



Developing Web Services with Java™ Technology

DWS-4050-EE6

Revision A.0

Copyright 2010 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Sun, Sun Microsystems, the Sun logo, Java, EJB, JSP, and JavaServer Pages are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Federal Acquisitions: Commercial Software Government Users Subject to Standard License Terms and Conditions

Export Laws. Products, Services, and technical data delivered by Sun may be subject to U.S. export controls or the trade laws of other countries. You will comply with all such laws and obtain all licenses to export, re-export, or import as may be required after delivery to You. You will not export or re-export to entities on the most current U.S. export exclusions lists or to any country subject to U.S. embargo or terrorist controls as specified in the U.S. export laws. You will not use or provide Products, Services, or technical data for nuclear, missile, or chemical biological weaponry end uses.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS, AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright 2010 Sun Microsystems Inc., 4150 Network Circle, Santa Clara, California 95054, Etats-Unis. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a.

Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Sun, Sun Microsystems, the Sun logo, Java, EJB, JSP, et JavaServer Pages sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

Législation en matière d'exportations. Les Produits, Services et données techniques livrés par Sun peuvent être soumis aux contrôles américains sur les exportations, ou à la législation commerciale d'autres pays. Nous nous conformerons à l'ensemble de ces textes et nous obtiendrons toutes licences d'exportation, de ré-exportation ou d'importation susceptibles d'être requises après livraison à Vous. Vous n'exporterez, ni ne ré-exporterez en aucun cas à des entités figurant sur les listes américaines d'interdiction d'exportation les plus courantes, ni vers un quelconque pays soumis à embargo par les Etats-Unis, ou à des contrôles anti-terroristes, comme prévu par la législation américaine en matière d'exportations. Vous n'utiliserez, ni ne fournirez les Produits, Services ou données techniques pour aucune utilisation finale liée aux armes nucléaires, chimiques ou biologiques ou aux missiles.

**LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES
EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI
APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A
L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFAÇON.**



**Developing Web Services with Java™
Technology**

About This Course

Course Goals

On completion of this course, you should be able to:

- Understand and explain why and how web services frameworks can be used to deploy and consume services.
 - Understand the trade-offs associated with the use of SOAP-based or RESTful web services.
 - Understand and use the JAX-WS technology to deploy and consume web services.
 - Understand and use the JAX-RS technology to deploy and consume web services.

