(name = "UserId")
    protected String userId;
```

At execution time, the Web Services runtime uses the annotation to map the JavaBean field back to to the correct XML element.

Finally, the XML schema may require the child elements of a complex type to appear in a strict order. In our example, the use of <xs:sequence> requires that the <Password> and <UserId> child elements must appear in that order inside a User. JavaBean specification has no way to set the order of the fields. The solution is to apply the @XmlType JAXB annotation to the JavaBean class and specify the order. Example:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "User", namespace = "http://svc.webage.com/data", propOrder = {
    "password",
    "userId"
})
public class User {

    @XmlElement(name = "Password")
    protected String password;
    @XmlElement(name = "UserId")
    protected String userId;
....//Getter and setter methods omitted
}
```

The @XmlAccessType annotation is used to specify that at execution time system should directly access the field variables (instead of the getter or setter methods) to access the field values.

8.6 Mapping <portType> to the SEI

- For each <portType> a Java interface is created which serves as the SEI for the implementation.
  - The name of the interface is same as the name of the port type.
  - The package name is derived from the namespace of the port type.
- For each <operation> in the port type a method is added to the SEI.
  - The name is same as the name of the operation with the first letter converted to lower case.

- The <output> message of the operation, if any, are mapped to the return value of the method.
- The <input> message of the operation is mapped to the input parameters of the method.
    - The nature of the mapping depends on if you are using wrapped or non-wrapped style code generation.
    - Wrapped style code generation is designed for the "document literal wrapped" style of SOAP and is highly recommended.
- All <fault> elements of an operation are mapped to Java exception.
    - Details are covered in the error handling chapter.

8.7 Mapping <portType> to the SEI

- Example mapping:
- The <portType> is defined as follows:

```
<wsdl:portType name="AccountingAppPortType">
    <wsdl:operation name="RegisterUser">
        <wsdl:input message="ns:RegisterUserRequest" />
        <wsdl:output message="ns:RegisterUserResponse"/>
    </wsdl:operation>
</wsdl:portType>
```

8.8 Mapping <portType> to the SEI

- The generated SEI will look something like this:

```
package com.webage.svc;

@WebService(name = "AccountingAppPortType", targetNamespace =
"http://svc.webage.com")
public interface AccountingAppPortType {
@WebMethod(operationName = "RegisterUser",...)
@WebResult(name = "ResultStatus", targetNamespace =
"http://svc.webage.com/data")
//...
public String registerUser(
    @WebParam(name = "UserToAdd", targetNamespace =
"http://svc.webage.com/data")
    User userToAdd,
    @WebParam(name = "ShippingAddress", targetNamespace =
"http://svc.webage.com/data")
    Address shippingAddress,
    @WebParam(name = "BillingAddress", targetNamespace =
"http://svc.webage.com/data")
    Address billingAddress);

}
```

Mapping <portType> to the SEI

A <portType> describes the abstract interface of a service. JAX-WS quite appropriately maps the <portType> to the SEI. The name of the SEI is same as the <portType>. That will be "AccountingAppPortType". The package name is derived from namespace of the port type which is set using the targetNamespace of the <wsdl:definitions> element. In our example, the namespace is "http://svc.webage.com". The SEI is created in the com.webage.svc package.

Our example SEI will look something like this:

```
package com.webage.svc;

@WebService(name = "AccountingAppPortType", targetNamespace =
"http://svc.webage.com")
public interface AccountingAppPortType {
//...
}
```

Every <operation> in the <portType> is mapped to a method in the SEI. The name of the method is the name of the operation with the first letter converted to lower case. For example, the "RegisterUser" operation will map to the "registerUser" method. The @WebMethod annotation is used to preserve the original case of the operation's name. Example:

```
@WebMethod(operationName = "RegisterUser", action = "urn:RegisterUser")
//...
public String registerUser(...);
```

If an operation has an <output> element, it is mapped to the return value of the method. Unfortunately, Java's return value is anonymous. That is, unlike the input parameters of a method, the return value has no name. To work around the problem, the code generator uses the @WebResult annotation to preserve the name of the output element's name.

```
@WebMethod(operationName = "RegisterUser", action = "urn:RegisterUser")
@WebResult(name = "ResultStatus", targetNamespace = "http://svc.webage.com/data")
//...
public String registerUser(...);
```

Input Parameter Mapping

The nature of the input parameters of a method depends on if you are using wrapped or non-wrapped style SOAP. It is highly recommended that you use the "document literal wrapped" style of SOAP. The details are discussed in the SOAP chapter. We will discuss the mapping for that style only.

Note: JSR-181 specifies that the default SOAP binding is document/literal/wrapped, so by default, services mapped using the '@WebService' annotation will use the 'wrapped' style. If you would rather use a different style, you can include an '@SOAPBinding' annotation on the service class to override the defaults.

According to the wrapped style, the <input> message has a single part only. All input data is packaged in a single XML element. The name of that element is same as the operation. Our example WSDL uses the document literal wrapped style. The <RegisterUser> XML element is used as the input for the RegisterUser operation. Notice how this element wraps all input data:

```
<xs:element name="RegisterUser">
    <xs:complexType>
        <xs:sequence>
            <xs:element minOccurs="0" name="UserToAdd"
                                type="data:User" />
            <xs:element minOccurs="0" name="ShippingAddress"
                                type="data:Address" />
            <xs:element minOccurs="0" name="BillingAddress"
                                type="data:Address" />
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

When you generate code for the wrapped style, the method gets a separate input parameter for each child element of the input element. The name of the parameter will be the name of the child element with the first letter converted to lowercase. For example, the registerUser method will get these input parameters:

- userToAdd of type User.
- shippingAddress of type Address.
- billingAddress of type Address.

The @WebParam annotation is used to preserve the original case of the element names. Example:

```
@WebMethod(operationName = "RegisterUser", action = "urn:RegisterUser")
@WebResult(name = "ResultStatus", targetNamespace = "http://svc.webage.com/data")
//...
public String registerUser(
    @WebParam(name = "UserToAdd", targetNamespace = "http://svc.webage.com/data")
    User userToAdd,
    @WebParam(name = "ShippingAddress", targetNamespace =
"http://svc.webage.com/data")
    Address shippingAddress,
    @WebParam(name = "BillingAddress", targetNamespace =
"http://svc.webage.com/data")
```

```
    Address billingAddress);
```

The Java code does not have to deal with the wrapper data type RegisterUser. If you choose wrapped style code generation, system will unwrap the contents of the wrapper and use each child element as input for the method. This makes programming a little easier.

Non-wrapped style code generation is appropriate when you are not using the document literal wrapped style of SOAP. In this case, there is no single XML element that wraps all input data. The <input> message can have many parts (one of each input parameter). If we generated code in non-wrapped style for our example WSDL, we will get this method signature.

```
@WebMethod(operationName = "RegisterUser", action = "urn:RegisterUser")
@WebResult(name = "RegisterUserResponse", targetNamespace = "http://svc.webage.com/
data", partName = "RegisterUserResponsePart")
public RegisterUserResponse registerUser(
    @WebParam(name = "RegisterUser", targetNamespace =
"http://svc.webage.com/data", partName = "RegisterUserRequestPart")
    RegisterUser registerUserRequestPart);
```

Detailed analysis of the non-wrapped style code generation is outside the scope of this class.

8.9 Mapping the SOAP <binding>

- The SOAP binding is mapped to the service implementation class.
- The SEI of the implementation is set to the interface derived from the <portType>.
- The code generator creates a skeletal class. Developers will have to provide the business logic for the methods.
- Example:

```
@WebService
(endpointInterface="com.webage.svc.AccountingAppPortType",
targetNamespace="http://svc.webage.com",
serviceName="AccountingAppService",
portName="AccountingAppServiceHttpSoap11Endpoint")
public class AccountingAppSoap11BindingImpl{

    public String registerUser(User userToAdd, Address
shippingAddress, Address billingAddress) {
        // TODO: Provide business logic
        return null;
    }
}
```

Mapping the SOAP <binding>

The SOAP <binding> section is mapped to the service implementation class. The name of the class is name of the binding plus "Impl". In our example, the name of the binding is "AccountingAppSoap11Binding". The service implementation class is named "AccountingAppSoap11BindingImpl". The package name will be derived from the namespace of the binding which is set using the targetNamespace attribute of the <wsdl:definitions> element.

System will generate a skeletal class. Developers will have to provide the business logic for the methods. In our case, the class will look something like this.

```
@WebService (endpointInterface="com.webage.svc.AccountingAppPortType",
targetNamespace="http://svc.webage.com", serviceName="AccountingAppService",
portName="AccountingAppServiceHttpSoap11Endpoint")
public class AccountingAppSoap11BindingImpl{

    public String registerUser(User userToAdd, Address shippingAddress, Address
billingAddress) {
        // TODO Auto-generated method stub
        return null;
    }
}
```

Notice, how the SEI is set using the endpointInterface attribute of @Webservice.

8.10 Customizing WSDL to Java Mapping

- JAX-WS provides a set of WSDL extension elements to customize the default mapping rules.
    - These XML elements are available from the http://java.sun.com/xml/ns/jaxws namespace.
- Example:
- To change package name of SEI and implementation class:

```
<wsdl:definitions ...>
<jaxws:package name="com.xyz.service.accounting"/>
...
</wsdl:definitions>
```

8.11 Customizing WSDL to Java Mapping

- To change the name of the SEI:

```
<wsdl:portType name="AccountingAppPortType">
<jaxws:class name="com.foo.bar.AccountSvcSEI"/>
...
</wsdl:portType>
```

- To change the name of a method:

```
<wsdl:operation name="RegisterUser">
<jaxws:method name="registerFunkyUser"/>
...
</wsdl:operation>
```

Customizing WSDL to Java Mapping

JAX-WS provides a few WSDL extension elements that can be used to customize the default mapping behavior. The code generator is supposed to honor the existence of these extension elements. These elements are officially known as "binding declarations". Although, they are popularly also referred to as WSDL annotations. They are *not* Java annotations but XML elements that extend WSDL. Here, we will discuss the most commonly used customizations.

The extension elements are available from the "http://java.sun.com/xml/ns/jaxws" namespace. As a result, first define a namespace prefix as shown in boldface below:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
          xmlns:ns="http://svc.webage.com"
          xmlns:data="http://svc.webage.com/data"
          xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
          targetNamespace="http://svc.webage.com">
```

To change the package name of the generated SEI and implementation class, use the <jaxws:package> element directly as a child of the <wsdl:definitions> element. For example:

```
<wsdl:definitions ...>
<jaxws:package name="com.xyz.service.accounting"/>
...
</wsdl:definitions>
```

To select wrapped style code generation, use the <jaxws:enableWrapperStyle> element under <wsdl:definitions>. Note, your vendor's code generation tool may have a GUI control, such as a checkbox, for the same purpose which you can use as an alternative.

```
<wsdl:definitions ...>
<jaxws:enableWrapperStyle>true</jaxws:enableWrapperStyle>
...
</wsdl:definitions>
```

To change the name of the SEI, use the <jaxws:class> element within a <wsdl:portType>. For example, the following will generate a SEI by the name "com.foo.bar.AccountSvcSEI".

```
<wsdl:portType name="AccountingAppPortType">
<jaxws:class name="com.foo.bar.AccountSvcSEI"/>
...
</wsdl:portType>
```

To change the name of a Java method, use the <jaxws:method> element inside <wsdl:operation>. For example, the following will create a method by the name "registerFunkyUser" instead of "registerUser".

```
<wsdl:operation name="RegisterUser">
          <jaxws:method name="registerFunkyUser"/>

          <wsdl:input message="ns:RegisterUserRequest" />
          <wsdl:output message="ns:RegisterUserResponse"/>
</wsdl:operation>
```

8.12 Java to WSDL Mapping

- JAX-WS applies a default set of rules to generate WSDL and XML schema from the Java classes.
- You can customize these rules using JAX-WS and JAXB annotations.
- We will learn about the default rules and ways to customize them.

Java to WSDL Mapping

Java to WSDL mapping is of course the reverse of the WSDL to Java mapping. For example, here the SEI is mapped to <portType>. The WSDL file and the associated XML schema is automatically generated from the Java classes. JAX-WS applies a set of default mapping rules for this. You can customize these rules using JAX-WS and JAXB annotations.

The entities are mapped as follows:

- JavaBean classes used in the SEI method signature are mapped to XML data types.
- The SEI, if present, is mapped to <portType>. Otherwise, the port type is mapped from the implementation class.
- The <operation> elements are generated from the methods in the SEI.
- SOAP <binding> is generated from the SEI, if present, otherwise from the implementation

class.

Example Java Classes

We will use an example Web service implementation throughout. We will use the same user registration Web Service that we have already seen.

The JavaBean classes are as follows:

```
package com.webage.svc.data;

public class User {

    protected String password;
    protected String userId;

    //Getter and setter methods omitted
    //to save space
}

public class Address {

    protected String city;
    protected String street;
    protected String zip;

    //Getter and setter methods omitted
    //to save space
}
```

The SEI is as follows:

```
package com.webage.svc;

@WebService
public interface AccountingAppPortType {
    public String registerUser(
        User userToAdd,
        Address shippingAddress,
        Address billingAddress);
}
```

The implementation class is as follows:

```
package com.webage.svc;

@javax.jws.WebService (endpointInterface="com.webage.svc.AccountingAppPortType")
public class AccountingAppSoap11BindingImpl{

    public String registerUser(User userToAdd, Address shippingAddress, Address
billingAddress) {
        // Implementation code
        return "OK";
    }
}
```

8.13 JavaBean to XML Mapping

- JAX-WS uses JAXB rules to map JavaBeans into XML data types.
- A JavaBean class is mapped to an XML complex type.
- You can optionally use JAXB annotations like @XmlType and @XmlElement to customize the mapping.
- Example:

The JavaBean:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "UserType", namespace =
"http://svc.webage.com/data")
public class Address {
    @XmlElement(name = "UserPassword")
    protected String password;
    @XmlElement(name = "LogonUserId")
    protected String userId;
    //...
}
```

8.14 JavaBean to XML Mapping

- Will be mapped to:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" ...
 targetNamespace="http://svc.webage.com/data">

  <xs:complexType name="UserType">
    <xs:sequence>
      <xs:element name="UserPassword" type="xs:string"
minOccurs="0"></xs:element>
      <xs:element name="LogonUserId" type="xs:string"
minOccurs="0"></xs:element>
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

JavaBean to XML Mapping

JAXB specification applies here. A JavaBean class is mapped to a <complexType>. The name of the type is same as the class with the first letter converted to lower case. Each field of the JavaBean is mapped to a child element. The name of the element is same as the field. For example, the User class is mapped to this:

```
<xs:complexType name="user">
    <xs:sequence>
      <xs:element name="password" type="xs:string" minOccurs="0"></xs:element>
      <xs:element name="userId" type="xs:string" minOccurs="0"></xs:element>
    </xs:sequence>
</xs:complexType>
```

The namespace is derived from the package of the SEI (and not the JavaBean class!). In our case, the namespace will be "http://svc.webage.com/".

You can use the @javax.xml.bind.annotation.XmlType annotation to change the generate element's name and namespace.

```
@XmlType(name = "UserType", namespace = "http://svc.webage.com/data")
public class User {
...
}
```

You can use the @javax.xml.bind.annotation.XmlElement to change the name of the children elements. If you apply this annotation to the member variables, you should also set the accessor type to field.

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "UserType", namespace = "http://svc.webage.com/data")
public class Address {
    @XmlElement(name = "UserPassword")
    protected String password;
    @XmlElement(name = "LogonUserId")
    protected String userId;
    //...
}
```

The generated schema will look like this (customization shown in bold face):

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" ...
 targetNamespace="http://svc.webage.com/data">

  <xs:complexType name="UserType">
    <xs:sequence>
      <xs:element name="UserPassword" type="xs:string" minOccurs="0"></xs:element>
      <xs:element name="LogonUserId" type="xs:string" minOccurs="0"></xs:element>
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

8.15 Mapping SEI to <portType>

- The SEI must be annotated with @WebService.
- By default the name of the port type is same as the SEI.
- The namespace of the port type is derived from the package name of the SEI.
  - Example: The package name "com.webage.svc" is translated into the namespace "http://svc.webage.com".
- The @WebService annotation can be used to customize the name of the port type and its namespace.
- Example:

The SEI:

```
@WebService(name = "AccountingAppInterface",
targetNamespace = "http://svc.webage.com/accounting")
public interface AccountingAppPortType {
...
}
```

8.16 Mapping SEI to <portType>

Will be mapped to:

```
<definitions name="AccountingAppService"
    targetNamespace="http://svc.webage.com/accounting" ...>
...
  <portType name="AccountingAppInterface">
...
  </portType>
</definitions>
```

Mapping SEI to <portType>

SEI is a Java interface that is annotated using @javax.jws.WebService. In the absence of an SEI, an SEI will be implicitly generated from the service implementation class. By default, the name of the <portType> is same as the SEI. The namespace is derived from the Java package name. In our example, the port type's name will be AccountingAppPortType. The namespace will be "http://svc.webage.com".

```
<definitions name="AccountingAppService"
    targetNamespace="http://svc.webage.com" ...>

...
  <portType name="AccountingAppPortType">
...
  </portType>
</definitions>
```

You can use the @WebService annotation to change the name of the port type and the namespace. The name attribute of the @WebService annotation sets the name of the <portType>.

```
@WebService(name = "AccountingAppInterface", targetNamespace =
"http://svc.webage.com/accounting")
public interface AccountingAppPortType {
...
}
```

This will generate the following port type. The customized portions are highlighted.

```
<definitions name="AccountingAppService"
    targetNamespace="http://svc.webage.com/accounting" ...>

...
  <portType name="AccountingAppInterface">
...
  </portType>
</definitions>
```

8.17 Mapping Java Method to <operation>

- Each method of the SEI is mapped to an <operation> in the <portType>
- By default, the name of the operation is same as the name of the method.
- You can customize the name of the operation using the @WebMethod annotation.
- Example:

The method:

```
@WebMethod(operationName = "RegisterNewUser")
```

```
public String registerUser(
    User userToAdd,
    Address shippingAddress,
    Address billingAddress);
```

8.18 Mapping Java Method to <operation>

Will be mapped to:

```
  <portType name="AccountingAppInterface">
    <operation name="RegisterNewUser">
      <input message="tns:registerUser">
      </input>
      <output message="tns:registerUserResponse">
      </output>
    </operation>
  </portType>
```

Mapping Java Method to <operation>

Each method in the SEI will be mapped to an <operation> in the <portType>. The name of the operation is same as the name of the method. The registerUser will be mapped as follows:

```
  <portType name="AccountingAppInterface">
    <operation name="registerUser">
      <input message="tns:registerUser">
      </input>
      <output message="tns:registerUserResponse">
      </output>
    </operation>
  </portType>
```

The @WebMethod annotation can be used to change name of the operation. For example:

```
@WebMethod(operationName = "RegisterNewUser")
public String registerUser(
    User userToAdd,
    Address shippingAddress,
    Address billingAddress);
```

This will create the operation as follows.

```
<operation name="RegisterNewUser">
...
</operation>
```

8.19 Input Parameter Mapping

- JAX-WS produces mapping suitable for document literal wrapped style SOAP. This is the recommended approach.
- A wrapper XML element is defined that includes all input parameters.
  - The name of the element is same as the operation name that is mapped to the Java method.
- The wrapper type contains a child element for each input parameter taken by the Java method. The names of these elements are arg0, arg1, arg2 and so on.
- You can change the names of the child elements using the @WebParam annotation.
- Example:

The method:

```
@WebMethod(operationName = "RegisterNewUser")
public String registerUser(
    @WebParam(name = "UserToAdd")
    User userToAdd,
    @WebParam(name = "ShippingAddress")
    Address shippingAddress,
    @WebParam(name = "BillingAddress")
    Address billingAddress);
```

8.20 Input Parameter Mapping

Will correspond to a SOAP body something like this:

```
<RegisterNewUser>
    <UserToAdd>...</UserToAdd>
    <ShippingAddress>...</ShippingAddress>
    <BillingAddress>...</ BillingAddress>
</RegisterNewUser>
```

Input and Output Mapping

System by default generates document literal wrapped style WSDL. This is the recommended approach. In this approach all input parameters for a method will be wrapped in a single XML element. The name of this wrapper XML element will be same as the name of the operation.

The wrapper element will contain a child element for each input parameter that the method takes. By default, the names of these child elements will be arg0, arg1 and so on. Consider the registerUser method:

```
@WebMethod(operationName = "RegisterNewUser")
public String registerUser(
    User userToAdd,
    Address shippingAddress,
    Address billingAddress);
```

A complex type called RegisterNewUser will be created. This is the input wrapper data type.

```
<xs:complexType name="RegisterNewUser">
  <xs:sequence>
    <xs:element name="arg0" type="ns1:UserType" minOccurs="0"></xs:element>

    <xs:element name="arg1" type="ns1:Address" minOccurs="0"></xs:element>
    <xs:element name="arg2" type="ns1:Address" minOccurs="0"></xs:element>
  </xs:sequence>
</xs:complexType>
```

Finally, an XML element also by the name RegisterNewUser will be defined.

```
<xs:element name="RegisterNewUser" type="tns:RegisterNewUser"></xs:element>
```

The wrapper complex type and element will be created in the namespace of the port type.

You can not change the name of the element. According the document literal wrapped style, it must be same as the operation name. You can, however, change the name and namespace of the child elements. This is done by annotating the input parameters using the @javax.jws.WebParam annotation.

```
@WebMethod(operationName = "RegisterNewUser")
public String registerUser(
    @WebParam(name = "UserToAdd", targetNamespace = "http://svc.webage.com/data")
    User userToAdd,
    @WebParam(name = "ShippingAddress", targetNamespace =
"http://svc.webage.com/data")
    Address shippingAddress,
    @WebParam(name = "BillingAddress", targetNamespace =
"http://svc.webage.com/data")
    Address billingAddress);
```

This will produce the following schema for the wrappers.

```
<xs:schema ...
    xmlns:tns="http://svc.webage.com/data"
    targetNamespace="http://svc.webage.com/data">

  <xs:element name="BillingAddress" type="tns:Address"></xs:element>
  <xs:element name="ShippingAddress" type="tns:Address"></xs:element>
  <xs:element name="UserToAdd" type="tns:UserType"></xs:element>
</xs:schema>

<xs:schema
```

```
    xmlns:tns="http://svc.webage.com/accounting"
    xmlns:ns1="http://svc.webage.com/data"
    targetNamespace="http://svc.webage.com/accounting">

<xs:element name="RegisterNewUser" type="tns:RegisterNewUser"></xs:element>

<xs:complexType name="RegisterNewUser">
    <xs:sequence>
      <xs:element ref="ns1:UserToAdd" minOccurs="0"></xs:element>
      <xs:element ref="ns1:ShippingAddress" minOccurs="0"></xs:element>
      <xs:element ref="ns1:BillingAddress" minOccurs="0"></xs:element>
    </xs:sequence>
</xs:complexType>

</xs:schema>
```

Now, the names of the child elements of the RegisterNewUser wrapper type has changed. The wrapper itself is defined in the namespace of the port type – "http://svc.webage.com/accounting". But, the namespace of the child elements is now "http://svc.webage.com/data".

8.21 Method Output Mapping

- As per the document literal wrapped style, an XML element will include all the response data.
  - The name of this element is made up of the name of the operation plus "Response".
- This element will contain a child element called "return" which will hold the output data.
- You can change the name of the child element using the @WebResult annotation applied to the method.
- Example:

```
@WebMethod(operationName = "RegisterNewUser")
@WebResult(name = "ResultStatus")
public String registerUser(...);
```

8.22 Method Output Mapping

Will translate to a SOAP response body like this:

```
<RegisterNewUserResponse>
```

```
    <ResultStatus>
    ...
    </ResultStatus>
</RegisterNewUserResponse>
```

Method Output Mapping

System will use the document literal wrapped style by default. An XML data type and element will be created for the return value. The names of these will be the name of the operation plus "Response". The type will have a child element called "return" that will hold the return data. For example:

```
<xs:complexType name="RegisterNewUserResponse">
    <xs:sequence>
      <xs:element name="return" type="xs:string" minOccurs="0"></xs:element>
    </xs:sequence>
</xs:complexType>

<xs:element name="RegisterNewUserResponse"
type="tns:RegisterNewUserResponse"></xs:element>
```

You can control the name of the child element using the @WebResult annotation.

```
@WebMethod(operationName = "RegisterNewUser")
@WebResult(name = "ResultStatus")
public String registerUser(...);
```

Now, the response data type will be as follows.

```
<xs:complexType name="RegisterNewUserResponse">
    <xs:sequence>
      <xs:element name="ResultStatus" type="xs:string" minOccurs="0"></xs:element>
    </xs:sequence>

</xs:complexType>
```

8.23 Bare Input and Output Mapping

- Document literal bare style is a variation of the document literal wrapped style.
- This style applies only when the Java method takes zero or one input parameter.
- This style reduces the number of XML types defined in the schema.
  - Essentially, no complex type is defined for the input wrapper. The wrapper element is made directly of the type of the sole input parameter of the method.

- Use the @SOAPBinding annotation to enable document bare style.
- Example:

```
@SOAPBinding(parameterStyle=ParameterStyle.BARE)
public Address findBillingAddress(User userToFind);
```

This will require the definition of the wrapper element only. The element will be of the type of the input parameter.

```
<xs:element name="findBillingAddress" type="ns1:User"/>
```

Bare Input and Output Mapping

The document literal bare style is a slight variation of the document literal wrapped style. It eliminates the need to define a new XML data type for the wrapper. This style is enabled using the @SOAPBinding annotation at the SEI level or at an individual method level.

The bare style applies when the Java method takes no more than one input parameter. No complex type is defined for the wrapper. A wrapper element is defined whose type is same as the type of the input. This is best learned through an example. Let us say that there is a method in the SEI:

```
public Address findBillingAddress(User userToFind);
```

In document literal wrapped style, a complex type and an XML element by the name findBillingAddress will be created.

```
<xs:element name="findBillingAddress" type="tns:findBillingAddress"></xs:element>
<xs:complexType name="findBillingAddress">
    <xs:sequence>
      <xs:element name="arg0" type="ns1:User" minOccurs="0"></xs:element>
    </xs:sequence>
  </xs:complexType>
```

The SOAP body will look like this.

```
<soap:Body>
    <ns:findBillingAddress>
        <arg0>
            <userId>...</userId>
            <password>...</password>
        </arg0>
    </ns:findBillingAddress>
</soap:Body>
```

The bare style will not create the complex type. The findBillingAddress element will still be created, since it is required by the document literal wrapped style. It's type will be directly set to the User.

First, we will need to annotate the method to use the document bare style.

```
@SOAPBinding(parameterStyle=ParameterStyle.BARE)
```

```
public Address findBillingAddress(User userToFind);
```

Now, only the wrapper element will be defined.

```
<xs:element name="findBillingAddress" nillable="true" type="ns1:User"/>
```

The SOAP body will now contain one less element.

```
<soap:Body>
	<ns:findBillingAddress>
		<userId>...</userId>
		<password>...</password>
	</ns:findBillingAddress>
</soap:Body>
```

The bare style creates less XML schema types but should be compatible to the document literal wrapped style. Remember, the document bare style is possible only when the Java method takes zero or one input parameter.

8.24 RPC Literal Style

- You can change the default SOAP style from document literal wrapped to RPC literal.
- This is done using the @SOAPBinding annotation at the SEI or method level.

```
@WebService
@SOAPBinding(style=Style.RPC)
public interface AccountingAppPortType {
...
}
```

- Although, JAX-WS does not completely eliminate the possibility, you should not use the document encoded or RPC encoded style. SOAP encoding is not WS-I compliant.

RPC Literal Style

By default, JAX-WS mapping uses the document literal wrapped style. You can optionally enable document literal bare style if the Java method has no more than one input. These are the recommended styles. However, in some cases you may need to use the RPC literal style. Please note, JAX-WS strongly discourages the use of any encoded style (either document or RPC). Encoded style is not WS-I compliant. As a result stick to document literal or RPC literal style.

To enable RPC literal style, use the @SOAPBinding annotation at the SEI or method level. SEI level configuration is better. Using different styles for different methods will make the service unnecessarily complicated to use.

For example, let us say that our SEI uses RPC literal.

```
@WebService
@SOAPBinding(style=Style.RPC)
public interface AccountingAppPortType {
    public String registerUser(
        User userToAdd,
        Address shippingAddress,
        Address billingAddress);
}
```

Now, the input message for the registerUser operation will contain multiple parts, one for each input parameter.

```
<message name="registerUser">
  <part name="arg0" type="ns1:UserType"></part>
  <part name="arg1" type="ns1:Address"></part>
  <part name="arg2" type="ns1:Address"></part>
</message>

<message name="registerUserResponse">
  <part name="return" type="xsd:string">
  </part>
</message>

<portType name="AccountingAppInterface">
  <operation name="registerUser" parameterOrder="arg0 arg1 arg2">
    <input message="tns:registerUser"></input>
    <output message="tns:registerUserResponse"></output>
  </operation>
</portType>
```

8.25 Service Provider Annotation

- In some cases, we want to take more control of the process of handling the request.
  - For example, if we want to parse the XML directly rather than let JAXB process it.
- For these situations, we can implement a **Web Service Provider**
- Instead of implementing the Service Endpoint Interface, the provider class implements Provider<T>, where type T can be
  - javax.xml.transform.Source
  - javax.xml.soap.SOAPMessage

- ◊ Any other type of request that the particular JAX-WS implementation might support

- A Web Service Provider's 'T invoke(T request)' method is invoked when a request arrives.
- The return value is the response, encapsulated in the same type class as the request.

8.26 Web Service Provider Example

```
@WebServiceProvider(wsdlLocation =
"WEB-INF/wsdl/Hello.wsdl",
serviceName = "HelloServiceProvider", portName =
"HelloPort",
targetNamespace = "http://jaxws.demo.webage.com/")
@ServiceMode(Service.Mode.MESSAGE)
public class HelloProvider implements Provider<Source> {
    public Source invoke(Source requestSource) {
        try {
            System.out.println("Got a request as a " +
                    requestSource.getClass());
            Transformer t =
      TransformerFactory.newInstance().newTransformer();

            StringWriter sw = new StringWriter();
            t.transform(requestSource, new
StreamResult(sw));
            System.out.printf("Message was : %s\n",
sw.toString());
        } catch (TransformerException ex) {

Logger.getLogger(HelloProvider.class.getName()).
              log(Level.SEVERE, null, ex);
        }
        return null;
    }  }
```

Notes on the example:

The sample code above simply copies the incoming message to the system console. It doesn't return any message. Using the Transformer isn't strictly part of using the service provider contract; we're just using it here as a simple way to copy the incoming message to an output stream.

8.27 Service Provider Annotations

- The service provider is marked with the '@WebServiceProvider' annotation.
- Can set the following attributes:
  - wsdlLocation: Location of the wsdl file
  - serviceName: Name of the service
  - portName: Port name to use in the wsdl
  - targetNameSpace: Target name space to use in the wsdl

8.28 Service Provider Annotations

- A Provider can also be annotated with
  - @ServiceMode(value="...")
  - where value is either
    - Service.Mode.MESSAGE
      - Provider will process the entire message
    - Service.Mode.PAYLOAD
      - Provider will process the contents of the soap body only

8.29 JAX-WS Clients

- JAX-WS implementations can generate a client from a WSDL file
- The generated client can be used to retrieve a proxy that allows you to access the web service that was defined in the WSDL file.
- JAX-WS clients can present both synchronous and asynchronous service

endpoint interfaces.

8.30 Synchronous and Asynchronous Calls

- If we had a synchronous call that looked like:

```
public String sayHello(
    @WebParam(name = "arg0", targetNamespace = "")
    String arg0);
```

- then if we trigger the JAX-WS code generator to generate asynchronous clients (implementation-specific), we would get

```
public Future<?> sayHelloAsync(
    String arg0,
    AsyncHandler<SayHelloResponse> asyncHandler);

public Response<SayHelloResponse> sayHelloAsync(
    String arg0);
```

8.31 Synchronous and Asynchronous Calls

- The 'Response<T>' interface defines extends 'java.util.concurrent.Future<T>'
- In addition to the 'Future' methods, lets you retrieve the response context.
- The 'AsyncHandler<T>' interface defines an asynchronous response handler (void **handleResponse**(Response<T> res))
  - the web services subsystem calls this response handler when a response has been obtained.
  - You can also call methods on the 'Future' that is returned, to check on the status, or cancel the request.

8.32 Summary

- JAX-WS controls how WSDL and XML schema will be mapped to Java classes and interfaces.
- JAX-WS and JAXB annotations are used to define this mapping.

- When you generate Java classes from WSDL, annotations are automatically added to the generated classes.
- If you are doing bottom-up development, you will have to use annotations to control the generated WSDL and XML schema used by the service.

Chapter 9 - Advanced JAX-WS API

Objectives

Key objectives of this chapter

- How to execute a Web Service in a minimal server environment by publishing it.
- How to work with the SOAP request and message context in the server side.
- How to work with the SOAP request and message context in the client side.
- How to develop a high performance Web Service that works with the raw SOAP XML documents.
- How to develop a client that works with the raw SOAP documents.
- Using these APIs in advanced cases.

9.1 Publishing a Web Service

- Publishing a service allows us to run a Web Service in a lightweight server environment without the need of a Java EE application server.
- A JAX-WS Web Service is really meant to run in a Web or EJB container. Do not use this approach in production.
- This approach can be useful to publish a Web Service interface from a desktop Java application.
- Use the publish() method of the javax.xml.ws.Endpoint class to launch a Web Service. It takes two parameters:
  - ◊ The endpoint URL.
  - ◊ An instance of the implementation class.
- Example. Let us say that the implementation class is CalculatorService. We will launch the service from a Java application.

```
Object implementor = new CalculatorService();
String address =
"http://localhost:8080/axis2/services/CalcService";
Endpoint.publish(address, implementor);
```

Publishing a Service

A JAX-WS Web Service is meant to execute within a EJB or Web container. However, it also provides a way to execute a Web Service in a lightweight server runtime. This option is meant for quick testing and proof of concept only in a situation where a full blown Java EE EJB or Web Container is not available. We do not recommend that you take this approach. We discuss it here for your information only.

This option can be useful where you have a Java desktop application that wishes to expose a Web service interface to interact with external programs. Using the API described here, any Java application can act as a Web Service provider. There is no need to package a complete Java EE server environment.

First, develop a Web Service implementation class, either using top-down or bottom-up approach. Then develop a Java application (a Java class with main() method) that publishes the implementation. Publishing is done using the javax.xml.ws.Endpoint class. For example, let us say that we have an implementation class called CalculatorService.

```
@WebService
public class CalculatorService {
    public int add(int value1, int value2) {
        //...
    }
}
```

Develop the application class that will publish an instance of the implementation class.

```
import javax.xml.ws.Endpoint;

public class CalculatorServer{

    protected CalculatorServer() throws Exception {
        System.out.println("Starting Server");
        Object implementor = new CalculatorService();
        String address = "http://localhost:8080/axis2/services/CalcService";
        Endpoint.publish(address, implementor);

        synchronized(Thread.currentThread()) {
            Thread.currentThread().wait();
        }
    }

    public static void main(String args[]) throws Exception {
        CalculatorServer svr = new CalculatorServer();
        System.out.println("Server ready...");
    }
}
```

The publish() method of the Endpoint class will begin to wait for SOAP requests to arrive over HTTP. The method takes two parameters:

- The endpoint URL of the Web Service. The method will use the endpoint URL address supplied (http://localhost:8080/axis2/services/CalcService) to determine the port number to listen to. It will bind the path of the URL to the Web Service.
- An instance of the service implementation class.

Not all vendors may support publishing of a service. Axis2 supports it. Some vendors will allow running a service in a EJB or Web container only.

9.2 Web Service Context

- A service implementation class can use the Web Service context to retrieve or set various meta information about a SOAP request and response.
- The context is obtained by annotating a member variable of type WebServiceContext with @Resource.

```
@javax.jws.WebService
public class AccountingAppSoap11BindingImpl{

    @Resource
    WebServiceContext ctx;
...
}
```

- Key methods of the WebServiceContext interface are:
  - Principal getUserPrincipal()
  - boolean isUserInRole(String role)
  - MessageContext getMessageContext()
- MessageContext gives us access to the meta information for the request and response.

Web Service Context

A service implementation class can access various meta information about a SOAP request and response using the service context. For example, you can find out the name of the <operation> invoked by the SOAP request and you can control the HTTP status code in the reply.

The service context is obtained by the implementation class by defining a variable of type javax.xml.ws.WebServiceContext and annotating it with with @javax.annotation.Resource. In other words, regular Java EE resource injection is used to obtain the service context. For example:

```
@javax.jws.WebService
public class AccountingAppSoap11BindingImpl{

    @Resource
    WebServiceContext ctx;
...
}
```

Key methods of the WebServiceContext interface are as follows.

Method	**Description**
java.scurity.Principal getUserPrincipal()	Returns the user object associated with the SOAP call. Authentication can be enabled using either HTTP basic authentication or WS-Security. The user credentials submitted in the SOAP request are used to validate the user. The Principal object contains information about that user. This method can come handy for audit logging.
boolean isUserinRole(String role)	Returns true if a user belongs to a role. Role based access control is currently not specified by JAX-WS. You vendor may or may not provide support for this as an extension.
MessageContext getMessageContext()	The message context object contains meta information about the request. The service implementation can also store response specific information there. We will discuss the details of message context next.

Example usage:

```
public String registerUser(User userToAdd, Address shippingAddress, Address
billingAddress) {
    logger.info("registerUser invoked by: " + ctx.getUserPrincipal().getName());

    //...
}
```

9.3 Message Context

- A message context is a hashmap that contains various properties of a SOAP request and response.
- It is an interface javax.xml.ws.handler.MessageContext and obtained from

the service context by calling getMessageContext().

```
MessageContext mc = ctx.getMessageContext();
```

- Use the get() method to retrieve a property. Various property names are defined as constants of MessageContext.

```
QName op = (QName) mc.get(MessageContext.WSDL_OPERATION);
```

- Use the put() method to set the value of a response property.

```
mc.put(MessageContext.HTTP_RESPONSE_CODE,
    new Integer(404));
```

Message Context

A service implementation uses the message context object to get or set information about a SOAP request or response. The interface javax.xml.ws.handler.MessageContext represents a message context. The context object is retrieved from the service context by calling the getMessageContext() methdo.

The MessageContext interface extends the java.util.Map interface. As a result, it is basically a hashtable. Various property values about the request and response are stored in that hashtable. The key of a property is a String and stands for the name of the property. The value can be any object. The MessageContext interface provides a set of predefined property names. For example, the following will retrieve the <operation> name for the SOAP request.

```
@Resource
WebServiceContext ctx;

public String registerUser(User userToAdd, Address shippingAddress, Address
billingAddress) {
    MessageContext mc = ctx.getMessageContext();

    QName op = (QName) mc.get(MessageContext.WSDL_OPERATION);
    logger.info("operation name: " + op.getLocalPart());

    //...
}
```

To change the value of the response property, use the put method. The following example sets the HTTP reply status to 404 File Not Found.

```
public String registerUser(User userToAdd, Address shippingAddress, Address
billingAddress) {
    MessageContext mc = ctx.getMessageContext();

    mc.put(MessageContext.HTTP_RESPONSE_CODE, new Integer(404));

    //...
}
```

An advanced service can use some of the useful properties stored in the message context.

Property name in MessageContext	**Description**
WSDL_SERVICE	The name of the <service>. Type: QName.
WSDL_PORT	The name of the <port>. Type: QName.
WSDL_INTERFACE	The name of the <portType>. Type: QName.
WSDL_OPERATION	The name of the <operation>. Type: QName.
SERVLET_REQUEST	The javax.servlet.http.HttpServletRequest object.
SERVLET_RESPONSE	The javax.servlet.http.HttpServletResponse object.
SERVLET_CONTEXT	The javax.servlet.ServletContext object.
HTTP_RESPONSE_CODE	The HTTP reply status. Type: Integer.

9.4 Working With Raw XML

- By default JAX-WS will map SOAP XML data to Java objects. This is convenient for the programmers, but has a few problems:
  - Mapping puts performance over head on CPU and memory.
  - Mapping also strongly ties the provider or consumer to the data type.
- JAX-WS provides an option to bypass the mapping layer and work directly with the XML request and response documents.
- With this option, the consumer or provider is responsible for parsing the XML documents using SAX or DOM API. With little extra coding, you can write versatile and high performance code.

Working With Raw XML

By default, a JAX-WS consumer or service provider works with Java classes only. The JAX-WS runtime deals with mapping data from SOAP XML to the Java objects. This paradigm was designed to improve programmers' productivity. Convenience, however, can come at the expense of performance. Converting XML to Java and back, known as marshalling and unmarshalling, has performance overhead. Firstly, CPU is used to copy the data. Secondly, duplication of data wastes memory. None of this is a problem when you are dealing with short SOAP messages. For Web Services that work with large messages, the performance concerns can not be ignored. For example, let us say that the average size of a SOAP message is 1MB and memory used to store the data in Java objects is roughly the same. Total memory used to store request data will be about 2MB. When you have 500 active requests, the total memory will be 1GB. If the data structure is complex, marshalling will waste a lot of CPU time that can otherwise be used by the business logic.

JAX-WS provides an option for the service provider and consumer to bypass its data mapping layer. In this case, the consumer or provider works directly with the SOAP document. This can significantly improve performance.

Performance is not the only benefit of working with raw XML messages. When you use the data mapping service of JAX-WS the provider or consumer becomes strongly typed. That is, the code becomes closely tied to the XML schema in WSDL. It becomes nearly impossible to deal with variability in the schema. You will need to write a separate consumer and provider for each variation in the schema. By working directly with the XML document you can create a generic provider or consumer. For example, let us say that you have a Web Service that accepts orders for products from customer organizations. Different organizations may use different schema for the order object. A generic service provider can parse the request document differently based on the organization.

To work with raw XML documents, a consumer uses the javax.xml.ws.Dispatch object. A service provider works with the javax.xml.ws.Dispatch interface. In both cases the application code is responsible for parsing and constructing XML documents. You can use familiar SAX or DOM API for that purpose. Both are fairly straightforward API. Which means, working with raw XML is not as complex as you may think. We will now discuss this API in more detail.