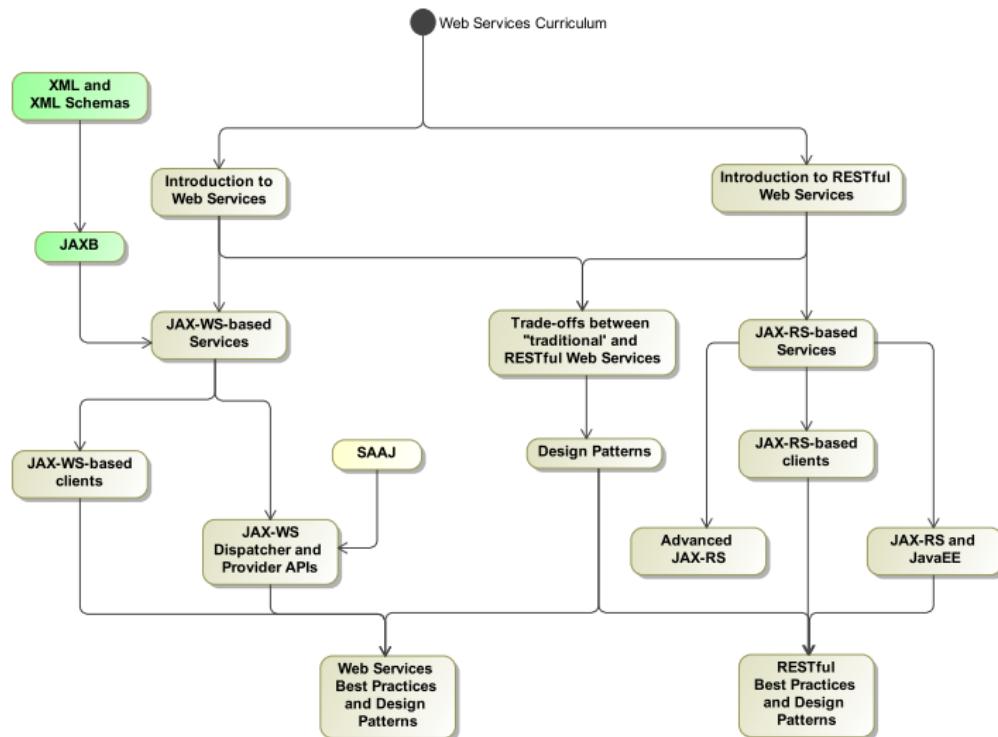
Course Goals

(continued)

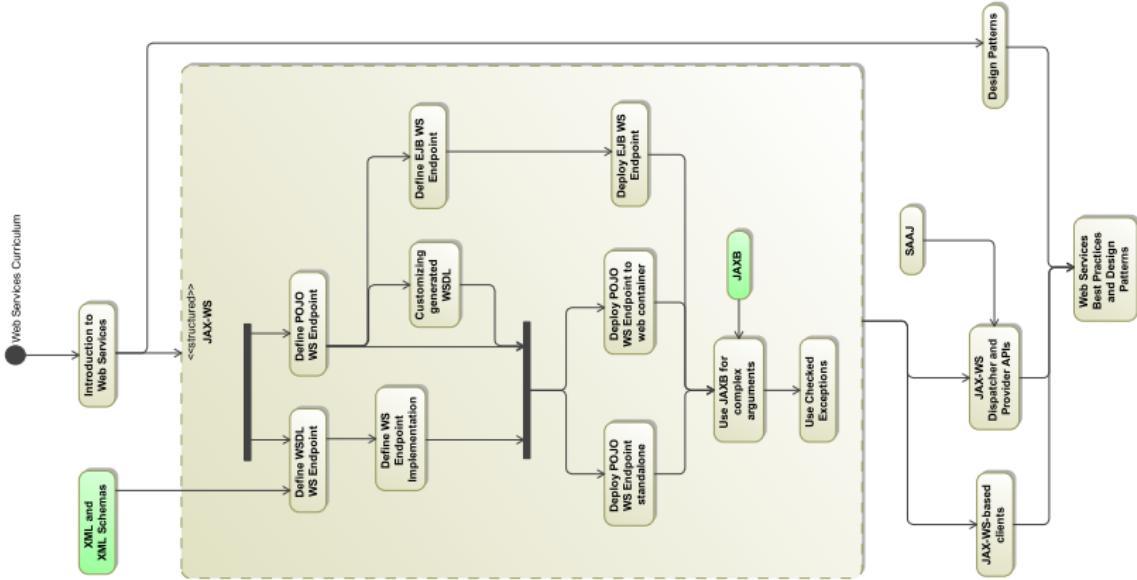
On completion of this course, you should be able to:

- Understand how to deploy web services that also leverage other Java Platform, Enterprise Edition (JavaEE) technologies, such as the web container infrastructure, Enterprise Java Beans, or the Java Persistence API.
 - Understand, explain, and apply best practices and design patterns when designing and deploying web services using either JAX-WS or JAX-RS technologies.

Course Map



Course Map – JAXWS



Topics Not Covered

Topics not covered:

- Object-oriented concepts, design and analysis, UML
- JavaTM programming language constructs
- Overview of the Java EE technologies

How Prepared Are You?

- Can you briefly describe the purpose of standard Java EE components?
 - Can you describe briefly the concepts distributed computing systems?
 - Are you comfortable with reading UML and can you create UML diagrams?

Introductions

- Name
- Company affiliation
- Title, function, and job responsibility
- Experience related to topics presented in this course
- Reasons for enrolling in this course
- Expectations for this course

How to Use Course Materials

To enable you to succeed in this course, these course materials contain a learning module that is composed of the following components:

- Goals – You should be able to accomplish the goals after finishing this course and meeting all of its objectives.
- Objectives – You should be able to accomplish the objectives after completing a portion of instructional content. Objectives support goals and can support other higher-level objectives.