9.5 Raw XML: Server Side

- The implementation class must be annotated with @WebServiceProvider and not with @WebService.
- The class must implement the Provider<T> interface. Where T is the data type representing a SOAP message. T can be one of:
  - A class that implements Source.
  - SOAPMessage class.
  - DataSource interface.
- The class must implement the T invoke(T) method. Example:

```
@WebServiceProvider(portName = "HelloPort", serviceName =
"HelloService")
public class HelloProvider implements Provider<SOAPMessage>
{

    public SOAPMessage invoke(SOAPMessage req) {
        SOAPMessage res = null;
        //...
        return res;
    }
}
```

Raw XML: Server Side

To implement a generic service provider that works directly with XML documents, the Java class must be annotated with @javax.xml.ws.WebServiceProvider instead of @WebService. A class can either be annotated with @WebServiceProvider or @WebService but not both. The class must also implement the javax.xml.ws.Provider<T> interface. Where T is the data type of the message. It can be one of the following:

- Any type that implements the javax.xml.transform.Source interface. For example, javax.xml.transform.stream.StreamSource.
- The javax.xml.soap.SOAPMessage class. In this case, JAX-WS stores a SOAP document in a SOAPMessage object.
- The javax.activation.DataSource interface. This is used to access a message as an input or output stream.

The Provider<T> interface requires the class to implement the T invoke(T) method.

An example provider class will look like this.

```
@WebServiceProvider(portName = "HelloPort", serviceName = "HelloService",
targetNamespace = "http://www.xyz.com")
public class HelloProvider implements Provider<SOAPMessage> {

    public SOAPMessage invoke(SOAPMessage req) {
        SOAPMessage res = null;
        //...
        return res;
    }
}
```

At execution time, when a SOAP request message arrives for the provider, JAX-WS creates an appropriate request object for it and calls the invoke method passing the request object as input. In the example above, a SOAPMessage object is passed as input to the invoke method.

The invoke method is responsible for constructing the reply document object. This object will be of the same type as the input. The reply object is then returned by the method.

9.6 XML Handling Strategies

- A provider class can choose to work with the entire SOAP message or just the payload (content of the body)

    - This option is set using the @ServiceMode annotation
    - The default is the payload which should be used when possible
- You can parse the input message using SAX or DOM. Both parsers will be able to parse an InputStream object
- You can get the InputStream by representing the message as a StreamSource
    - The class will need to implement Provider<Source>

```
@WebServiceProvider(portName = "HelloPort",
      serviceName = "HelloService",
      targetNamespace = "http://www.xyz.com")
@ServiceMode(value = javax.xml.ws.Service.Mode.MESSAGE)
public class HelloProvider implements Provider<Source> {
```

- If you wish to use DOM, you can also represent the message as a SOAPMessage.
    - The class will need to implement Provider<SOAPMessage>

XML Handling Strategies

A provider may be interested in the contents of the SOAP body only. This contains the business data payload. Alternatively, the provider may be interested in the whole SOAP message. This is known as the service mode. The service mode is set using the @ServiceMode annotation. The possible values can be javax.xml.ws.Service.Mode.MESSAGE and javax.xml.ws.Service.Mode.PAYLOAD. For example:

```
@WebServiceProvider(portName = "HelloPort", serviceName = "HelloService",
targetNamespace = "http://www.xyz.com")
@ServiceMode(value = javax.xml.ws.Service.Mode.MESSAGE)
public class HelloProvider implements Provider<Source> {
```

If the mode is MESSAGE, the invoke method will receive the entire message. It must return the full SOAP reply document. If the mode is PAYLOAD, the method will receive the contents of the <Body> element. It must return the contents of the <Body> element for the reply.

There are mainly two ways you can process a SOAP request:

- Using the SAX API. This request the least CPU and memory overhead. If performance is a concern, you want to go with the SAX option.
- Using the DOM API. DOM puts heavy demand on CPU and memory. It is useful for developing a generic service that works with a widely varying XML schema.

DOM and SAX parsers can parse an input stream. There are two ways to get the input stream for a

SOAP request. The class can implement Provider<Source>. In that case, JAX-WS will represent the request message as a StreamSource object. The class can also implement Provider<DataSource>. We will discuss the first option. In the example below, we parse the request message using a SAX parser.

```
@WebServiceProvider(...)
@ServiceMode(value = javax.xml.ws.Service.Mode.MESSAGE)
public class HelloProvider implements Provider<Source> {
    Logger logger = Logger.getLogger("Default");
    String helloResponse = "<soapenv:Envelope
xmlns:soapenv=\"http://schemas.xmlsoap.org/soap/envelope/\"><soapenv:Body><HelloRes
ponse xmlns=\"http://helloproviderservice.org/types\"><argument>Hello
World</argument></HelloResponse></soapenv:Body></soapenv:Envelope>";

    public Source invoke(Source reqSource) {
        try {
            StreamSource req = (StreamSource) reqSource;
            InputSource inputSource = null;
            if (req.getInputStream() != null) {
                inputSource = new InputSource(req.getInputStream());
            } else if (req.getReader() != null) {
                inputSource = new InputSource(req.getReader());
            }

            SAXParser sp = SAXParserFactory.newInstance().newSAXParser();
            sp.parse(inputSource, new MyHandler());
        } catch (Exception e) {
            logger.log(Level.SEVERE, "Failed", e);
        }
        return new StreamSource(
            new ByteArrayInputStream(helloResponse.getBytes()));
    }
}

class MyHandler extends DefaultHandler {
    public void startElement(String uri, String localName, String name,
        Attributes attributes) throws SAXException {
        logger.info("Starting element: " + localName + "-" + name);
    }
}
```

Instead of SAX, you can parse the input source using DOM.

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Node n = db.parse(inputSource);
```

If you decide to use DOM, you can also implement Provider<SOAPMessage> from the class. The advantage is that, JAX-WS will automatically build the DOM document. It will also give you programmatic access to various parts of the SOAP document, such as the header. For example:

```
public class HelloProvider implements Provider<SOAPMessage> {
    public SOAPMessage invoke(SOAPMessage req) {
        Node n = req.getSOAPBody(); //Get the DOM node for the body.
        Return makeSOAPMessage(...);
    }
}
```

The following example shows how to construct the reply SOAPMessage. Here, the input parmeter is a String that contains the entire SOAP message (starting with the <Envelope> element).

```
private SOAPMessage makeSOAPMessage(String msg) {
    try {
        MessageFactory factory = MessageFactory.newInstance();
        SOAPMessage message = factory.createMessage();
        message.getSOAPPart().setContent(
            (Source) new StreamSource(new StringReader(msg)));
        message.saveChanges();
        return message;
    } catch (Exception e) {
        return null;
    }
}
```

9.7 Raw XML: Client Side

- The client can bypass JAX-WS data mapping.
- It can represent the SOAP request and response as DOM documents or JAXB objects.
- The client uses a Dispatch object to send a SOAP request and obtain the reply.

```
QName portName = new QName("http://www.xyz.com",
       "StockQuotePort");
Dispatch<DOMSource> dispatch = s.createDispatch(portName,
       DOMSource.class, Service.Mode.PAYLOAD);
// Create the request message using DOM
DOMSource request = ...

DOMSource reply = dispatch.invoke(request);
```

Raw XML: Client Side

A client can also bypass the data mapping layer of JAX-WS. In that case, it needs to construct the SOAP request message from scratch and parse the reply document. There are not many business uses of this approach. Generic clients, such as testing tools, use this approach to test any Web Service. We will keep the discuss of this topic very brief.

Like the service provider, the consumer can choose to work with the entire SOAP message or just the payload. It can construct the request document as a DOM tree or a JAXB object. JAX-WS will convert the reply to the same type of object. For example, if you decide to build the request message as a DOM document, JAX-WS will parse the reply and construct a DOM document from it.

A client takes this sequence of steps:

1. Create a javax.xml.ws.Service object.
2. Using the service object create a javax.xml.ws.Dispatch object.
3. Call the invoke() method of the Dispatch object to send a SOAP request. The invoke() method returns the reply object.

We will take a look at an example step by step. First, we create the service object.

```
QName serviceName = new Qname("http://www.xyz.com", "StockQuoteService");
Service s = Service.create(serviceName); //Name of the <service> in WSDL.
```

Then create the Dispatch object. In this example, we choose to work with DOM document at payload level.

```
QName portName = new QName("http://www.xyz.com", "StockQuotePort");
Dispatch<DOMSource> dispatch = s.createDispatch(portName,
    DOMSource.class, Service.Mode.PAYLOAD);
```

Next, we build the DOM document for the payload.

```
DocumentBuilder db = DocumentBuilderFactory.newDocumentBuilder();
Document doc = db.newDocument();
Element root = doc.createElementNS("http://www.xyz.com",
    "getPrice");
root.setNodeValue("IBM");
DOMSource request = new DOMSource(doc);
```

Finally, make the SOAP call.

```
DOMSource reply = dispatch.invoke(request);
```

9.8 Summary

- In this chapter we looked at various advanced programming API of JAX-WS.
- A Web Service can be executed in a lightweight runtime environment by publishing it.
- A service implementation can access meta information about the request and response using the service context.
- Advanced service providers can bypass the data mapping layer of JAX-WS and work directly with raw XML.

    - This option needs to be considered for higher performance and flexibility.

- An advanced client can also bypass the data mapping layer and work directly with XML documents.

Chapter 10 - Building an EJB Based Web Service

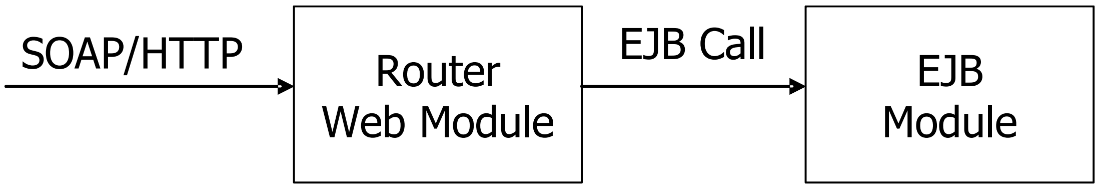
Objectives

Key objectives of this chapter

- Introduction to EJB based web service
- Why use EJB for web service implementation
- A web service without explicit SEI
- A web service with an explicit SEI

10.1 Introduction

- JAX-WS allows us to implement a web service using a stateless session bean in an EJB module.
  - ◊ This way, the EJB can be accessed over IIOP as well as SOAP/HTTP.
- Originally, EJB and EJB containers were designed for IIOP communication only. To interface them with the SOAP/HTTP traffic, the application server needs to provide special support. This can be done in one of these ways:
  - ◊ A special URL context root is created for the EJB module. Any HTTP traffic sent to that context root is assumed to be SOAP requests for the EJB web services in that EJB module. JBoss works this way. See details below.
  - ◊ A router web module is generated by the development tool. This web module routes all SOAP requests to the EJB container where the web service implementation exists. IBM WebSphere works this way.



Special context root

In JBoss, if you implement a web service using a stateless session EJB, a special URL context root is created by the EJB container using the name of the EJB module and the service. For example, if the EJB module is MyEJBModule and the service name is MySvc, all SOAP request is sent to http://<host>:<port>/MyEJBModule/MySvc URL.

10.2 Why Use EJB as Service Implementation?

- There are some advantages to using stateless session bean as web service implementation.
  - You may already have business logic in stateless session beans. As a web service, they can now be exposed to client written in any programming language.
  - EJB web services can use the EJB security annotations like @RolesAllowed to indicate the Java EE security roles authorized to access the entire web service or individual operations
    - You would still need to use WS-Security to send the user/password with the web service request and the security configuration of the server to map security roles to individual users
  - An EJB includes support for distributed transaction. If the web service logic needs to modify multiple resources (such as database, messaging queue or legacy systems like CICS and SAP), EJB will be a very robust choice for web service implementation.
  - If the consumer uses WS-AtomicTransaction, the EJB can inherit the transaction context of the consumer. This way, the client can invoke multiple operations within the same transaction context. This increases the reliability of the work done by the client.

10.3 Implementing EJB Web Service

- An existing stateless session bean can be easily converted to become a web service by the use of @WebService and @WebMethod annotations.

```
@Stateless(name="HelloBean")
@WebService
public class HelloBean implements Hello {
      @WebMethod
      public String sayHello(String name) {
            return "Hello " + name;
      }
}
```

Implementing EJB Web Service

In this example, the HelloBean class implements the EJB remote interface Hello. The Hello interface looks like this.

```
@Remote
public interface Hello {
        public String sayHello(String name);
}
```

This interface has nothing to do with web service. We have specifically exposed the sayHello method as a service operation using the @WebMethod annotation. This makes the operation exposed as a web service. If the remote interface has multiple methods, we can selectively expose some of them as web service by applying the @WebMethod annotation.

10.4 Using a Service Endpoint Interface (SEI)

- You may not wish to expose all methods of the EJB as web service. If that is the case, it may be cleaner to define a separate interface for the web service. That way, the EJB can implement one or more of a remote, local and web service interface.
- Define an SEI using the @WebService and @WebMethod annotation.

```
@WebService
public interface HelloSvc {
        @WebMethod
        public String getMessage();
}
```

- Declare the SEI from the bean class.

```
@Stateless(name="HelloBean")
@WebService(endpointInterface="com.webage.ejb.HelloSvc")
public class HelloBean implements Hello, HelloSvc {
        public String sayHello(String name) {
                return "Hello " + name;
        }
        public String getMessage() {
                return "Hello World";
        }
}
```

Using a Service Endpoint Interface (SEI)

Note that the @WebService annotation is required on the interface and the implementation class.

In this example, the EJB implements a remote interface (Hello) and a web service interface (HelloSvc). They can contain different methods. The endpointInterface attribute of the @WebService annotation points to the web service interface. Only the methods in that interface that are annotated using @WebMethod will be exposed as web service.

In this approach, the bean implementation class becomes cleaner since we do not use @WebMethod annotation there.

10.5 Summary

- Exposing EJBs as web services can provide certain benefits
- The @WebService annotation can be used on an EJB in much the same way as used in a POJO
- It is possible for the web service to expose different methods than those available to EJB clients

Chapter 11 - Error Handling

Objectives
Key objectives of this chapter ▪ Errors and exceptions in web services ▪ Fault ▪ System problems ▪ Business rule violations

11.1 Introduction

- Types of errors that may take place:
  - Business rule violation. Happens in the server side.
  - System problems. Such as database unavailable. Happens in the server side.
  - Communication problem. Can happen in client or server side.
  - Configuration problem. Such as client can not open key database before the SOAP message can be encrypted. This can happen in either side.
- If there is a communication problem, the client runtime will throw **javax.xml.ws.WebServiceException.**
- **WebServiceException** is also thrown to indicate configuration problems.
- A client should catch **WebServiceException** and log the error message.
  - This will help resolve the problem either in the client or the server side.

Introduction

Configuration Problems

Depending on the vendor, if there is a configuration problem in the server, any combination of this can happen:

A detail error log with stacktrace is created in the server. The nature of the error is hidden from the client and an opaque error is thrown in the client.

Just the error message is logged in the server. The same error is thrown in the client side. The client has the opportunity to do a stack trace.

In any case, the client should perform as much logging of RemoteException as possible. This may aid debugging the problem in the server side.

11.2 Fault

- The other two types of problems (business rule violation and system issues) occur in the server side and should be reported to the client.
- Error conditions are sent back in the reply using the <soap:Fault> element.
- The Fault element appears inside the <soap:Body> element only once.
- The Fault element can have these children elements:
  - faultcode: An error identifier
  - faultstring: A short error message
  - faultactor: A string identifying the source
  - detail: More details

    Can contain arbitrary user defined child elements (such as the AppErrorDetails element shown below).

Fault

SOAP Example

```
<soapenv:Envelope xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Header/>
 <soapenv:Body>
   <soapenv:Fault>
   <faultcode>FAULT CODE GOES HERE</faultcode>
   <faultstring>SHORT DESCRIPTION</faultstring>
   <detail>
     <md:AppErrorDetails xmlns:md="http://mycom.com">
       <description>XYZ</description>
       <code>ABC</code>
     </md:AppErrorDetails>
   </detail>
  </soapenv:Fault>
 </soapenv:Body>
```

```
</soapenv:Envelope>
```

11.3 Designing Faults

- If an operation is expected report error conditions to the client, we must declare a fault for the operation in the WSDL document.
- First define an XML element that will contain custom error information.
- Declare a message with that element as the type.
- Declare a fault for the operation using that message.
- This element goes inside the <detail> element in the SOAP reply message.

Designing Faults

Define the data type of the fault in the WSDL schema section.

```
<xsd:element name="AppErrorDetails">
   <xsd:complexType>
         <xsd:sequence>
                  <xsd:element name="description" type="xsd:string"/>
                  <xsd:element name="code" type="xsd:string"/>
         </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

Declare a message.

```
<wsdl:message name="AppErrorDetailsMsg">
    <wsdl:part element="md:AppErrorDetails" name="d" />
</wsdl:message>
```

Specify the fault.

```
<wsdl:portType name="MovieServiceAbstract">
    <wsdl:operation name="addMovie">
        <wsdl:input message="m:addMovieRequest" />
        <wsdl:output message="m:addMovieResponse" />
        <wsdl:fault name="f" message="m:AppErrorDetailsMsg"/>
    </wsdl:operation>
</wsdl:portType>
```

You also need to specify the fault in the binding element.

11.4 System Problems

- If there is an unexpected and temporary system problem at the server side, the client does not need to know the details.
  - The server code should log or notify the problem and then throw RuntimeException.
- The server runtime will convert this exception to a SOAP fault.
  - There is no need to design and declare a fault for the operation.
  - This makes development easy since almost any method can experience system issues at some point. Still, we don't have to declare faults for them.
  - You can supply basic error message to the constructor of the RuntimeException class. A good server runtime will store this error in the faultstring element. See example below.
- The client runtime will read the SOAP Fault element and throw a RuntimeException.
  - A good client runtime will give access to the error message in the faultstring element.

System Problems

Example server code.

```
try {
  //Write business logic here
} catch (Exception e) {
  //Unexpected error
  //Log for your own sake
  e.printStackTrace();
  //Report to client
  throw new RuntimeException(e.getMessage());
}
```

Example SOAP snippet showing how RuntimeException is converted to fault.

```
<soapenv:Fault>
  <faultcode>soapenv:Server.generalException</faultcode>
  <faultstring><![CDATA[java.lang.RuntimeException: Write failed. Disk out of
space.]]></faultstring>
</soapenv:Fault>
```

11.5 Business Rule Violation

- For top-down development, vendor tool will generate a Java class for the fault message element type.
  - The class will extend java.lang.Exception.
  - The operation will be declared to throw this class.
- For bottom-up development:
  - Write the custom exception class.
  - Write the business logic method. Declare to throw the custom exception class.
  - Generate WSDL. Vendor tool will automatically define an XML element and message in the WSDL file. The operation will have a fault.
- At runtime, the implementation method can throw the custom exception.
  - The server runtime will serialize the object as XML element within the <detail> element of the Fault.
  - The client runtime, will deserialize and construct the custom exception object and throw it.
  - The client can catch that exception and report custom error details.

Business Rule Violation

Server Code

```
public void addMovie(…) throws AppErrorDetails {
          //…

          AppErrorDetails d = new AppErrorDetails();
          d.setDescription(…);
          d.setCode(…);
          throw d;
}
```

Client code

```
try {
          sei.addMovie(…);
} catch (AppErrorDetails d) {
          System.out.println(d.getDescription());
}
```

11.6 Summary

- Errors and exceptions in web services
- Fault
- System problems
- Business rule violations

Chapter 12 - Managed Web Service Client Binding

Objectives

Key objectives of this chapter

- Understand the weakness of using a packaged WSDL for endpoint address
- Learn how to write service clients that can be linked administratively to service endpoints
- See some of the administrative tasks that might be performed with web service applications

12.1 Using a Packaged WSDL

- Typically the code generated for a client to interact with a service will look to a WSDL file packaged with the application for the location of the service
  - This "endpoint location binding" is required to communicate with the service
- The problem with this is that as the client application is deployed to different environments it will likely be linked to different locations of the service
  - You aren't going to link the client in a testing environment to the production location of the service
- Adding a different version of the WSDL with an updated location would change the EAR file before deploying it in the next environment
  - This may violate change management standards

Using a Packaged WSDL

If the schema of data used by a service or the operations on a service change a new WSDL should be used in the client application to reflect those changes. Any generated code should also be regenerated based on the new WSDL to reflect those changes. These kinds of changes to the structure of the service can significantly impact a client.

The scenario we are trying to avoid is modifying the WSDL when the only change is the location of the service.

12.2 Managed Web Service Clients

- A better solution is to code the client application to allow for modifying the endpoint location binding administratively
  - The application can be configured as appropriate in different environments without modifying any aspect of the application EAR file
- By using the @WebServiceRef annotation a service client can take advantage of the dependency injection of a reference to the web service at run-time
  - The server will create an object at run-time and "inject" it into a field with this annotation so the client code does not need to call the constructor

```
public class MyServlet extends HttpServlet {
      @WebServiceRef
      private OrderService service;

      ... // in some method
      ~~service = new OrderService();~~  // not needed
      Order order = service.getOrderPort();
      order.placeOrder(...);
```

- The ability to inject web service references in this way is only available to "managed" client components which includes Servlets, JSF Managed Beans, and EJBs

Managed Web Service Clients

The constructors of the generated Service class for the client typically look for the local WSDL file which is why using the constructors directly was not very flexible. By using "injection", the client code can allow the server to create the service object, modify it's properties, and then supply it to the client code. Although at first glance the code above might look like it would throw NullPointerException the "injection" will provide an object at run-time and avoid this.

A standalone Java application will only have dependency injection performed for it if it runs in the "Application Client" container defined by Java Enterprise. This is software typically provided by a Java EE application server vendor and installed on the client machine.

A standalone Java application running only in the standard Java SE JRE (Java Run-time Environment) will not have dependency injection available through the use of this annotation.

12.3 Injecting the Service Port Directly

- When the @WebServiceRef annotation is used on a field that extends the javax.xml.ws.Service class you still need to get the "Port" before calling the operations on the service

```
@WebServiceRef
private OrderService service;

... // in some method
Order order = service.getOrderPort();
order.placeOrder(...);
```

- If the 'value' attribute is added to the annotation to point to the Service class you can use the annotation on a field of the Port type instead

```
@WebServiceRef(value=OrderService.class)
private Order order;

... // in some method
~~Order order = service.getOrderPort();~~ // not needed
order.placeOrder(...);
```

12.4 Web Service Administration Tasks

- Package consumer and provider applications as web modules or EJB modules in Java EE EAR files.
- Install the provider and consumer enterprise applications.
- Create JDBC and JMS resources as required by the provider.
- Attach web service policies to the consumer and provider to enable advanced features, such as:
  - Security
  - Transaction
  - Reliable messaging
- Configure security encryption keys for provider and consumer.
- Configure service endpoint URL in provider and consumer.

Web Service Administration Tasks

Developers are responsible for developing web service providers and consumers. In Java, they do that by following the JAX-WS specification. JAX-WS extends the functionality of Java EE. For example, a service provider is created in a web module or EJB module. This seamless integration with Java EE makes it easy for the programmers to learn and use JAX-WS. The same applies for the administrators.

First task for the administrators is to package a web service provider and consumer. They are packaged as per the Java EE packaging standard. That is, you continue to create web module, EJB module and enterprise archive files (EAR).

Next, you need to install the consumer or provider. Once again, there is very little that is new here. You install enterprise applications from EAR files as usual. A web service provider may use JDBC data sources or JMS resources. They need to be created.

In this chapter we will not repeat the steps to package and install applications or JDBC and JMS resource creation.

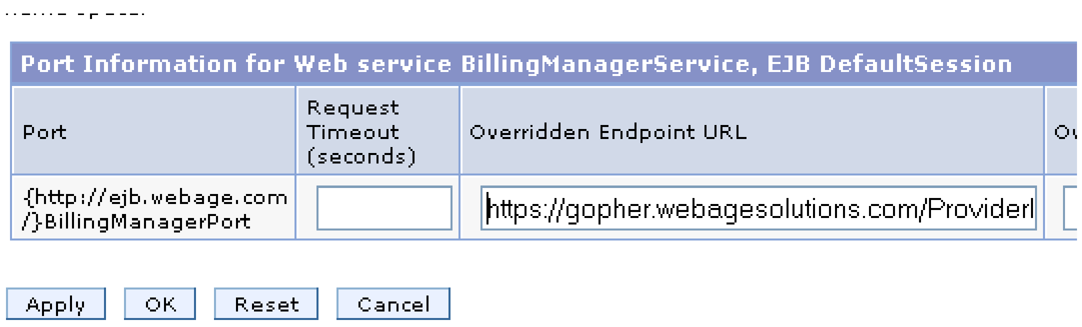
After a web service application is installed, a few web service specific administration tasks may have to be performed. Advanced web service providers and consumers use functionality like transaction, security and reliable messaging. To enable these features, you need to attach web service policies to these applications.

If the web service uses security then encryption keys need to be configured.

The endpoint information coded in the WSDL file may not be suitable for production environment. This applies to both consumer and provider. Typically, the development time URL is used. We need to configure the provider so that it exports the correct URL in the WSDL. If the consumer is using a local copy of the WSDL, the service endpoint URL needs to be properly configured for a consumer so that it can connect to the provider.

12.5 Configure Endpoint URL in Consumer Side

- By default, developers configure a consumer to connect to the provider in development environment.
- The endpoint URL must be changed in the consumer side so that it connects to the provider in production.
- This is accomplished by editing the port information in the consumer side.
- This option to configure the endpoint is NOT presented during application installation and must be configured after the application is installed



Port Information for Web service BillingManagerService, EJB DefaultSession

Port	Request Timeout (seconds)	Overridden Endpoint URL	Ov
{http://ejb.webage.com/}BillingManagerPort		https://gopher.webagesolutions.com/ProviderI	

- This is accessed from the '**Web services client bindings**' link from the module of the client and editing the '**Port Information**'

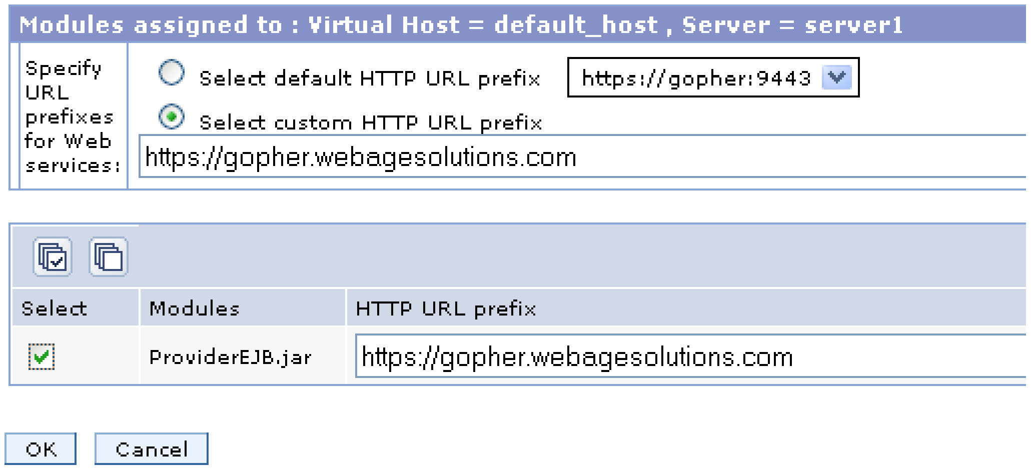
Configure Endpoint URL in Consumer Side

A client typically uses a local copy of the WSDL file where the endpoint address of the server is coded. By default, this address is set for development machines and will not work in production. Follow these steps to change the URL in the consumer side.

Open the properties page of the application where the consumer is located. Click **Manage Modules**. Click the EJB or web module where the client is located. Click **Web services client bindings**. Under **Port Information** click **Edit**. Enter the correct endpoint URL for production as shown above. Save changes.

12.6 Configure Endpoint URL in Provider Side

- In production, a consumer will connect to a provider through a public web site.
- The endpoint URL in the WSDL published by the provider may not reflect that hostname and port number.
- The correct host name and port number is entered by changing the URL prefix of the provider.
- This option to configure the endpoint is NOT presented during application installation and must be configured after the application is installed



Modules assigned to : Virtual Host = default_host , Server = server1

Specify URL prefixes for Web services:	○ Select default HTTP URL prefix https://gopher:9443
	[✓ Select custom HTTP URL prefix] https://gopher.webagesolutions.com

Select	Modules	HTTP URL prefix
✓	ProviderEJB.jar	https://gopher.webagesolutions.com

- This is set from the '**Provide HTTP endpoint URL information**' link from the application of the service provider

Configure Endpoint URL in Provider Side

By default, the provider will expose a WSDL that contains an endpoint URL that may not be fit for production. The URL has the local host name of the WebSphere machine and the port number used by the web container. In most cases, a client will connect through a web server and has to use an external host name and port 80 or 443. The steps discussed below will fix the host name and port number in the endpoint URL of the exported WSDL file.

We will set the URL prefix where the correct host name and port number will be mentioned. Open the properties page for the enterprise application of the service provider. Click **Provide HTTP endpoint URL information** under **Web Services Properties**. Enter a custom URL prefix as shown below.

Save changes. The server has to be restarted for the changes to take effect. After that the correct endpoint URL will show up in WSDL. For example:

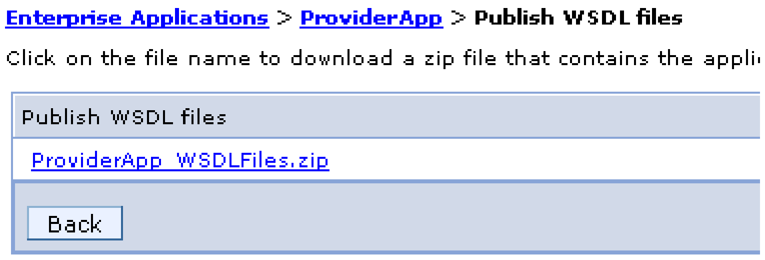
<soap:address location="**https://gopher.webagesolutions.com**/ProviderEJB_HTTPRouter/BillingManagerService"/>

In the InfoCenter you may see reference to a custom JVM property called com.ibm.ws.webservices.enableHTTPPrefix. The InfoCenter incorrectly states that this property needs to be created and set to true for WebSphere 7.0. Although this property was required for previous WebSphere versions to be able to override the endpoint URL prefix it is not required for WebSphere 7.0. In fact, you could even configure this property with a value of 'false' with no effect.

12.7 Publishing WSDL File

- You can export the WSDL for a service provider.

    - ◊ This will be a zip file with relevant documents like WSDL and XML schemas
- This WSDL is distributed to teams and organizations that will develop consumers for the web service.
- If you have corrected the endpoint URL in the provider side, it will be reflected in the exported WSDL.


Enterprise Applications > ProviderApp > Publish WSDL files

Click on the file name to download a zip file that contains the appli

Publish WSDL files

ProviderApp_WSDLFiles.zip

- Changing the endpoint address that appears in this "published" WSDL is the only purpose of setting the endpoint URL prefix shown previously
    - ◊ The ability of the server to listen for requests for the service is ultimately determined by the host alias of the 'Virtual Host' the application is mapped to just like a regular web application

Publishing WSDL File

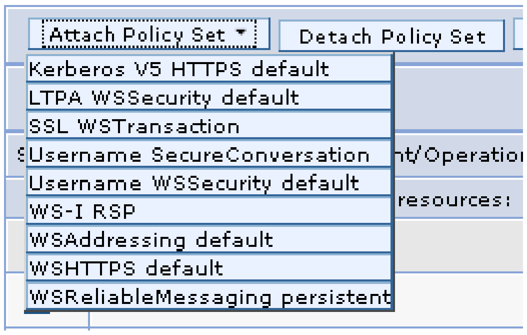
The WSDL file for a provider is published and distributed to the consumer. A consumer uses it to connect to the provider. To export the WSDL, open the application's properties page in admin console. Click **Publish WSDL files** under **Web Services Properties**.

Download the zip file that contains the WSDL file. This WSDL file will have the correct endpoint URL as modified using the steps described in the previous section. Distribute the WSDL file to the development team responsible for consumer development.

12.8 Working with Policy Sets

- Advanced styles of web services interaction, such as transaction, security and reliable message is enabled by attaching policy sets to the service provider and consumer.
- A policy set is a collection of policies, such as security and transaction.
- Typically, the same policy set is attached to the provider and consumer.

- Custom policy sets can be imported into a profile and then attached to consumers and provider.



Working with Policy Sets

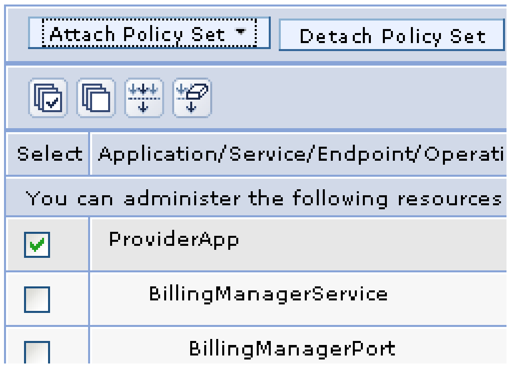
In this lecture, policy sets are not covered in detail but simply presented as the way many advanced web service properties are set.

In WebSphere, advanced styles of web services interaction, such as transaction, security and reliable message is enabled by attaching policy sets to the service provider and consumer. A policy set is a collection of policies. For example, if you need security and transaction, you will create a policy set that has these two policies enabled.

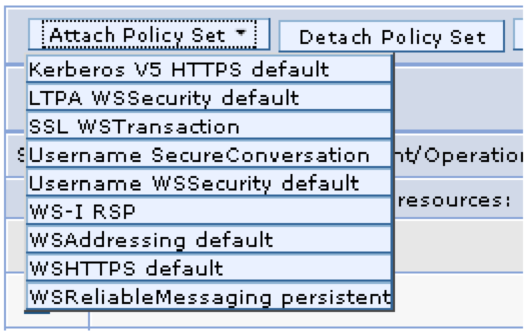
Typically, the same policy set is attached to the provider and consumer. Otherwise, they will not communicate very well. For example, if security is enabled in the provider, it must also be enabled in the exact same manner in the consumer.

By default a profile comes with a set of policy sets predefined and imported into it. In some cases, additional policy sets need to be created and imported. Details of this is outside the scope of this class. We will learn how to attach existing policy sets.

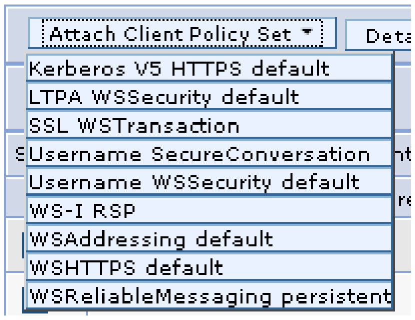
To attach a policy set to a provider, open the properties page of the enterprise application. Click **Service provider policy sets and bindings** under **Web Services Properties**. Select the check-box for the application or the service.



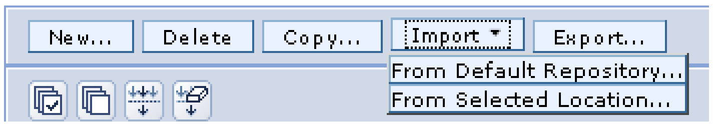
Then click the **Attach Policy Set** button and select the available policy set.



To do the same for a consumer, open the properties page of the application. Click **Service client policy sets and bindings** under **Web Services Properties**. Select the check box for the whole application or a specific client. Then click **Attach Client Policy Sets** and select a policy set.


Attach Client Policy Set
Kerberos V5 HTTPS default
LTPA WSSecurity default
SSL WSTransaction
Username SecureConversation
Username WSSecurity default
WS-I RSP
WSAddressing default
WSHTTPS default
WSReliableMessaging persistent

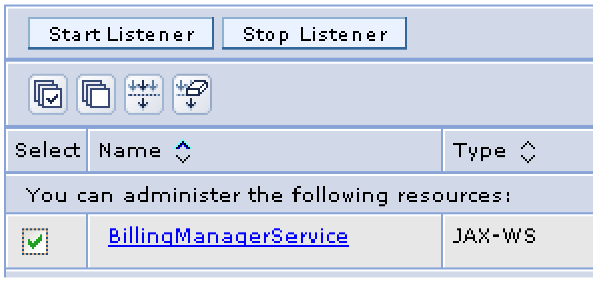
Custom policy sets are packaged in ZIP files. To import such a policy set, in admin console, click **Services** > **Policy sets** > **Application policy sets**. Then click **Import** > **From Selected Location** as shown below.



After it has been imported, a policy set can be attached to a consumer and provider.

12.9 Stopping a Service Listener

- You can stop a service from accepting requests without stopping the application or the server.
- In admin console, click **Services > Service providers**.
- Select the check-box for a provider.
- Click **Stop Listener**.



Select	Name	Type
You can administer the following resources:		
✓	BillingManagerService	JAX-WS

12.10 Summary

- Writing web service client code that can be configured administratively at run-time is preferred to updating WSDL files packaged with the application
- The @WebServiceRef annotation can be used in certain application components to "inject" a service reference that is configured administratively
- WebSphere provides support for many administrative tasks related to web service configuration

Chapter 13 - Web Services Security (WS-Security)

Objectives

Key objectives of this chapter

- The basics of how Public Key Infrastructure (PKI) works.
- What WS-Security attempts to achieve.
- How message encryption works.
- How message signature works.
- How authentication works.

13.1 The Challenges

- Web security is essentially enforced by SSL and HTTPS.
  - This is mostly sufficient when the consumer of the document is a human being. The user can use her judgment to validate the document.
  - In SOA, the consumer is another application. Validation will require digital signature. Otherwise, the SOAP message is open to man in the middle attack.
  - Also, messages may be temporarily stored and then routed to multiple destinations at a later time. SSL can not support such use cases.
- XML is fairly descriptive and size of SOAP documents are already fairly large. Encryption can add additional performance problem.
  - We will see how we can add security without encrypting the whole message.

13.2 Public Key Infrastructure (PKI)

- Party A has two keys. One private and the other public.
  - The public key is distributed widely to almost anyone.
- **Confidentiality:** Before party B sends a message to A, it encrypts the message using party A's public key.
  - Now, only party A can decrypt the message with the private key. This

creates a "for your eyes only" situation and achieves confidentiality.

- **Source identity assertion**: When party A sends a message to anyone, the message is encrypted using party A's private key.
  - Anyone can decrypt it using party A's public key. This ensures that the message indeed came from party A.

13.3 Digital Signature

- The purpose is to make sure that a message can not be altered after it has been sent. Useful for:
  - **Non-repudiation:** Sender can not claim that he did not send the message.
  - **Integrity:** Malicious parties can not alter a message to their benefit.
- The sender:
  - Creates a hash of the message. A hash uniquely summarizes a message. Two different messages can not have the same hash value.
  - Encrypts the hash using her private key. This is called the signature. Then the message and the signature (encrypted hash) is sent.
- The receiver:
  - Decrypts the hash using the sender's public key. This ensures the identity of the sender.
  - Hashes the message. Compares this hash with the hash decrypted in the previous step. If they match, the message has not been altered.

Digital Signature

Digital signature builds on the identity assertion aspects of PKI. Instead of encrypting the whole message (that will cause the message size to grow), only a digest of the message is encrypted. This encrypted digest is called the signature. Now, we achieve identity assertion without creating a large encrypted message. The message remains in plain text.

13.4 Certificates

- In PKI, the public key should be widely circulated.

- A public key itself provides no guarantee that it belongs to the person that it claims to be.
  - Public key is just a series of numbers and has no personal information. Even if it did have such information, can you trust them?
  - Faked public key can cause serious security breach. For example, party C can send out fake public keys pretending to be party B (by spoofing B's e-mail address for example). Party A encrypts a message unknowingly using the fake public key and hence C gets to read it (while A thought it was for B's eyes only).
- To solve this problem, a public key is never distributed on its own. It is included inside a certificate.
  - The certificate also includes the party's personal information, such as name, employer etc.
- Certificates are relatively more trustworthy than just the public key on its own.

13.5 Overview of Web Services Security

- The OASIS group currently owns the Web Services Security specification.
  - www.oasis-open.org
- Main goal of the specifications is to secure SOAP messages. Specifically, the following areas are covered:
  - **Integrity:** Ensures that a message came from the right source and was not altered on the way. This is achieved by signing the message before sending it.
  - **Confidentiality:** The message must be opaque to any party other than the sender and receiver. This usually involves encrypting the message body with the receiver's public key.
  - **Security identifier token:** Identifies the sender principal. For example, user ID and password combination.
- Specification allows for securing only the sensitive portions of a SOAP message to avoid increasing the message size.

Overview of Web Services Security

Web page for Web Services Security:

http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss

13.6 SOAP Message Security

- OASIS specification adds a <wsse:Security> element as a child of the SOAP <Header> element.
  - All configuration required to secure the message (integrity etc.) are stored within this element.
- Various other elements are defined by the specification to store the configuration data within <wsse:Security>.
  - See a list of namespaces and recommended prefixes below.
- Needless to say, you need a high level programming model and support from the runtime to format the SOAP message properly.
  - Doing this from your own code can be time consuming and expensive.
  - Make sure that your client and server runtime supports SOAP security.
  - Security specification is changing fast. Both sides must adhere to compatible specification levels.

SOAP Message Security

Example SOAP Message

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" …>
<soapenv:Header>
<wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-secext-1.0.xsd" soapenv:mustUnderstand="1">
…
</wsse:Security>
</soapenv:Header>
<soapenv:Body>
…
</soapenv:Body>
</soapenv:Envelope>
```

Other namespaces used

```
ds - http://www.w3.org/2000/09/xmldsig#
```

```
S11 - http://schemas.xmlsoap.org/soap/envelope/
S12 - http://www.w3.org/2003/05/soap-envelope
wsse - http://docs.oasis-open.org/wss/2004/01/oasis-
200401-wss-wssecurity-secext-1.0.xsd
wsu - http://docs.oasis-open.org/wss/2004/01/oasis-
200401-wss-wssecurity-utility-1.0.xsd
xenc – http://www.w3.org/2001/04/xmlenc#
```

13.7 Message Integrity

- Recall, message integrity is ensured using digital signature. The message must include:
  - The signature: A digest of the message encrypted using sender's private key.
  - The public key of the sender inside of the sender's certificate. Otherwise, the receiver may not have it.
  - Information about algorithms used to digest the message and encrypt the digest.
- The <ds:SignatureValue> element holds the signature.
  - The <ds:DigestValue> element holds the unencrypted digest of the message for added verification.
- The <ds:KeyInfo> element holds the certificate.
  - You can also include the certificate in a <wsse:BinarySecurityToken> element. In that case, <ds:KeyInfo> will refer to the token's ID.
- Elements such as <ds:SignatureMethod> contain unique identifier of the algorithm used.
- The actual business data within the <soapenv:Body> is sent in open text as usual.
- The server should also similarly sign the response message. This time, the server's certificate is sent in the reply.

Message Integrity

Example Signed Message

```
<wsse:Security …>
```

```
<wsse:BinarySecurityToken wsu:Id="CERT_ID_123">SENDER
CERTIFICATE</wsse:BinarySecurityToken>
<ds:Signature...>
<ds:SignedInfo>
...
<ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
…
<ds:DigestValue>HywTMTJb2kCKI069cq1DuqlvK8o=</ds:DigestValue>
</ds:SignedInfo>
<ds:SignatureValue>kJR09ZvbW23fOKfdDbIreqsWsxU/
KauzBa04blduAfZ2G9ON3a1b2QwF5acZcLw0eZjfaa+azLUNyOYB0C/
A54TdVgJUOs4ZMyaEPwYVips3Wuln9g9H9HRtIgwcfihkCGMBQDMpn4fzQVGVEj3umOgxjLKi9NP7iXiwYY
SA3AQ=</ds:SignatureValue>
<ds:KeyInfo>
<wsse:SecurityTokenReference>
<wsse:Reference URI="#CERT_ID_123"
ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-
profile-1.0#X509"/>
</wsse:SecurityTokenReference>
</ds:KeyInfo>
</ds:Signature>
</wsse:Security>
```

13.8 Message Confidentiality

- Recall, the sender needs to encrypt the message using the receiver's public key.
    - This means, the public key of the receiver must be stored in the sender's side.
    - Optionally, the sender may generate a random key, encrypt the message using the key, encrypt the random key with receiver's public key and add it to the message. This is called symmetric key encryption which is faster than asymmetric key encryption.
- As an application server may maintain several keys in a database, the sender needs to supply an identifier of the (public) key used to encrypt.
    - This will allow the receiver to use the appropriate private key to decrypt.
- The specification allows you to encrypt:
    - The header
    - Any element in the body

      An XPath expression is used to specify the element to be encrypted.

  - The entire body

Message Confidentiality

The specification is very flexible. You can configure the system so that only one of request or response can be encrypted. If only the response needs to be encrypted, the consumer can send its public key as a part of the request. The service provider can send its public key as a part of the response. In all other cases, a party needs to already have the other's public key.

13.9 Message Confidentiality

- The sender specifies the identity of the receiver's public key using the <ds:KeyInfo> element.
  - Either a friendly name of the key is specified or a key identifier is specified using <wsse:KeyIdentifier>. For example, a key identifier can be a hash of the distinguished name of the key. Either way, this helps the receiver use the right private key to decrypt.
  - If a random key is used, then it is encrypted using receiver's public key and stored in the <CipherValue> element.
- Within the <soap:Body> encrypted element data is stored in <CipherValue> elements.
- The specification does not cover how to specify the key file location and key name to be used at runtime.
  - Vendor must provide custom deployment descriptor for that.

Message Confidentiality

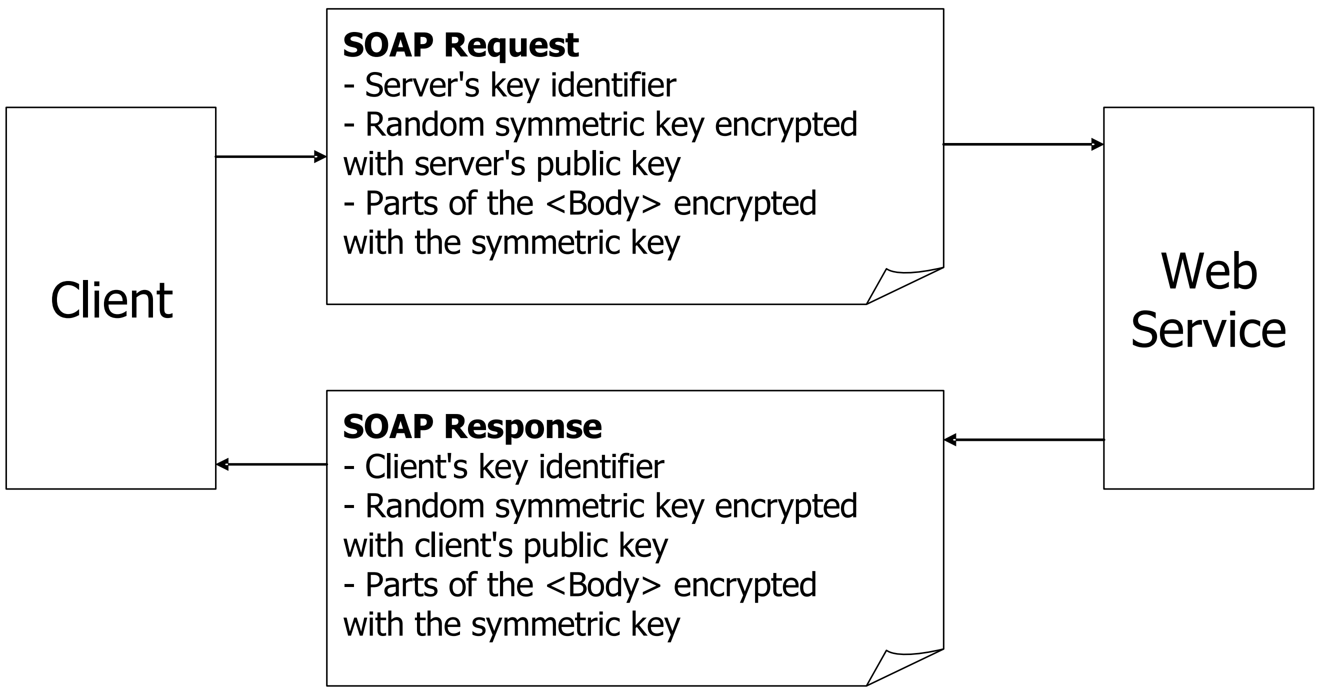
Specifying the key used to encrypt

```
<EncryptedKey …>
<ds:KeyInfo …>
<wsse:SecurityTokenReference>
<wsse:KeyIdentifier …>/62wXObED7z6c1yX7QkvN1thQdY=</wsse:KeyIdentifier>
</wsse:SecurityTokenReference>
</ds:KeyInfo>
<CipherData>
<CipherValue>WvTmsOVX…Hu+KA=</CipherValue>
</CipherData>
</EncryptedKey>
```

Encrypted Element in the Body

```
<soapenv:Body>
<p384:addMovie xmlns:p384="http://www.mycom.com/movie">
<movie rating="sss" title="sss">
<description>ssss</description>
<EncryptedData …>
<EncryptionMethod Algorithm="…"/>
<CipherData>
<CipherValue>7BM/a4M9ILFpqM1QUEfeTSu0=</CipherValue>
</CipherData>
</EncryptedData>
<release_date xsi:nil="true"/>
</movie>
</p384:addMovie>
</soapenv:Body>
```

13.10 Symmetric Encryption Example


Client
SOAP Request
- Server's key identifier
- Random symmetric key encrypted with server's public key
- Parts of the <Body> encrypted with the symmetric key
Web Service
SOAP Response
- Client's key identifier
- Random symmetric key encrypted with client's public key
- Parts of the <Body> encrypted with the symmetric key

13.11 Authentication Using Identity Token

- Access to the operations of a web service can be restricted to a few well known users.
- SOAP security specification allows the client to send his or her identity

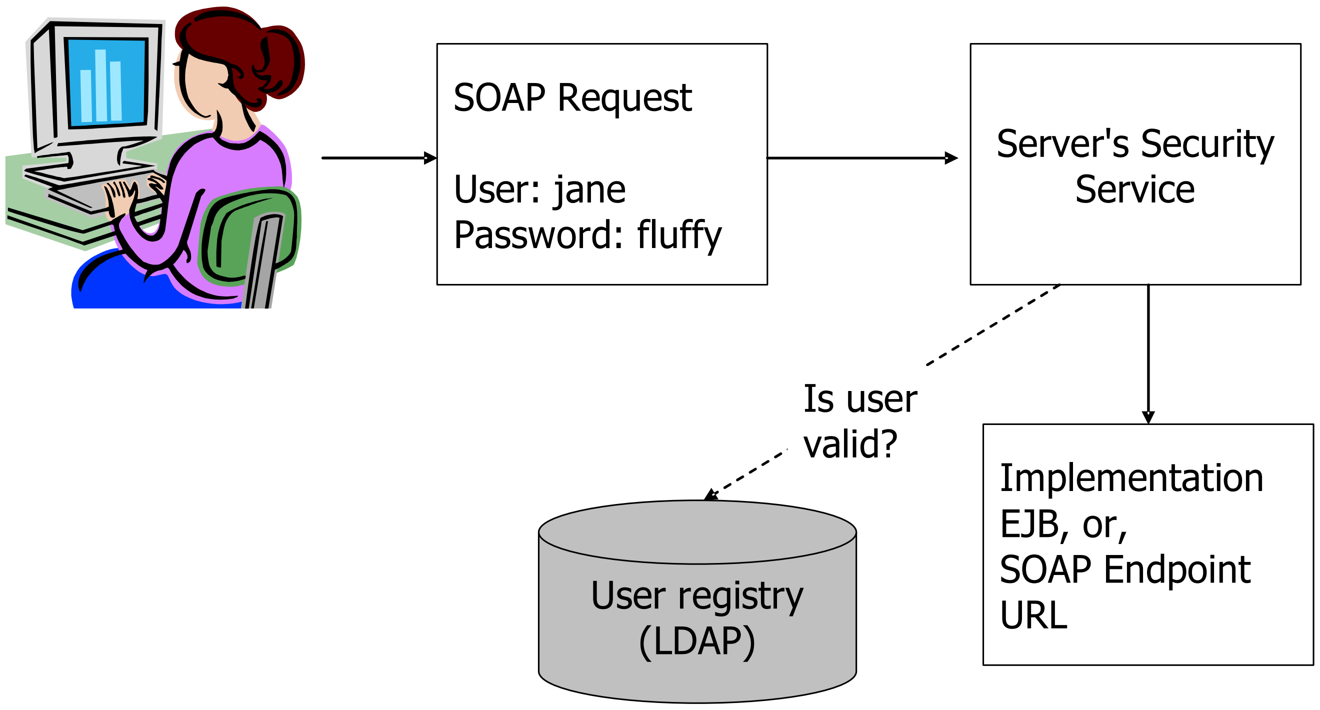
credentials in one of two ways:
  - Simple user ID and password
  - X.509 Certificate
- The server side uses this information to:
  - Authenticate: Make sure that the credentials are valid.
  - Authorize: Make sure that the user has access to the Web service.
- Application servers will most likely use existing security framework to perform authorization and authentication.
  - For example, the user must belong to the same user registry used by the server to protect Servlets and EJBs.
- Caveat: There is no clear way to implement role based security for the services. Specification simply covers authentication. In other words, anyone with a valid credential can access any service.
  - This can be a problem if you are running multiple services out of the same server runtime with different security requirements.

Authentication Using Identity Token

JEE uses a role based security model. Roles roughly translate to job roles. For example, Manager or Order Entry Clerk. Using JEE deployment descriptors or annotations, you can specify which roles can access what EJB or Web URLs. The roles are then mapped to actual users or user groups when the application is deployed to a server.

SOAP security specification does not cover how you specify the access control rules for the services. JAX-WS and JSR-109 do not cover this either. As long as the user can be authenticated (credentials are valid), the user can access any web service in the server.

13.12 Authentication



13.13 Authentication

- Within the <wsse:Security> element, <wsse:UsernameToken> element is used to specify user ID and password.
  - <wsse:Username> stores user ID.
  - <wsse:Password> stores password.
- Credentials are in plain text. Can selectively encrypt <wsse:UsernameToken> element using WS-Security.
- In a Web-based application, users manually enter user ID and password to login.
  - In Web services, client makes a call without any human interaction. The user ID and password must be configured somewhere.
- The user must belong to the server's user registry (such as an LDAP server).
- Usually, a user account will be setup for the client by the server side IT

team. The user information will be communicated to the IT team of the client side.

Authentication

Example SOAP Message

```
 <soapenv:Header>
 <wsse:Security…>
         <wsse:UsernameToken>
            <wsse:Username>jane</wsse:Username>
            <wsse:Password…>fluffy</wsse:Password>
         </wsse:UsernameToken>
</wsse:Security>
</soapenv:Header>
```

13.14 Transport Level Security

- Secures the entire client server communication (every byte is encrypted).
- Most common mechanism Secure Socket Layer (SSL).
  - In this case, SOAP requests are sent using HTTPS protocol.
  - SSL 3.1 is now renamed to TLS.
- SSL provides data integrity and confidentiality.
- Advantages of SSL
  - Well understood, relatively easy to setup.
  - Web Services Security specification only works when the client is running in a J2EE container. For stand alone Java clients or clients without a runtime unable to support Web Services security, use SSL.
- Disadvantages:
  - Applies to HTTP protocol only. If you are sending SOAP message over a messaging protocol such as JMS or SMTP, SSL will not apply.
  - No digest is sent as a part of the message. As a result, it can not be stored for non-repudiation purposes. Neither can the message be forwarded to another party and still maintain authenticity of the source.

13.15 Audit Tracking

- In SOA, messages arrive from outside the organization or department that cause certain business tasks to be carried out. In case of a dispute, we should be able to prove at a later time that such a message had indeed been sent by the sender. This is called non-repudiation.
- Non-repudiation is implemented by:
  - Capturing the original request message in a durable storage. This is called audit logging.
  - Proving that the message came from the sender. This is done using message integrity feature already discussed.
  - Proving that the message in the audit log is exactly same as what was sent. This is also solved using the integrity feature.

13.16 Audit Tracking

- Essentially, we need a mechanism for:
  - Storing important messages (both request and response).
  - Viewing the messages. Due to the immense number of such messages, a query facility should be available.
  - Expiring very old messages and removing them from storage.
- In the beginning, you can develop a custom solution for these needs. Your ESB may have an audit logging component that you can insert in the request processing workflow (business process or mediation flow) to automatically store requests and responses.
- Eventually, you should look for an off the shelf solution to manage audit trail. For example, Tivoli Compliance Insight Manager (http://www-01.ibm.com/software/tivoli/products/compliance-insight-mgr).

Audit Tracking

Your SOA governance should clearly spell out the audit logging policy. For example:

1. A set of criteria to determine which messages are logged. For example, all messages coming from outside the organization should be logged.

2. How long will the message be kept?
3. What happens when there is a dispute? How will the message be retrieved and its integrity proven?

13.17 Identity Assertion Using SAML

- One problem with the identity token requirement of WS-Security is that, a user may have different identity and password in different systems. It will be easier if the consumer could send a generic token that can later be validated by the provider by some means.
  - This is achieved through the identity assertion mechanism.
- In this mechanism, the consumer submits its credentials to an identity provider. This provider validates the credentials and sends back an assertion statement.
  - This statement asserts that the user is who she says she is.
- The consumer then sends this assertion statement to the service provider as a part of the SOAP request.
- The provider can use "back-channel" or a separate request to work with an identity provider to make sure that the assertion statement is valid. Multiple identity providers can work with each other. For example, organization B's identity provider can validate an assertion statement issued by organization A's identity provider.
- Security Assertion Markup Language (SAML) is a specification managed by OASIS that deals with how one can obtain an assertion statement and send it as a part of a SOAP request.

Identity Assertion Using SAML

The assertion statement typically contains:

1. User's ID.
2. Any other attribute about the user, such as age and country of residence.

13.18 SAML SOAP Example

```
<soap:Header><wsse:Security>
<saml:Assertion
  xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  AssertionID="_26de37af534864ef42c110ec23097ebc"
IssueInstant="2007-01-24T01:31:09.327Z"
  Issuer="www.webagesolutions.com">
  <saml:AuthenticationStatement
    AuthenticationInstant="2006-02-26T01:11:18.537Z"
    AuthenticationMethod=
       "urn:oasis:names:tc:SAML:1.0:am:password">
    <saml:Subject>
      <saml:NameIdentifier>
        janedoe
      </saml:NameIdentifier>
    </saml:Subject>
    <saml:SubjectLocality IPAddress="192.168.1.102"/>
  </saml:AuthenticationStatement>
  <saml:Attribute
  xmlns:x500="urn:oasis:names:tc:SAML:2.0:profiles:attribute:X500"
  NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri"
  Name="urn:oid:2.5.4.42" FriendlyName="givenName">
  <saml:AttributeValue xsi:type="xsd:string"
x500:Encoding="LDAP">Jane</saml:AttributeValue>
</saml:Attribute>
</saml:Assertion>
</wsse:Security></soap:Header>
```

SAML SOAP Example

This example shows how to send the assertion statement as a part of a SOAP request. The AssertionID uniquely identifies a request for assertion made by the consumer. The identity provider replied back with the assertion statement. the same statement is then inserted in the <wsse:Security> section of the SOAP request as we see above. As a part of the assertion, we see the <saml:Subject> element. This clearly states who the user is. In addition, we see <saml:Attribute> elements. They are optional. They specify additional properties of the user, such as the given name in this example.

Chapter 14 - REST Services

Objectives

Key objectives of this chapter

- Identify the core elements of RESTful services
- Describe the importance of URI resources within REST
- Explain the distinctions between REST and SOAP services
- Understand when RESTful services are most applicable
- REST vs. gRPC

14.1 Many Flavors of Services

- Web Services come in all shapes and sizes
  - XML-based services (SOAP, XML-RPC, RSS / ATOM Feeds)
  - HTTP services (REST, JSON, standard GET / POST)
  - Other services (FTP, SMTP)
- While SOAP is the most common style of service, increasingly organizations are utilizing REST for certain scenarios
  - REpresentational State Transfer (REST), first introduced by Roy Fielding (co-founder of the Apache Software Foundation and co-author of HTTP and URI RFCs)
  - REST emphasizes the importance of resources, expressed as URIs
  - Used extensively by Amazon, Google, Yahoo, Flickr, and others

14.2 Understanding REST

- REST applies the traditional, well-known architecture of the Web to Web Services
  - Everything is a resource or entity – for example, Orders, Customers, Price quote.
  - Each URI uniquely addresses an entity or set of entities

    - /orders/10025 – Order number 10025
    - /trains/BOM/DEL/02-23-2012 – All trains from Mumbai to New Delhi on 02-23-2012.
- Uses HTTP reply status code to indicate outcome of an operation.
  - 200 or 204 (success), 404 (invalid URI), 500 (general error).
- Uses the **Content-Type** request and response header to indicate data format.

Understanding REST

REST heavily leverages the HTTP protocol. This creates a very familiar environment to provide and consume web services. For example, you can just enter a URL to issue a GET request. This simplicity and familiarity has driven the surge in popularity of this type of web services.

14.3 Understanding REST

- Leverages HTTP method for operating on resources (entities)
  - **GET** – Retrieves a resource or a collection of resources. A read-only operation
  - **DELETE** – Removes a resource
  - **PUT & POST** - “it depends”
- PUT & POST depend on where the resource identifier (primary key) is determined
  - Resource identifier is generated after the request reaches the server
    - eg. an order ID generated by a database insert
    - **POST:** Creates a resource
    - **PUT:** Updates a resource as a whole (e.g., entire Order resource)
  - Resource identifier is part of the data sent by the client
    - eg. a flight number assigned outside the service
    - **PUT:** Creates a new resource

- **POST:** Does a partial resource update (e.g., current flight status)

Understanding REST

PUT and POST are very similar methods. Both can create a new resource or update an existing resource. Updating a resource is easier to distinguish since a PUT is used when the entire content of the resource is being replaced and a POST is used when a partial update is performed. Which method to use when creating a new resource depends on how the resource identifier is determined. If the server-side REST service determines the resource identifier (perhaps auto-generated by a database) then a POST is used with a URI that does not include any resource identifier. If the client determines the resource identifier (perhaps by using a natural key like a social security number) then a POST is used and the URI has the resource identifier included (as if the resource already exists).

14.4 Principles of RESTful Services

- Roy Fielding in his doctoral thesis tried to analyze the wild success of the web and how can some of its architecture principles be applied to any software. Thus RESTful services were born.
- There are four essential elements to any RESTful service
  - **Addressable resources** – A URI uniquely points to a resource or collection of resources,
  - **Uniform operations** – a small number of HTTP methods (GET, POST, PUT, DELETE) that work uniformly for all entities.
  - **Resource representation** – A representation is document in a certain format (XML, JSON etc.) that describes a resource. A consumer and provider of services should be able to negotiate the format.
  - **Stateless** – Services should be stateless. That is, a consumer should be able to complete a business task by issuing a single HTTP request.

Principles of RESTful Services

Source: Architectural Styles and the Design of Network-based Software Architectures, Roy Thomas Fielding, 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

14.5 REST Example – Create

- If the entity identifier is created by the service a POST request is used to create
- Request

```
POST /RESTWeb/catalogs/products HTTP/1.1
Content-Type: text/xml
Content-Length: 142

<?xml version="1.0" encoding="UTF-8"?>
<product>
    <price>125.99</price>
    <name>Baseball bat</name>
</product>
```

- Response

```
HTTP/1.1 201 Created
Location: http://localhost/RESTWeb/catalogs/products/1029
Content-Length: 0
```

REST Example – Create

In this example the primary key, the product ID in this case, is generated by the service when inserting new data and is not part of the information sent in by the client with the initial request. Because of this the POST request is sent and the 'Location' header in the response is critical to know what address can be used to access the created entity later.

14.6 REST Example – Retrieve

- A GET request should always only retrieve data
  - A '404 Not Found' error should be returned if the data doesn't exist
- Request

```
GET /RESTWeb/catalogs/products/1029 HTTP/1.1
Accept: text/xml
```

- Response

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: 137

<?xml version="1.0" encoding="UTF-8"?>
<product>
  <price>125.99</price>
  <name>Baseball bat</name>
  <code>1029</code>
</product>
```

14.7 REST Example – Update

- A PUT request can update existing data
- Request

```
PUT /RESTWeb/catalogs/products/1029 HTTP/1.1
Content-Type: text/xml
Content-Length: 144

<?xml version="1.0" encoding="UTF-8"?>
<product>
    <price>75.99</price>
    <name>Baseball bat</name>
    <code>1029</code>
</product>
```

- Response

```
HTTP/1.1 204 No Content
Content-Length: 0
```

REST Example – Update

Since for this service, new entities are created with a POST request, a PUT request will update an existing entity. Note that the identifier is part of the address the request is sent to.

14.8 REST Example – Delete

- The DELETE request is the simplest with no body to request or response
- Request

```
DELETE /RESTWeb/catalogs/products/1029 HTTP/1.1
```

- Response

```
HTTP/1.1 204 No Content
Content-Length: 0
```

14.9 REST Example – Client Generated ID

- If the identifier for data is provided by the client, a PUT request is used to create a new entity with the service
  - It can also update if the entity already exists
- Request

```
PUT /RESTWeb/flights/AA1215 HTTP/1.1
Content-Type: text/xml
Content-Length: 150

<?xml version="1.0" encoding="UTF-8"?>
<flight>
    <origin>MCO</origin>
    <dest>ORD</dest>
    <status>On Time</status>
    <code>AA1215</code>
</flight>
```

- Response

```
HTTP/1.1 201 Created
Location: http://localhost/RESTWeb/flights/AA1215
Content-Length: 0
```

REST Example – Client Generated ID

In the above example, if the request were updating existing data instead of creating a new entity the request would be the same. The response would likely be a '204 No Content' status code instead of '201 Created' to indicate the data was updated successfully. There would also be no need for a 'Location' header in the response.

14.10 SOAP Equivalent Examples

- In each case, an HTTP POST is used for the <u>same resource URI</u> with a <u>different SOAP message structure</u>
  - The content of the message, usually the first child element of the <Body>, contains information about the action that should be performed
  - There will always be bodies to the requests and responses which will be different SOAP messages depending on the action
- POST **/orderservice.asmx** HTTP/1.1
  - Send a SOAP message and put data in the message body to create an order
- POST **/orderservice.asmx** HTTP/1.1
  - Send a different SOAP message to retrieve the order
  - Response is a custom SOAP response message
- POST **/orderservice.asmx** HTTP/1.1
  - Send another SOAP message and put data in the message body to update the order
- POST **/orderservice.asmx** HTTP/1.1
  - Send another SOAP message to delete the order

14.11 REST Example – JSON

- It is very common for REST services to support communication with JSON, JavaScript Object Notation
  - This is much easier than XML for JavaScript clients that have been very common with REST services
- JSON uses curly brackets for the boundaries of objects along with comma-separated name/value pairs for properties
- Request

```
GET /RESTWeb/catalogs/products/1029 HTTP/1.1
Accept: application/json
```

- Response

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 62

{
  "price" : 125.99,
  "name" : "Baseball bat",
  "code" : 1029
}
```

14.12 REST vs SOAP Communication

- REST leverages the HTTP protocol
  - Well known protocol in the public domain and has best practices and established tools and techniques for handling requests
  - Four methods that are well-known (GET, POST, PUT, DELETE)
  - Developers and administrators already know HTTP well
  - Requests can be bookmarked
  - Responses can be cached
  - Resources are inherently unique and identifiable (URIs)

- With SOAP, developers create their own protocol
  - Everything is sent as an HTTP POST, message must be parsed to understand what is happening
  - Existing tools, techniques, and skill sets must be updated

14.13 More REST vs SOAP

- SOAP has enterprise-grade extensions
  - Security protocols (WS-Security, WS-Trust, WS-SecureConversation, etc.)
  - Policy protocols (WS-Policy, WS-SecurityPolicy)
  - Addressing (WS-Addressing, WS-Event, WS-Notification, WS-ReliableMessaging)
  - And etc…
- SOAP is transport-independent
  - SOAP messages can be placed on any transport layer (HTTP, SMTP, FTP, message queues, etc.)
  - REST is tied to the HTTP protocol

14.14 REST vs SOAP Summary

- REST
  - Ideal for use in Web-centric environments, especially as a part of Web Oriented Architecture (WOA) and Web 2.0
  - Takes advantage of existing HTTP tools, techniques, & skills
  - Little standardization and general lack of support regarding enterprise-grade demands (security, transactions, etc.)
- SOAP

  - ◊ Supports robust and standardized security, policy management, addressing, transactions, etc.
  - ◊ Tools and industry best practices for SOA and WS assume SOAP as the message protocol

14.15 Famous RESTful Services

- Twitter - Short SMS pub-sub communication
- Flickr - Twitter for photos
- Atom - RSS in REST format
- Amazon S3 - Has a REST interface for cloud-management

14.16 Additional Resources

- RESTful Web Services, Web services for the real world, Leonard Richardson, Sam Ruby, O'Reilly Media.
- RESTful Java with JAX-RS, Bill Burke, O'Reilly Media.

14.17 What is gRPC?

- gRPC is a Cloud Native Computing Foundation (CNCF) project.
- gRPC is an open-source and high-performance framework originally created by Google.
- Based on a test, it can be up to 25 times faster than REST services. (source: https://husobee.github.io/golang/rest/grpc/2016/05/28/golang-rest-v-grpc.html)
- It has support for languages such as C++, Java (incl. support for Android), Objective-C (for iOS), Python, Ruby, Go, C#, Node.js
- Google has been using a lot of the underlying technologies and concepts in gRPC for a long time.
- The current implementation is being used in several of Google's cloud products and Google externally facing APIs.
- It is also being used by Square, Netflix, CoreOS, Docker, CockroachDB, Cisco, Juniper Networks and many other organizations and individuals.
- It can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking and authentication.

14.18 Protocol Buffers

- REST mostly uses JSON as the data interchange format. gRPC communicates using binary data via Protocol Buffers (protobuf) for serializing structured data.
- Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data.
- Protocol Buffers are similar to XML but much smaller, faster, and simpler. This results in a large performance boost compared to that of JSON or XML via REST.
- You define how you want your data to be structured once, then you can use the special generated source code to easily write and read your

structured data to and from a variety of data streams and using a variety of languages.

- Language support for gRPC depends on language adoption of proto3 (the latest version of Protocol Buffer) which currently supports Java, C#, C++, Python, Ruby, JavaScript and Objective-C.

14.19 REST vs. gRPC

- Protobuf vs. JSON
- HTTP/2 vs. HTTP 1.1
- Messages vs. Resources and Verbs
- Streaming vs. Request-Response
- Web Browser Support

14.20 Protobuf vs. JSON

- One of the biggest differences between REST and gRPC is the format of the payload.
- REST messages typically contain JSON. gRPC, on the other hand, accepts and returns Protobuf messages.
- From a performance point of view, Protobuf is a very efficient and packed format.
- JSON, on the other hand, is a textual format.

14.21 HTTP/2 vs. HTTP 1.1

- REST depends heavily on HTTP (usually HTTP 1.1) and the request-response model.
- gRPC uses the newer HTTP/2 protocol.
- Some of the major drawbacks of HTTP 1.1 are as follows:
  - **HTTP 1.1 Is Too Big and Complicated -** HTTP 1.1 allows for many

optional parts that contribute to its size and complexity.

- ◊ **The Growth of Page Size and Number of Objects -** The trend of web pages is to increase both the total size of the page and the number of objects on the page that require individual requests.
  - Since each object requires a separate HTTP request, this multiplication of separate objects increases the load on web servers significantly and slows down page load times for users.
- ◊ **Latency Issues -** HTTP 1.1 is sensitive to latency.
  - A TCP handshake is required for each individual request, and larger numbers of requests take a significant toll on the time needed to load a page.
- ◊ **Head of Line Blocking -** The restriction on the number of connections to the same domain significantly reduces the ability to send multiple requests in parallel.

14.22 HTTP/2 vs. HTTP 1.1 (Contd.)

- The major benefits of HTTP/2 are as follows:
  - ◊ **HTTP/2 protocol is binary –** Although, textual protocols are easier for humans to troubleshoot and construct requests manually, in practice, most servers today use encryption and compression anyway.
    - The binary framing goes a long way towards reducing the complexity of handling frames in HTTP 1.1.
  - ◊ **HTTP/2 uses multiplexed streams -** A single HTTP/2 TCP connection can support many bidirectional streams.
    - These streams can be interleaved (no queuing), and multiple requests can be sent at the same time without a need to establish new TCP connections for each one.
    - Servers can now push notifications to clients via the established connection (HTTP/2 push).

14.23 Messages vs. Resources and Verbs

- REST is built very tightly on top of HTTP, but most implementations don't fully adhere to the REST philosophy and use only a subset of its principles.
- The reason is that it's actually quite challenging to map business logic and operations into the strict REST world.
- The conceptual model used by gRPC is to have services with clear interfaces and structured messages for requests and responses.
- This model translates directly from programming language concepts like interfaces, functions, methods, and data structures.
- It also allows gRPC to automatically generate client libraries for you.

14.24 Streaming vs. Request-Response

- REST supports only the request-response model available in HTTP 1.x.
- gRPC takes full advantage of the capabilities of HTTP/2 and lets you stream information constantly.
- There are several types of streaming which are supported by HTTP/2
  - **Server-Side Streaming -** The server sends back a stream of responses after getting a client request message.
  - **Client-Side Streaming -** The client sends a stream of multiple requests to the server. The server sends back a single response,
  - **Bidirectional Streaming - T**he client and the server send information to each other in pretty much free form (except the client initiates the sequence).

14.25 Strong Typing vs. Serialization

- REST
  - The REST services typically use JSON as the data-interchange format.
  - The JSON must be serialized and converted into the target programming language both on the server side and client side.

    - The serialization is another step in the chain that introduces the possibility of errors as well as performance overhead.
    - JSON is theoretically more flexible because you can send dynamic data and don't have to adhere to a rigid structure.
- **gRPC**
    - The gRPC service contract has strongly typed messages that are converted automatically from their Protobuf representation to your programming language of choice both on the server and on the client.
    - gRPC is rigid compared to REST, but it does provide support for several languages and offers major performance benefit over REST.

14.26 Web Browser Support

- Support for gRPC in the browser is not as mature.
- Today, gRPC is used primarily for internal services which are not exposed directly to the world.
- If you want to consume a gRPC service from a web application or from a language not supported by gRPC then gRPC offers a REST API gateway to expose your service.
- The gRPC gateway plugin generates a full-fledged REST API server with a reverse proxy and Swagger documentation.
- With this approach, you do lose most of the benefits of gRPC, but if you need to provide access to an existing service, you can do so without implementing your service twice.

14.27 REST vs. gRPC – In a Nutshell

- In the world of microservices, gRPC will eventually become dominant.
- The performance benefits and ease of development are far too compelling to pass up.
- REST will still be around for a long time.
- REST still excels for publicly exposed APIs and for backward compatibility

reasons.

14.28 Summary

- While SOAP is the most common style of service, increasingly organizations are utilizing REST for certain scenarios
- REST focuses upon resources, expressed as URIs
- URIs can be bookmarked and even cached
- REST leverages the well-known HTTP protocol, SOAP requires developers to invent a protocol
- SOAP offers more enterprise-grade extensions
- SOAP is transport-independent
- REST is ideal for use in Web-centric environments, especially as a part of Web Oriented Architecture (WOA) and Web 2.0
- SOAP is typically a better fit for back-end integration requiring high security, transaction support, and/or complex messaging

Chapter 15 - Introduction to JAX-RS

Objectives

Key objectives of this chapter

- Learn to use JAX-RS
- Work with the basic JAX-RS annotations
- Configure JAX-RS for deployment

15.1 The JAX-RS Specification

- JSR-311 – Released end of 2008
- 5 big implementations – Jersey, Apache CXF, RESTEasy, Restlet, Apache Wink
- Included with Java EE 6, or added as a library for Java EE 5
  - Java EE 6 supports JAX-RS 1.1
- Annotation-based

15.2 The Resource Class

- JAX-RS implements RESTful services by means of a Resource class, which is responsible for acting upon the domain objects which represent the resources
- A Resource class is a plain old Java class (POJO) that uses JAX-RS annotations (e.g., @Path) to implement a corresponding Web resource
- The methods on the object correspond to various calls that can be made to a REST Resource
- A new instance of the Resource class is created for each request for that resource
- Resource classes are typically POJOs that are part of a Web module, but can also be implemented in Java EE 6 as stateless session beans, singleton session beans, or managed beans (JSF). We will focus our attention on the POJO-based implementation.

The Resource Class

The simplest case of a resource class, holding only an in-memory map of Strings.

```
public class MyResource {

  private Map<String, String> underlyingResources;

  public MyResource() {
     underlyingResources = new HashMap<String, String>();
  }

  //Hook this method up to GET all the resources as text/plain
  public String getResourceList() {
     return underlyingResources.toString();
  }

  //Hook this method up to GET a single resource as text/plain
  public String getResource(String id) {
    return underlyingResources.get(id);
  }

  //Hook this method up to POST a single resource, from text/plain
  public String addResource (String id, String data) {
    underlyingResources.put(id, data);
    return "OK";
  }

}
```

15.3 A Bunch of Annotations

- @Path
- @PathParam
- HTTP Methods
  - @GET, @PUT, @POST, @DELETE

A Bunch of Annotations

There are more annotations, but these are the most important ones.

1. What URL are we talking about?
2. What extra data is coming in?
3. Which HTTP call are we handling with this method?

3 questions, 6 Annotations to answer.

15.4 @Path

- @Path tells you what relative URI will map to this resource
- The URI can consist of letters, numbers, and some special characters: _ - * . ! ~ ( ) '
- The basics are simple:
  - ◊ Place @Path in front of a class to define the root URI for the service (e.g., /resources)
  - ◊ Place @Path in front of a method to further qualify the path. The path is appended to the root URI (e.g., /resources/{resourceId}).

```
@Path("/resources")
public class MyResource {
  //...

  public String getResourceList() { … }

  @Path("{resourceId}")
  public String getResource(String id) { … }
}
```

15.5 Using Path Parameters

- Paths can consist of one or more *template parameters*, which are variables defined inside the URI. A template parameter is surrounded with curly braces (e.g., {resourceId}).
- @PathParam is placed before a method parameter to inject (store) the value of a template parameter into the method parameter

```
  // method
  @Path("{resourceId}")
  public String getResource
       (@PathParam("resourceId") String id) { … }
```

- @PathParam can alternately be placed before an instance variable, and you'll get the same results

```
  // instance variable
  @PathParam("resourceId")
  String id;

  // method that references the path parameter
  @Path("{resourceId}")
  public String getResource() { // use 'resourceId' … }
```

Using Path Parameters

If the template parameter is defined at the method level, like in the example above, it's more common to store the template parameter in a method parameter.

On the other hand, if the template parameter is defined at the class level as part of the root resource URI, then it's more common to store the template parameter inside an instance variable. Also, if you have multiple resource methods that respond to a GET request, this prevents you from having to repeat the @PathParam annotation for each method. Instead, you define it only once for an instance variable.

15.6 HTTP Method Binding

- To complete the resource class, you need to bind the resource methods to the HTTP methods using an annotation

```
@Path("/resources")
public class MyResource {
  //...

  @GET
  public String getResourceList() { … }

  @GET
  @Path("{resourceId}")
  public String getResource
       (@PathParam("resourceId") String id) { … }
}
```

- In the example above, a GET /resources request would invoke the getResourceList method and a GET /resources/1 request would invoke the getResource method
- Methods that are annotated with an HTTP method binding (e.g., GET, POST, PUT, DELETE) are referred to as resource methods and must be

public

HTTP Binding

Remember that each of the HTTP methods has a meaning. Determine which HTTP methods you will support, then build resource methods to support those HTTP approaches.

Don't forget that there is a distinction for many HTTP methods between operating on a single resource as compared to operating on the whole set of resources. And so even for a single list of resources, you may still have as many as 7 different standard operations: GET, PUT, DELETE on the whole list, and GET, PUT, DELETE, POST to individual items. Oddly, usually POST an individual item is called with the list-url, not the item url.

15.7 More Complex Paths

- A path is composed of one or more path segments, each of which is preceded by a forward slash
  - e.g., /resources/{resourceId}
  - This path consists of two path segments
- When defining a template parameter, you can optionally define a regular expression by specifying a colon after the template parameter name and then the regular expression
  - e.g., @Path("{resourceId**: \\d+**}") or @Path("{resourceId**: [0-9]+**}")
  - This states that the resourceId must consist of one or more digits
    - If the URI requested does not match the regular expression, then the server does not execute the resource method and returns a 404
    - Without the regular expression, you can specify any legal set of characters for the resourceId's path segment (i.e., letters, numbers, special characters)

15.8 More Complex Paths

- A regular expression is not limited to matching one path segment of a URI. You can match more than one path segment using .+ as your regular expression.

- This can lead to situations where a request URI matches more than one path
- In the example below, which method gets called when the client makes a GET /resources/1/subresource request?
  - In short, the answer is the method which is more specific, which in this case is the getSubresource method

```
@Path("/resources")
public class MyResource {
  @GET
  @Path("{resourceId: .+}")
  public String getResource
       (@PathParam("resourceId") String id) { … }

  @GET
  @Path("{resourceId}/subresource")
  public String getSubresource(
    @PathParam("resourceId") String id) { … }
}
```

15.9 More Complex Paths

- JAX-RS employs a most specific match wins algorithm when matching a request URI to a resource method
- The JAX-RS provider collects the set of deployed URIs and sorts them as follows:
  1. The primary key of the sort is the number of literal characters in the deployed URI and the sort is in descending order
  2. The secondary key is the number of template parameters embedded within the deployed URI and the sort is in descending order
  3. The tertiary key is the number of non-default template parameters (i.e., template parameters that define regular expressions) and the sort is in descending order
- Looking back at the example on the previous slide, /resources/{resourceId}/subresource has 23 literal characters, whereas /resources/{resourceId: .+} only as 11 literal characters, so

/resources/{resourceId}/subresource has more literal characters and the getSubresource method is invoked

- ◊ Had both URIs had the same number of literal characters, then the second sort and possibly the third sort would have been used to determine the winner

15.10 Configuring JAX-RS for Deployment

- A JAX-RS application is packaged as a WAR file. Hence, all of your resource classes and provider classes (discussed later) are included underneath the WEB-INF/classes and/or WEB-INF/lib directories of the WAR file.
- There are several ways to configure a JAX-RS application, but the recommended way in Java EE 6 is to configure javax.ws.rs.core.Application as a Servlet:

```
<servlet>
  <servlet-name>javax.ws.rs.core.Application
  </servlet-name>
</servlet>
<servlet-mapping>
  <servlet-name>javax.ws.rs.core.Application
  </servlet-name>
  <url-pattern>
    /rest/*
  </url-pattern>
</servlet-mapping>
```

- Hence, the URI for invoking a RESTful service would look like this:
  - ◊ http://<host:port>/<context-root>/<servlet-path>/<resource-path>
  - ◊ e.g., http://localhost:8080/RESTWeb/rest/resources/1

Configuring JAX-RS for Deployment

Note that you don't need to define a servlet-class element for the Servlet.

Other approaches for configuring a JAX-RS application can be found at:

http://pic.dhe.ibm.com/infocenter/wasinfo/v8r0/topic/com.ibm.websphere.nd.multiplatform.doc/info/

ae/ae/twbs_jaxrs_configjaxrs11method.html

15.11 Summary

- JAX-RS has 6 basic annotations:
  - @Path, @PathParam, @GET, @PUT, @POST, @DELETE
- We also need to configure the web.xml

Chapter 16 - JAX-RS Data Injection

Objectives

Key objectives of this chapter

- Understand where data can be extracted from the HTTP request
- Use JAX-RS Annotations to inject the data into our applications

16.1 Sources for Injected Data

- Path Parameters
  - Matrix Parameters
- Query String
- HTML forms
- Cookies
- HTTP Headers

16.2 Path Parameters

- Scoping concerns
  - For nested resources

```
http://www.xyz.com/rest/account/1/customer/1
```

- Collection Parameters
- Using PathSegment
  - in place of other data from path parameters

Path Parameters

```
package javax.ws.rs.core;
public interface PathSegment {
  String getPath();
  MultiValuedMap <String, String> getMatrixParameters ();
}
```

16.3 Query Parameters

- @QueryParam
- If a URL comes in with query parameters, they can be injected into your java code
- Many searches that aren't by ID will pass in query parameters

```
http://www.myservice.com/rest/myResource?name=tapas
```

  - Rather than putting @PathParam in the method parameters, we use @QueryParam

```
public List<Customer> getCustomers(@QueryParam("name")
String name) {
```

- Sometimes, you can see cases like the following with multiple parameters of the same name:

http://www.myservice.com/rest/myResource?
name=tapas&**name**=kyle

  - In those cases, you could handle using @QueryParam but with a collection:

```
public List<Customer> getCustomers(@QueryParam("name")
List<String> names) {
```

Query Parameters

A different solution to the same problem would be to rely upon the @Context annotation and the UriInfo interface

```
public interface UriInfo {
  MultivaluedMap<String, String> getQueryParameters();
  MultivaluedMap<String, String> getQueryParameters(boolean decode);
  //other methods
}

@GET
public Customer getCustomer(@Context UriInfo data) {
  String name = info.getQueryParameters().getFirst("name");
  //...
```

16.4 HTML Form Input

- If request is GET, use @QueryParam
- IF request is POST/PUT with *application/x-www-formurlencoded* data
  - ◊ Then use @FormParam
- First, we need a form to submit data from

```
<FORM action="/rest/myResource" method="post">
  First Name: <INPUT name="firstName" /> <BR />
  Last Name: <INPUT name="lastName" /> <BR />
  <INPUT type="submit" />
</FORM>
```

  - ◊ Then we use @FormParam instead of @QueryParam

```
@Path("myResource")
public class MyResourceHandler

@POST
public void addResource( @FormParam("firstName")
String first, @FormParam("lastName") String last) { //...
```

16.5 Cookies

- If you want access to cookies
  - ◊ Stored on / managed by browser
  - ◊ Accessible only to domain that created them
  - ◊ key/value pair + a little more
- @CookieParam
  - ◊ String (value) or javax.ws.rs.core.Cookie object
- If you just want the value:

```
public String getResource(@CookieParam("myCookieName")
 String value) { //...
```

  - ◊ Alternatively, if you want the whole cookie, with its extra data

```
public String getResource(@CookieParam("myCookieName")
 Cookie cookie) { //...
```

Cookies

Annoying that they built another cookie object, when they already had javax.servlet.http.Cookie

The Cookie class

```
package javax.ws.rs.core;

public class Cookie {
  public String getName() { … }
  public String getValue() { … }
  public String getVersion() { … }
  public String getDomain() { … }
  public String getPath() { … }
 //...
}
```

Alternatively again, if you want to, you can use @Context

```
public String getResource(@Context HttpHeaders headers) {
  Set<String> cookieKeys = headers.getCookies().keySet();
  // etc.
```

16.6 Matrix Parameters

- Matrix parameters: @MatrixParam
- Handling urls like:

```
http://www.xyz.com/rest/myResource/someId;key=value
```

- This is handled by a @PathParam annotation like the following

```
@Path("/myResource/{id}")
```

- Matrix Parameters allow you to pass in key-value pairs that may be associated with only a portion of the path instead of the whole request like query parameters
- Just like query parameters they don't impact the path matching
- To extract the data above, you use annotations as below:

```
@GET
@Path("/myResource/{id}")
public String getResource(@PathParam("id") String id,
@MatrixParam("key") String value) { //...
```

16.7 HTTP Headers

- The HTTP protocol specifies header information to be submitted with HTTP Requests
- We might want that information for our processing

```
@HeaderParam
```

- Alternatively, raw headers
- To figure out which browser the client is sending from for an HTML form POST request, we would search the user-agent HTTP header value. This client is using Internet Explorer on Vista.

```
user-agent:
       Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0)
```

  - To extract that data simply in one of our methods:

```
public void addCustomer(@HeaderParam("user-agent")
       String browser, /* other data */) { ...
```

HTTP Headers

A different solution to the same problem would be to rely upon the @Context annotation and the HttpHeaders interface

```
package javax.ws.rs.core;

public interface HttpHeaders {
  List<String> getRequestHeader(String name);
  MultivaluedMap<String, String> getRequestHeaders();
  //other methods
}

@POST
public Customer addCustomer(@Context HttpHeaders headers) {
  String userAgent = headers.getRequestHeader("user-agent").get(0);
  //...
```

16.8 Default Values

- Sometimes we want optional parameters
- @DefaultValue

```
public String getResourceByName
     (@QueryParam("name") String name,
        @DefaultValue(5) @QueryParam("count") int listSize)

}
```

16.9 Parameter Conversion

- JAX-RS will attempt to convert data into the types supplied in your methods
- If conversion fails, 400 or 404 error, depending on which kind of parameter
  - Easiest fix: use regular expressions to constrain path input
- Prevent decoding using @Encoded

```
public String getResource(@Encoded @QueryParam("data")
                                   String data) { … }
```

16.10 Custom Types

- JAX-RS will attempt to convert data into the types supplied in your methods
  - Natural support for: String, int, double, byte[], File, MultivaluedMap<String, String>, (xml) Source
- If conversion fails, 400 or 404 error, depending on which kind of parameter
  - Easiest fix: use regular expressions to constrain path input
- To use custom types, simply add JAXB support/annotations
  - @XMLRootElement, @XMLAccessorType, @XMLAttribute, @XMLElement

Custom Types

An Example:

```
@XMLRootElement(name="customer")
@XMLAccessorType(XMLAccessType.FIELD)
public class Customer {
  @XMLElement String name;
  @XMLElement String address;
  //getters/setters
}
```

Once Customer is defined, then it can be used in input or output, and will be handled automatically by JAX-RS:

```
public Customer getCustomer(@PathParam("id") String id) { // ...
```

or:

```
public void addCustomer(@PathParam("customer") Customer cust) { // ...
```

Note that if you want to have XML and JSON support you must use the JAXB annotations. If you want to use a particular Java type only with JSON, JAXB annotations are still suggested since it is recognized by all JAX-RS implementations. Some JAX-RS implementations may let you use Java types without JAXB annotations with JSON, but often require configuration that is implementation-specific and would not be portable to other JAX-RS implementations.

16.11 Summary

- JAX-RS can extract data from 5 different locations in the HTTP request
- Most of the locations have multiple methods for extracting data
- There is usually a simple standard way to do so with an @XyzParam
- JAXB can use custom data types

Chapter 17 - Designing a RESTful Service

Objectives

Key objectives of this chapter

- Methodology for service identification.
- Identify what constitutes a REST resource.
- How to specify the interface of a service operation.

17.1 Introduction

- Spend extra time designing the interface of a service before you start coding. A simple and consistent interface will make it easier for clients to consume the service.
- A service interface document answers these questions:
  - What does the service do? This is done using plain documentation.
  - Exactly how a client should invoke the service? Factors like URI syntax, URL parameters, HTTP method and MIME types are specified.
  - How should the client interpret the response? The MIME type of the response body and possible HTTP status codes are specified.

Introduction

The interface of a SOAP service is captured in a WSDL document. Creation of a WSDL is practically mandatory. There is no such mandatory step for RESTful services. You may be tempted to dive right into coding. That will be a serious mistake. First, you should go through the business requirements and identify the services. Next, for each service operation, you should design the interface. Details of this methodology are covered in this chapter.

17.2 The Design Methodology

- Capture the business requirements as use cases. For example:
  - UC001 – Upload Flight Status.
  - UC002 – Retrieve Flight Status.
  - UC003 – Upload Flight Pricing.

  - UC004 – Get Price Quote for a Flight.
- Analyze the requirements and identify the resources – such as Flight and Price Quote.
- Model the data schema for the resources. The schema may be manifested as JSON and/or XML documents.
- Analyze the requirements and list the business operations performed on the resources:
  - Upload flight status
  - Get flight status
- Group related operations into services. Example:
  - Flight status service – Supports operations such as upload status and get status.
  - Pricing service – Supports uploading of prices and getting a price quote.
- REST uses a canonical protocol, HTTP, which mandates that the service operations are GET, PUT, POST, and DELETE. State your business operations in terms of the canonical protocol.

17.3 Ingredients of a Service Operation Interface

- The URI syntax – Uniquely points to a resource or collection of resources.
  - /flights/AA1245
- URL parameters – Input data can be passed as URL parameters, used mainly for GET requests.
  - /flights/MIA/LHR/02-23-2012**?sort=price**.
- HTTP method. Indicates the action requested by the client.
- The contents of the request and response body.
- The MIME type (representation formats) of the request or response body.
  - application/xml, text/xml, application/json
- HTTP reply status codes. A service can indicate an error or other

meanings through the status code.

Ingredients of a Service Interface

This slide itemizes various factors you will need to decide when designing a service interface.

The representation format requires further explanation. A service may "speak" in different languages, meaning, it may support different data formats to represent the same resource. For example, a Javascript client can use the JSON format where as a C# client can use XML. A service needs to clearly specify the data formats that are supported and what does a client have to do to work with a specific format. Depending on the format, the Content-Type header of the HTTP request and response as well as the Accept header of the request needs to be set. Details of this are described later in a separate chapter.

17.4 What Constitutes a REST Resource

- A record, or a collection of records.
  - ◊ Data that could fit row-wise into a database table
  - ◊ Also known as an entity.
  - ◊ Usually are nouns in a business requirements document.
- Data must be web and URL accessible
- Same URL must represent the same resource across time
- Example - Twitter
  - A single tweet
  - A given user's tweets, as a list

17.5 Resource Identifiers

- An Individual Resource Identifier is equivalent to a URL
  - ◊ According to the principles of Web Oriented Architecture, and of RESTful services, a unique URL identifies a unique resource
  - ◊ Example: http://www.funnythings.com/jokes/123456
  - Resources can also be lists of resources. A listing of (many) jokes from the site would use almost the same request format as the individual

resource.

- ◊ Example: http://www.funnythings.com/jokes/

17.6 MIME Types

- One of REST's important features is the ability for the same resource to return the same data in different formats.
- Consider supporting these MIME types:
  - ◊ **application/xml** – Useful for full blown programming languages like Java and C#.
  - ◊ **application/json** – Easier for Javascript.
  - ◊ **text/html** – Very easy for a web page to consume since the data is ready for display.
  - ◊ **text/plain** – Appropriate for very simple data types. Can allow for scripting through 'curl' or similar tools.

MIME types

The client needs to specify (just like any HTTP request, using the HTTP headers) what kinds of responses it will be able to read. If the client specifies a narrow range of legitimate MIME Type responses (most likely JSON) and the server supports that response type, then the server software will return data in the appropriate format.

17.7 HTTP Methods

- **GET** – For reading data.
  - ◊ The URI path includes unique identifier of the resource - /orders/**1002**.
  - ◊ URL parameters are allowed to qualify the read request. For example, sort order or filtering parameters. Request body is usually empty.
  - ◊ Reply status 404 if resource doesn't exist.
- **PUT** - Its primary purpose is to update a resource as a whole (e.g., entire Order resource). Its secondary purpose is to create a resource where the consumer chooses the resource identifier (the URI is determined on the client-side). The HTTP request body contains the input data.

- ◊ Reply HTTP status code should be 200 (OK) if an existing resource has been updated.
- ◊ Reply HTTP status code should be 201 (Created) if a new resource is created.
- ◊ The "Location" reply header should contain the URL to the created or updated resource.
- ◊ The PUT method is **idempotent** meaning that the result of a successful PUT request is independent of the number of times it is executed. For example, if you do an update using a PUT, it doesn't matter how many times you do it, the result will always be the same.

17.8 HTTP Methods

- **POST** – Its primary purpose is to create a resource (or sub-resource) where the server chooses the resource identifier (primary key). Its secondary purpose is to do a partial resource update (e.g., total amount of Order resource). The HTTP request body contains the input data.
  - ◊ Reply HTTP status code should be 201 (Created) if a new resource is created and the server has generated a new resource id.
  - ◊ Reply HTTP status code should be 200 (OK) if an existing resource has been (partially) updated.
  - ◊ The "Location" reply header should contain the URL to the created or updated resource.
  - ◊ The POST method is *not* idempotent. For example, if you create a new resource using a POST, you'll get back a new resource each time you repeat the request, instead of the same resource. Hence, the result each time is different.
- **DELETE** – Delete a resource. Reply body is empty.
  - ◊ The URI path includes unique identifier of the resource - /orders/**1002**.

HTTP Methods

For each service operation, you will need to carefully select the HTTP method. In most cases the choice is easy and a general guideline is provided above. You need to take into account some subtleties, though. Both POST and PUT methods are used for updating object state, however, for partial object updates, you must use the POST method (and not PUT). So, if your intent is only to update the phone number in a customer record, use the POST request; using a PUT request to transfer only this one attribute will be in violation of REST interaction principles. In PUT updates, the complete state of the resource has to be transferred with the HTTP request; so in our example, the whole customer record infoset must be submitted to the server even if all of its attributes, except for the phone number, are the same as in the original record.

DELETE is used when a resource is permanently removed from record or flagged as such. What happens when you need additional information before you can delete a resource. For example, a customer wants to cancel a reservation and you want to know the reason for cancellation. Since the request body is empty, a DELETE request can not send any input data beyond what is in the URI. In a situation like that, deviate from the norm and use a POST request (considering this is a partial request with only the reservation status being changed).

17.9 Request and Response Body Structure

- Document the structure of the request and response body if any. This defines the data model of the service interface.
- If using XML, use XML schema definition (XSD) to describe the structure.
- If using JSON use examples and documentation to describe the structure.

17.10 Example Operation Interface Document

Name: Upload flight status.

Description: This service operation updates the status field of an existing flight record in the database.

URI Syntax: /flights/<*FLIGHT-NUMBER*>

HTTP method: POST

Supported MIME types: application/xml, application/json

Request body: A flight status record in the same format as stated in Content-type request header (attach schema here).

Response body: Empty.

Response status: 200 (OK) – updated, 500 – General error.

17.11 Formal (Machine-Readable) Specifications

- One of the things that made REST popular is the lack of schema requirements
- This informal nature is convenient for rapid development of "Back End for Front End" services
  - That's a special case because it's often the same developers working on the front end and the back end
- Informality doesn't really work well for Enterprise Integration
- Over the past few years, there has been growing interest in formal specifications

17.12 Formal (Machine-Readable) Specifications (cont'd)

- WADL was proposed by Sun Microsystems in 2009
  - WSDL for REST
  - XML based
  - Very dead
- Swagger/Open API
  - Part of the Open API Initiative since January 2016
  - Includes a Specification and Tools for various platforms
  - Technically, Swagger is the UI framework, Open API is the specification
- RAML
  - "RAML is a development language designed to describe APIs in a clear way..."
  - Developed as open source, licensed under Apache Software License 2.0

17.13 Summary

- Always formally capture business requirements. Use case documents are great for that.
- Analyze the requirements and identify the resources.
- Do the data modeling and design the schema for the resources.
- Analyze the requirements and identify the service operations.
- Group related operations in a service.
- Design the interface of each identified operation.
- Create the service interface document.

Chapter 18 - JAX-RS Content Types

Objectives
Key objectives of this chapter ■ Explain what internet media types are ■ Use the JAX-RS @Consumes annotation to specify the required request Content-Type ■ Use the JAX-RS @Produces annotation to specify the response Content-Type ■ Use the JAX-RS MediaType Class to improve code quality ■ Return a response consisting of XML ■ Use JAXB to simplify sending and receiving XML ■ Control content negotiation with the @Consumes and @Produces annotations statically ■ Control content negotiation dynamically

18.1 Internet Media Types

- Request and response formats are identified by internet media type
  - Also known as MIME type or Content-Type
- Example: `text/html; charset=UTF-8`
- Standard values are defined by the IANA[1]
- Prefix `x-` indicates a non-standard type or subtype
- Prefix `vnd.` indicates vendor-specific

Notes

[1]Internet Assigned Number Agency

18.2 Use of Media Type in REST

- A resource may be represented using different formats like XML and JSON. A service should attempt to support as many formats as possible to reach out to a large audience of client developers.
- If a request body contains data, the client should set the **Content-Type** header to the correct MIME type.
  - In the server side, you should be able to define a separate method that exclusively deals with that representation of the resource.
- If a client has a preference of one representation over the other, it should be able to express that by setting the **Accept** header in the request. The service should do its best to honor that preference and return data in the preferred format. This is called **content negotiation**.
- If a response body contains data, the service should set the **Content-Type** header to indicate the MIME type.
- The JAX-RS API provides excellent support to implement all aspects of media type usage identified above.

18.3 The @Consumes Annotation

- Different clients can send data in the request body using different format.
- The @Consumes annotation matches a media type set in the Content-Type request header with a service class or method.
  - It is important that the client sets the Content-Type header properly.
- Example:

```
@PUT
@Consumes("application/xml")
public void addNewsXML(NewsItem n) {
    // Store the resource
}
@PUT
@Consumes({"text/plain", "text/html"})
public void addNewsText(String n) {
    // Store the resource
}
```

The @Consumes Annotation

The @Consumes annotation helps us assign a different method for different data formats in the request. The example in the slide shows a good use of custom methods to handle different formats. XML represents structured data. You can describe the author, category and text for a news article. When a news article is posted as plain text or HTML, we have no structure and we get just the article text. This is why a separate method for it is appropriate.

You don't have to always create a separate method for each data type. For example, XML and JSON can share the same method:

```
@PUT
@Consumes({"application/xml", "application/json"})
public void addNews(NewsItem n) {
    // Store the resource
}
```

18.4 Content Negotiation

- When a client requests to retrieve a resource (GET request), it can specify its preference for the data format in the Accept header.

```
GET /example.com/orders/1023
Accept: application/xml
```

- JAX-RS will automatically serialize a Java object in the appropriate format. For example, the following will serialize Order into XML or JSON based on Accept header.

```
public Order getOrder(...) {}
```

- Since a service may not support all possible formats, a client can express relative preference for multiple formats. A few basic rules apply:
  - A weight value can be set using the q parameter. Default value is 1. The MIME type with the highest weight is selected.

```
Accept: text/html;q=0.5,application/xml;q=0.8
```

  - When several formats have the same weight, the first in the order is selected.

```
Accept: application/xml, application/json
```

18.5 The @Produces Annotation

- The @Produces annotation can be used to assign a different method for a different reply format MIME type. JAX-RS will automatically match the incoming Accept header with the correct method.
- Example:

```
//Matched with: Accept: text/html
@GET
@Produces("text/html")
public String getNewsHTML() {}

//Matched with: Accept: application/xml, application/json
@GET
@Produces({"application/xml", "application/json"})
public NewsItem getNews() {}
```

18.6 The MediaType Class

- The JAX-RS MediaType class defines Strings constants for commonly-used media types. Use them with @Consumes and @Produces to avoid typo:

```
@GET
@Produces({MediaType.APPLICATION_XML,
MediaType.APPLICATION_JSON})
public Response getNews() {}

@PUT
@Consumes(MediaType.APPLICATION_XML)
public void addNewsXML(NewsItem n) {}
```

18.7 JAXB

- JAX-RS is integrated with JAXB to support much easier handling of XML or JSON.
- Consider the following domain class with JAXB annotations

```
@XmlRootElement
```

```
public class Person {
   private String name;
   @XmlElement public String getName() {return name;}
   public void setName(String name) {this.name = name;}
   // etc.
}
```

- With JAX-RS a service method can return an instance of the domain class and the JAX-RS runtime will use JAXB to perform the serialization

```
@GET
@Produces(MediaType.APPLICATION_XML)
public Person getMyResourceAsXml(
      @PathParam("name") String name) {
   Person person = new Person();
   person.setName(name);
   // etc.
   return person;
}
```

18.8 Dynamic Content Negotiation

- We have seen how to use the @Produces annotation to handle request for a different media type using different methods. With dynamic content negotiation, the same method can inspect the media type preferences of the client and act differently.
- The Request object has a selectVariant(...) method that
  - Takes a list of Variant objects handled by the web service
  - Compares the list with the Accept, Accept-Encoding, Accept-Language, and Accept-Charset headers of the client request
  - Returns the best fit of all the Variant objects in the list or null
- Example

```
@GET
public Response get(@Context Request request) {
    List<Variant> vs = Variant.mediatypes(
        MediaType.TEXT_XML, MediaType.TEXT_PLAIN).build();
    Variant v = request.selectVariant(vs);
    if (v == null) {
        return Response.notAcceptable(vs).build();
    } else if (v.getMediaType() == MediaType.TEXT_XML {...
        // Calculate the response for variant v
```

JAXB

Note that if you want to have XML and JSON support you must use the JAXB annotations. If you want to use a particular Java type only with JSON, JAXB annotations are still suggested since it is recognized by all JAX-RS implementations. Some JAX-RS implementations may let you use Java types without JAXB annotations with JSON, but often require configuration that is implementation-specific and would not be portable to other JAX-RS implementations.

18.9 Summary

- The JAX-RS @Consumes and @Produces annotations are used to
  - Match Content-Type and Accept headers in the request
  - Set the Content-Type header in the response
- The JAX-RS MediaType Class defines useful constants
- A JAX-RS web service may return XML using, for example, JAXB
- A JAX-RS web service may return JSON
- The JAX-RS runtime uses content negotiation to match the request with the best service method

Chapter 19 - Building Complex JAX-RS Responses

Objectives

Key objectives of this chapter

- Set response codes and headers with the Response and ResponseBuilder classes
- Return cookies
- Return exceptions

19.1 HTTP Response Status Codes

- The first line of an HTTP response includes a numerical status code
- Example HTTP response illustrating a status of 200:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 49
...
```

- Example HTTP response status codes, including the standard phrase, which are often used by RESTful web services
  - `200 OK`
  - `201 Created`
  - `204 No Content`
  - `304 Not Modified`
  - `400 Bad Request`
  - `401 Unauthorized`
  - `403 Forbidden`
  - `404 Not Found`
  - `405 Method Not Allowed`
  - `406 Not Acceptable`
  - `415 Unsupported Media Type`
  - `500 Internal Server Error`
- A request for a non-existent resource might result in a 404
- How can we set the response status code programmatically?

19.2 Introduction to the JAX-RS Response Class

- Using Response to set the response status code, a header, and the response body

  ```
  @GET
  @Produces(MediaType.TEXT_PLAIN)
  public Response getTheResource() {
     return Response.status(200).type(MediaType.TEXT_PLAIN).
            entity("the result").build();
  }
  ```

- The output—some header values modified for generality

  ```
  HTTP/1.1 200 OK
  Server: SomeServer
  X-Powered-By: SomeInfo
  Content-Type: text/plain
  Content-Length: 10
  Date: SomeDate

  the result
  ```

- Using the Response class we can set the response status code and specify headers
- To specify a response status code of 200 we could also have used `ok()`

Introduction to the JAX-RS Response Class

Be sure to import javax.ws.rs.**core**.Response rather than javax.ws.rs.Response.

In later examples we will continue to "generalize" values for headers such as Server and Date

19.3 Using the Response and Response.ResponseBuilder Classes

- Many of the Response class methods return an instance of Response.ResponseBuilder, e.g.

`ok()`

```
status(...)
```

- In typical use, the “chain” of method invocations starts with one of these methods, followed by zero or more methods in the next group

- An instance of Response.ResponseBuilder has many methods that return that same Response.ResponseBuilder instance, e.g.

```
type(...)           // set the Content-Type header
tag(...)            // set the ETag (entity tag) header
language(...)       // set the Content-Language header
header(..., ...)    // set any header
entity(...)         // set the payload of the message
```

- Lastly, Response.ResponseBuilder has a method that returns a Response

```
build()
```

- This allows the methods to be chained as shown previously

```
return Response.status(200).type(MediaType.TEXT_PLAIN).
       entity("the result").build();
```

19.4 Example: Conditional HTTP GET

- Some requests are frequently made and have seldom-changing results
  - Example: map tiles
- Why waste bandwidth sending the same data repeatedly?
- The client and the service can work together to solve this problem
  - Remember: the service holds no application state
- The service
  - Sends response with Last-Modified header and/or ETag header
- The client, when requesting the same resource
  - Sends If-Modified-Since header and/or If-None-Match header (matching value(s) sent earlier)
- The service
  - If the response has not changed sends a response code of 304 Not Modified with no entity body
  - Otherwise, returns the proper, new response

19.5 Returning Cookies

- A service may store client information on the server and provide the client with a key to this data; as a result
  - The service is less scalable because an increase in clients results in an increase in state information storage space needed on the server
  - Failover is harder to implement because the state information must be replicated on multiple servers

```
@GET
@Produces(MediaType.TEXT_PLAIN)
public Response getTheResource() {
   final int oneDayInSeconds = 60 * 60 * 24;
   NewCookie nc = new NewCookie(
         "myKey", "someValue", "/myService", "acme.com",
         "REST probably shouldn't use cookies",
         oneDayInSeconds, false);
   return Response.status(200).type(MediaType.TEXT_PLAIN).
          cookie(nc).language("en").tag("myTag").
          entity("the result").build();
}
```

19.6 Cookies in Response Headers

- The resulting response

```
HTTP/1.1 200 OK
Server: someServer
Set-Cookie: myKey=someValue; Version=1; Comment="REST ↴
probably shouldn't use cookies"; Domain=acme.com; Max-↴
Age=86400; Path=/myService
ETag: myTag
Content-Language: en
Content-Type: text/plain
Content-Length: 10
Date: someDate

the result
```

19.7 Reading Cookies

- If your service sends cookies to the client and the client returns them, your service will need to read the cookies
  - ◊ The service is less scalable because an increase in clients results in an increase in state information storage space needed on the server
- Example

```
public Response getTheResource(@Context HttpHeaders
headers) {
   Collection<Cookie> cookies =
     headers.getCookies().values();

   if (cookies.isEmpty()) {
      System.out.println("None");
   } else {
      for (Cookie cookie : cookies) {
         System.out.println("- " + cookie.toString());
      }
   }
}
```

19.8 Returning an Exception

- JAX-RS includes WebApplicationException
- Constructing WebApplicationException from a Response

```
if ( ... ) {
   ResponseBuilder builder = Response.
                              status(Status.NOT_FOUND);
   builder.type(MediaType.TEXT_HTML);
   builder.entity("<h3>Person Not Found</h3>");
   throw new WebApplicationException(builder.build());
} else {
   // ...
}
```

- A WebApplicationException may be constructed from an int

```
throw new WebApplicationException(500);
```

- ◊ The JAX-RS runtime will generate HTML that includes the status code

19.9 ExceptionMappers

- To unify exception handling, the @Provider annotation may be used to register Exception Mappers
- One or more ExceptionMappers, each supporting a different type, may be registered
- The JAX-RS runtime picks them up by scanning
- Example

```
@Provider
public class MyExceptionMapper implements
ExceptionMapper<InvalidCustomerException> {
   @Override
   public Response toResponse(
              InvalidCustomerException exception) {
      return Response.status(500).build();
   }
}
```

- Non WebApplicationExceptions thrown by service methods are caught by the JAX-RS runtime and passed to the matching ExceptionMapper

19.10 Summary

- The JAX-RS Response Type can be used to set a specific status code
- The JAX-RS Response and Response.ResponseBuilder classes can also be used to set headers
- The JAX-RS NewCookie class can be used to return cookies, although use of cookies may not be RESTful
- Exceptions may be thrown by web service methods
  - ◊ WebApplicationException
  - ◊ Application exceptions

Chapter 20 - Clients of JAX-RS Services

Objectives

Key objectives of this chapter

- Use Apache HttpClient
- Use JAX-RS implementation client classes such as those from
  - Jersey
  - RESTEasy
  - Apache CXF
- Explain the importance of HATEOAS
- Build links with UriBuilder and UriInfo
- Use Atom Links
- Use WADL to define a RESTful web service

20.1 Java Web Service Clients

- In theory REST client code could be coded at the socket level:

```
Socket requestSocket = new Socket("localhost", 7001);
PrintWriter pw =
   new PrintWriter(requestSocket.getOutputStream(), true);
pw.write("GET /01/svc/myResource/Roy%20Fielding ⮯
          HTTP/1.1\r\nHost: localhost\r\n\r\n");
pw.flush();
while (true) {
   int x = requestSocket.getInputStream().read();
   if (x == -1) {
      break;
   }
   System.out.print((char)x);
}
pw.close();
requestSocket.close();
```

- Fortunately, libraries exist to handle these details
  - Apache HttpClient

  - JAX-RS

20.2 Java Web Service Clients

- JAX-RS implementations typically include client classes
  - Jersey
  - RESTEasy
- Apache CXFLibraries are also available that handle hypertext
  - Atom Links
  - UriBuilder and UriInfo
- Clients may also use machine-readable documentation for RESTful web services: WADL

20.3 Apache HttpClient

- The Apache HttpComponents include an HttpClient library to simplify REST client code

```
DefaultHttpClient httpclient = new DefaultHttpClient();
String resource = "Roy%20Fielding";
try {
   HttpGet httpget = new HttpGet(
      "http://localhost/01/svc/myResource/" + resource);
   HttpResponse response = httpclient.execute(httpget);
   HttpEntity entity = response.getEntity();
   BufferedReader br = new BufferedReader(
                       new InputStreamReader(
                       response.getEntity().getContent()));
   String line;
   while ((line = br.readLine()) != null) {
      System.out.println(line);
   }
   EntityUtils.consume(entity);
} catch (Exception e) {
   // handle...
} finally {
   httpclient.getConnectionManager().shutdown();
}
```

Apache HttpClient

Import from org.apache.http packages. The necessary JAR files are available from http://hc.apache.org/index.html

20.4 More Apache HttpClient Code

- Using HttpClient to access the status line and cookies

```
System.out.println(
   "Status Line: " + response.getStatusLine());

System.out.println("Cookies:");
List<Cookie> cookies =
   httpclient.getCookieStore().getCookies();
if (cookies.isEmpty()) {
   System.out.println("None");
} else {
   for (Cookie cookie : cookies) {
      System.out.println("- " + cookie.toString());
   }
}
```

20.5 JAX-RS Implementation Client Libraries

- JAX-RS 2.0 will support a common client API

```
Client client = ClientFactory.newClient();

String result = client.target("http://localhost/svc/➘
          myResource/Tim%20Berners-Lee")➘
          .request(MediaType.TEXT_PLAIN).get(String.class);
```

- In the mean time, the various JAX-RS implementations provide client libraries
  - JBoss RESTEasy

- ◊ Jersey
- ◊ Apache CXF

20.6 RESTEasy Example

- This example illustrates use of org.jboss.resteasy.client.ClientRequest and ClientResponse

```
ClientRequest request = new ClientRequest(
   "http://localhost:8080/01/svc/myResource/Roy
%20Fielding");
ClientResponse<String> response =
request.get(String.class);

if (response.getStatus() != 200) {
   throw new RuntimeException("Failed with HTTP error code
"
    + response.getStatus());
}
BufferedReader br = new BufferedReader(new
InputStreamReader(
   new ByteArrayInputStream(response.getEntity().
       getBytes())));

String output;
while ((output = br.readLine()) != null) {
   System.out.println(output);
}
```

- The RESTEasy client API is built on HttpClient

20.7 HATEOAS

- One of the four REST interface constraints outlined in Roy Fielding's dissertation is “Hypertext as the engine of application state”
- A web service response should typically contain the complete URIs for the client application to choose between; each of these URIs represents a

possible next state for the client application

- “Hypertext as the engine of application state” is abbreviated HATEOAS
- JAX-RS does not currently support HATEOAS directly
  - By this we mean that JAX-RS does not provide much help including hyperlinks in a web service response
- JAX-RS 2.0 *will* support hypermedia

20.8 Building Links with UriBuilder and UriInfo

- The current version of JAX-RS does include some classes which may be used to return links
- To get the URL of the current request, for use in building new links in the response:
  - Get a UriInfo object, either by injection into a resource class field or as a method parameter

```
@Context UriInfo ui
```

  - Get the complete URL

```
String urlString =
   ui.getAbsolutePath().toASCIIString();
```

- To build new links, it's easier to use a UriBuilder than to manipulate Strings directly
- The following code adds “/” and the person name to the end of the original URL

```
UriBuilder ub = ui.getAbsolutePathBuilder();
ub = ub.path(person.getName());
String newUrl = ub.build().toString();
```

20.9 Using Atom Links for State Transitions

- The Atom Syndication Format
  - An XML vocabulary developed for syndication and editing
  - Has a linking model that implements HATEOAS

- In Atom, the link element rel attribute may have values such as "edit" and an href attribute that is a full link
- Example excerpt from an Atom 1.0 feed

```
<entry>
   <link rel="alternate"
         type="text/html"
         href="http://someHost/someRoot/some.html"/>
   <link rel="edit"
         href="http://serviceHost/svc/someId/edit"/>
   …

</entry>
```

20.10 WADL

- Web Application Description Language (WADL) is the REST counterpart to WSDL for SOAP-based web services
  - WADL provides a machine-readable description of services accessed via HTTP
- JAX-RS does not require a REST service provide a WADL description though
  - It is often used by JAX-RS implementations though to provide a service description
- WADL can be read by service development or testing tools, like soapUI, for example

20.11 WADL Example

```
<application xmlns="http://wadl.dev.java.net/2009/02"
  xmlns:ex="http://localhost">
  <grammars>
    <include href="exampleSchema.xsd" />
  </grammars>
  <resources base="http://localhost/01/svc/">
    <resource path="myResource/{name}">
```

```
      <method name="GET">
        <request>
          <param name="name" type="xsd:string"
               style="template"/>
        </request>
        <response status="200">
          <representation mediaType="application/xml"
                        element="ex:resource"/>
        </response>
      </method>
    </resource>
  </resources>
</application>
```

20.12 Summary

- The Apache HttpClient library simplifies RESTful web service client code
- Implementers of the JAX-RS specification typically include their own client libraries
  - Jersey
  - RESTEasy
  - Apache CXF
- A RESTful web service should satisfy the HATEOAS constraint
- Use the JAX-RS classes UriBuilder and UriInfo to build hyperlinks
- Use Atom Links to achieve HATEOAS
- Use WADL to define a RESTful web service

Chapter 21 - Securing JAX-RS Services

Objectives

Key objectives of this chapter

- Explain HTTP basic user access authentication
- Integrate a REST service with JEE web security
- Write a client to a secure web service
- Use the JAX-RS security annotations

21.1 HTTP Basic Authentication

- HTTP/1.1 supports two mechanisms for user access authentication
  - basic
  - digest
- Typical basic user access authentication scenario
  - Client makes request without providing user name and password
  - Server/service responds with
    - 401 Unauthorized status code
    - required authentication scheme (basic)
    - the authentication realm (some text)
  - Client updates request to include an Authorization header with value the base64 encoding of username+":"+password" and re-sends
  - Server/service authenticates and sends response
- A client may pre-emptively send the Authorization header in its initial request

21.2 Example Client

- Setting credentials in the client

```
Credentials cred = new UsernamePasswordCredentials("username", "pw");
HttpClient client = new HttpClient();
```

```
        client.getState().setCredentials(AuthScope.ANY, cred);
        client.getParams().setAuthenticationPreemptive(true);

        ClientExecutor ex = new ApacheHttpClientExecutor(client);

        ClientRequest request = new ClientRequest(
              "http://localhost:8080/01/svc/myResource/Roy%20Fielding");

        request.setHttpMethod("GET");
        request.accept("text/plain");

        ClientResponse<String> response = ex.execute(request);

        System.out.println("Status Line: " + response.getStatus());
        System.out.println("Entity : " + response.getEntity(String.class));
```

Notes

This example uses RESTEasy.

Imports are from packages org.apache.commons.httpclient and org.jboss.resteasy.client.

Note the use of setAuthenticationPreemptive(true) to force the credentials to be sent with the first request rather than waiting for a 401.

21.3 The WWW-Authenticate and Authorization Headers

- Example response to an unauthorized request

```
401 Unauthorized
WWW-Authenticate: Basic realm="Person Service"
```

- The client re-sends the request with an additional header encoding username colon password

```
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

21.4 Java EE Security Roles

- In Java EE resources, including web service requests, are protected based on roles
- The association of users with roles is not part of Java EE
- The security-constraint below restricts GETs to users in the role MyRole

```
<security-constraint>
```

```
      <web-resource-collection>
         <web-resource-name>MyResource</web-resource-name>
         <url-pattern>/svc/*</url-pattern>
         <http-method>GET</http-method>
      </web-resource-collection>

      <auth-constraint>
         <role-name>MyRole</role-name>
      </auth-constraint>

   </security-constraint>

   <login-config>
      <auth-method>BASIC</auth-method>
   </login-config>

   <security-role>
      <role-name>MyRole</role-name>
   </security-role>
```

21.5 Integration with Web Container Security

- JEE web.xml security constraints do not support JAX-RS well in some situations because web.xml URL pattern matching is very very limited
  - ◊ URL patterns in web.xml only support simple wildcards
  - ◊ Example:
    `/{customerName}/ssn/{ssnValue}`
    cannot be mapped as a web.xml URL pattern
    `/*/ssn/*`
    because multiple asterisks are not supported
- Fortunately JAX-RS provides security annotations to completely solve this problem

21.6 Java EE Security Annotations

- Individual web service methods may be annotated with any of the following
  - ◊ `@RolesAllowed({"Customer"})`

  - `@PermitAll`
  - `@DenyAll`

Notes

RESTEasy requires the following in web.xml to enable role-based security.

```
<context-param>
   <param-name>resteasy.role.based.security</param-name>
   <param-value>true</param-value>
</context-param>
```

21.7 SecurityContext

- A SecurityContext may be inserted into a secure resource class.

  ```
  @Context
  SecurityContext securityContext;
  ```

- The SecurityContext may be used for auditing and/or for fine-grained security control
- Methods include
  - getUserPrincipal()
    - Use toString() or getName() to get the user name
  - getAuthenticationScheme()
    - Returns, for example, SecurityContext.BASIC_AUTH, which is "BASIC"
  - isSecure()
    - Whether this request was made via a secure channel such as HTTPS
  - isUserInRole(String role)
    - Useful when @RolesAllowed specifies multiple roles

21.8 Restrictions Based on Content Type

- Suppose, for example, that we want a document service to allow unrestricted access to the text representation of a document but allow only authors to access the same document as XML
- This can be achieved by combining the @RolesAllowed and @Provides annotations
  - Annotate the text service method with @PermitAll and @Provides(MediaType.TEXT_PLAIN)
  - Annotate the XML service method with @RolesAllowed("Author") and @Provides(MediaType.TEXT_XML)

21.9 Summary

- HTTP basic user access authentication allows clients to specify a user name and password
- A JEE security-constraint is used, in conjunction with annotations, to protect REST services
- Write a client to a secure web service
- A REST service can be made secure with JAX-RS security annotations

Chapter 22 - REST vs. SOAP

Objectives

Key objectives of this chapter

- Define REST and SOAP
- Distinguish RPC SOAP and Document SOAP from REST
- List the strengths of each
- Select the appropriate solution

22.1 Defining REST

- To be considered RESTful, an architecture must satisfy five architectural style constraints; an optional constraint, code on demand, may also be satisfied
- REST does not specify the implementation of these constraints
- The constraints are not arbitrary; an architecture that complies with them will exhibit improved
  - Performance
  - Scalability
  - Simplicity
  - Modifiability
  - Portability
  - Reliability

22.2 The Six REST Architectural Style Constraints

- Client/Server
  - Benefit: Servers and clients may be replaced and developed independently as long as the interface between them is not altered
- Stateless
  - Benefit: Servers are more scalable

- Cacheable
  - Benefits: Prevent clients reusing stale or inappropriate data in response to further requests, can partially or completely eliminate some client/server interactions, improving scalability and performance
- Layered system (a client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way)
  - Benefits: Intermediary servers may improve system scalability by enabling load-balancing and by providing shared caches, they may also enforce security policies

22.3 The Six REST Architectural Style Constraints

- Uniform interface between clients and servers
  - Benefits: The uniform interface between clients and servers simplifies and decouples the architecture, which enables each part to evolve independently
- Code on demand (optional)
  - Servers can temporarily extend or customize client functionality by the transfer of executable code, e.g., Java applets
  - Benefit: increased flexibility

22.4 The Four REST Interface Constraints

- Identification of resources
  - Each resource has a URI
  - Each resource is accessed through a defined set of HTTP methods (GET, PUT, POST, DELETE)
- Manipulation of resources through representations
  - Resource can have one or more representations (e.g., application/xml, application/json, text/html, etc.)
  - Clients and servers negotiate to select representation

- Self-descriptive messages
  - Requests and responses contain not only data but can also contain additional headers such as whether caching is allowed, authentication requirements, etc.
- Hypermedia as the engine of application state
  - A REST API must be hypertext driven
  - That is, responses should include complete URIs for other resources (hyperlinks) rather than requiring clients to construct resource URIs based on out-of-band information about how the API works

22.5 Hypermedia Examples

- A web service that returns a list of search results each of which includes a complete link (URL), perhaps with other information such as a summary
  - The client can easily retrieve a individual result by following the link
- A web service that returns a map image plus complete links (URLs) for north, south, east, and west
- A non-hypermedia example:
  - A web service that returns a map image but requires the client to “calculate” the URLs for north, south, east, and west

22.6 Defining SOAP

- SOAP defines a specific XML format for requests and responses
- The detailed interface for a specific SOAP service is defined in its WSDL
- The request may be interpreted as an RPC invocation (RPC SOAP) or not an RPC invocation (Document SOAP)
  - RPC SOAP exposes named methods
  - Document SOAP allows any document satisfying the WSDL to be sent as a request
- A SOAP service may be implemented over many different transport

protocols

22.7 RPC SOAP vs. REST

- Method Information
    - RPC SOAP allows the service to define arbitrary method names
    - REST services are limited to the HTTP methods,
        - GET
        - PUT
        - DELETE
        - POST
        - Other methods (HEAD, OPTIONS)
- Data Scoping
    - With RPC SOAP, the data to be operated on is defined within the request
    - With REST the data to be operated on is defined in the URL
        - This is always true for GET requests and usually true for DELETE requests
        - However, there is data in the request body for POST and PUT requests

22.8 Document SOAP vs. REST

- Document SOAP ("SOAP FTP") allows more flexibility in the request; the WSDL can define the request to be any XML document
- The difference between SOAP data scoping and REST data scoping remains

22.9 Where SOAP Shines

- Works with many transport protocols

    - HTTP
    - JMS
    - FTP
    - SMTP
- Supports the RPC paradigm
    - Familiar
    - Easy for developers to understand
- Interface is strictly defined with WSDL
    - May become a disadvantage if the interface needs to be changed
- Better support for security
    - WS-Security
- Better support for transactions
    - WS-AtomicTransaction
- Other advanced features (Reliable Messaging, etc)

22.10 Where REST Shines

- Optimized for HTTP
    - Faster and more scalable due to caching
- Simple to implement; simple for clients
- Supports many data formats
    - XML
    - JSON
    - text
    - etc.
- Fewer interoperability problems
- Easier to make small changes in the service API

    - Don't need to regenerate client proxies

22.11 Selecting an Appropriate Solution

- REST is ideal for CRUD access to named resources
- SOAP provides better security and transactional support
- The ideal solution for a particular situation may be a hybrid

22.12 Summary

- Differences between REST and SOAP
    - With REST the resource to be operated on is part of the URI; with SOAP it is defined within the body of the request
    - With REST the operations are limited to the standard set of HTTP methods; with SOAP, arbitrary methods may be defined
- REST is simpler and more scalable; ideally suited for CRUD operations on named resources
- SOAP provides better support for security and transactions
- The appropriate choice depends on the requirements