How to Use Course Materials

(Continued)

- Lecture – The instructor presents information specific to the objective of the module. This information helps you learn the knowledge and skills necessary to succeed with the activities.
- Activities – The activities take on various forms, such as an exercise, self-check, discussion, and demonstration. Activities help you facilitate the mastery of an objective.

How to Use Course Materials

(Continued)

- Visual aids – The instructor might use several visual aids to convey a concept, such as a process, in a visual form. Visual aids commonly contain graphics, animation, and video.

Typographical Conventions

Courier New is used for the names of commands, files, directories, programming code, and on-screen computer output; for example:

- Use `ls -al` to list all files.
- `system%` You have mail.

Courier New is also used to indicate programming constructs, such as class names, methods, and keywords; for example:

- The `getServletInfo` method gets author information.

Typographical Conventions

(Continued)

Courier New Italics is used for variables and command-line placeholders that are replaced with a real name or value; for example:

- To delete a file, use the `rm filename` command.

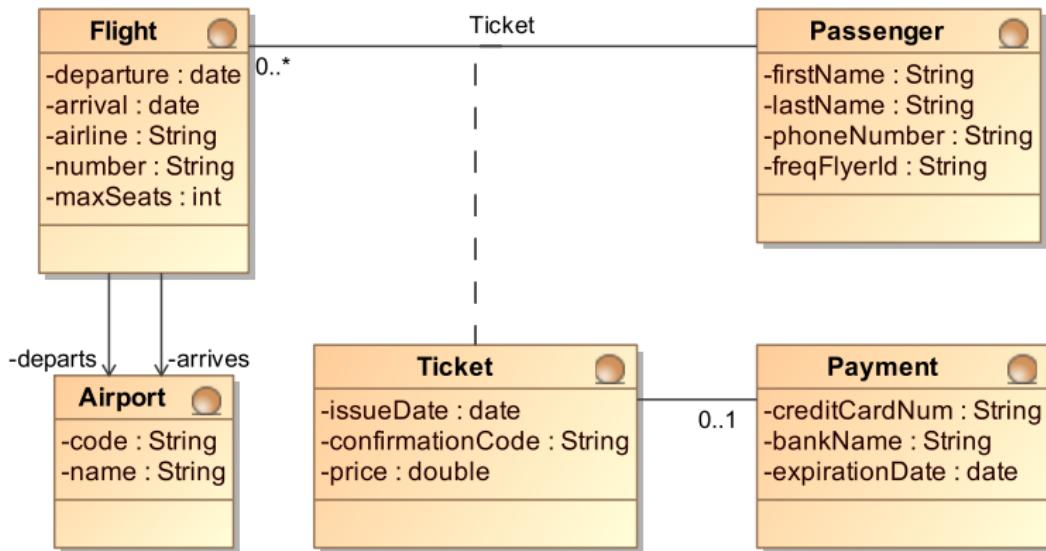
Traveller Project

Use Cases



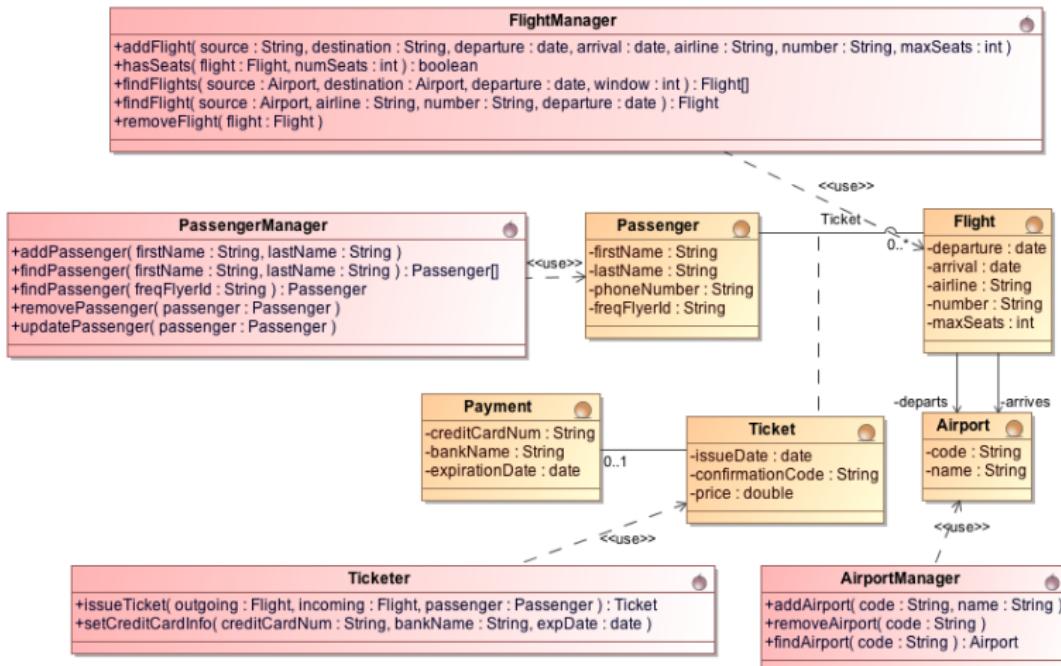
Traveller Project

Domain



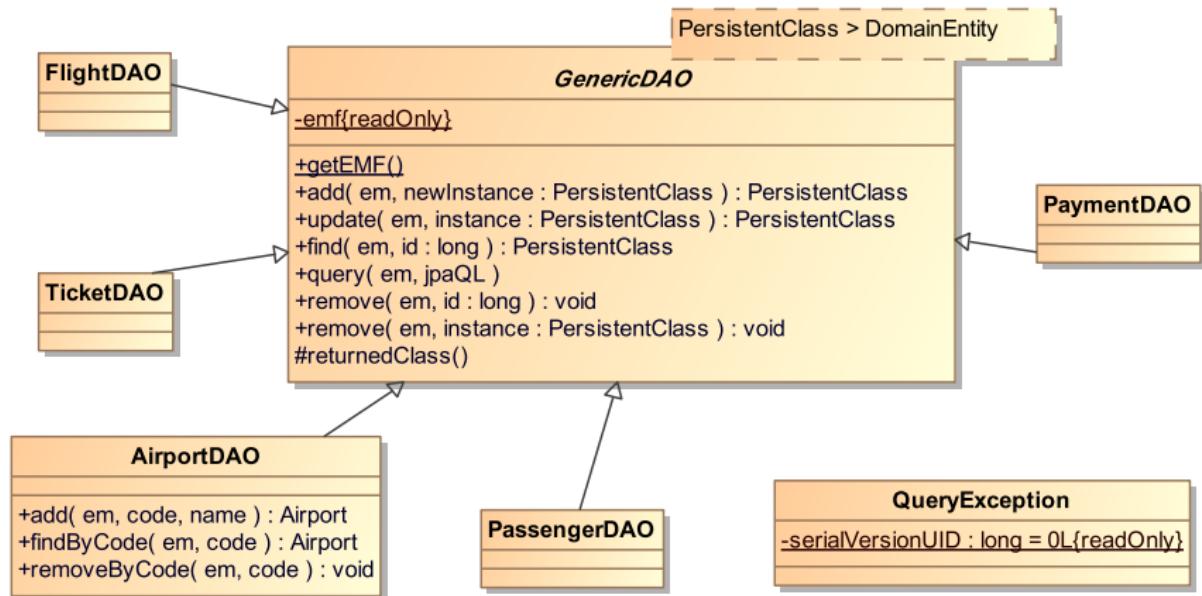
Traveller Project

Services



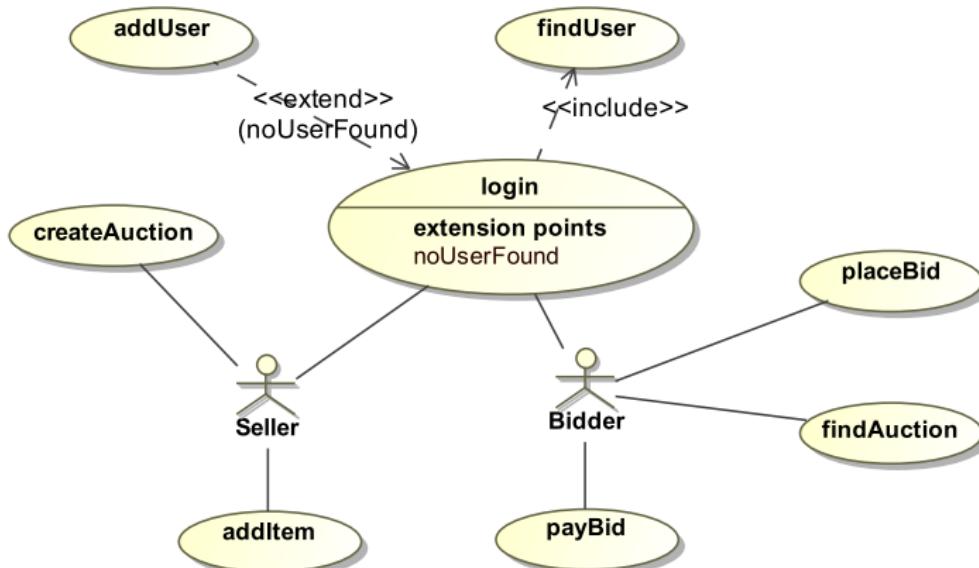
Traveller Project

DAOs



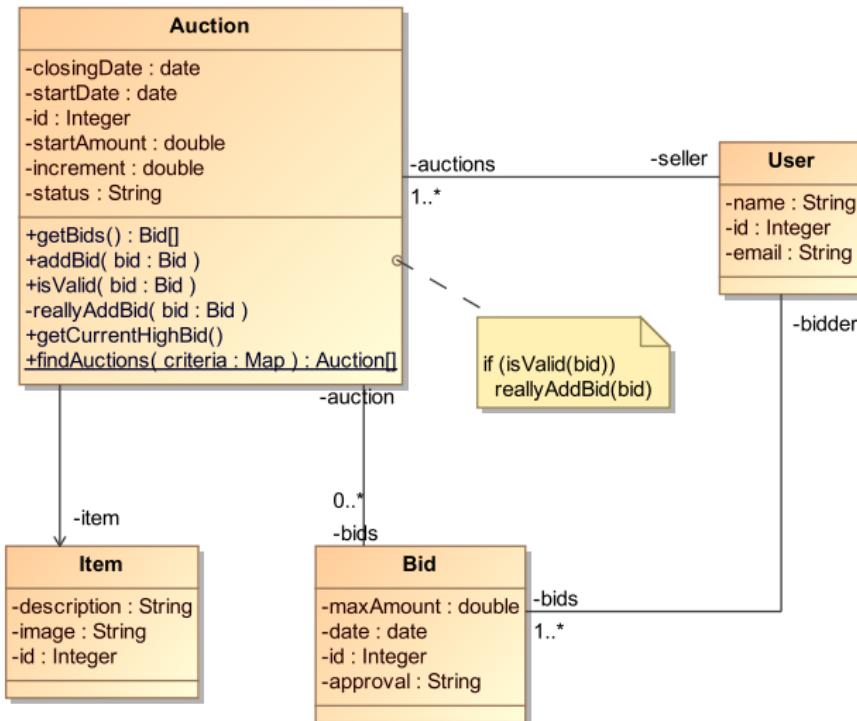
Auction Project

Use Cases



Auction Project

Domain





**Developing Web Services with Java™
Technology**

Module 1

Introduction to Web Services

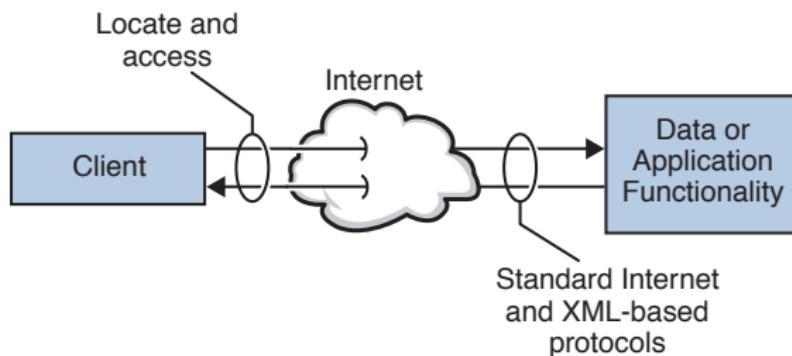
Objectives

On completion of this module, you should:

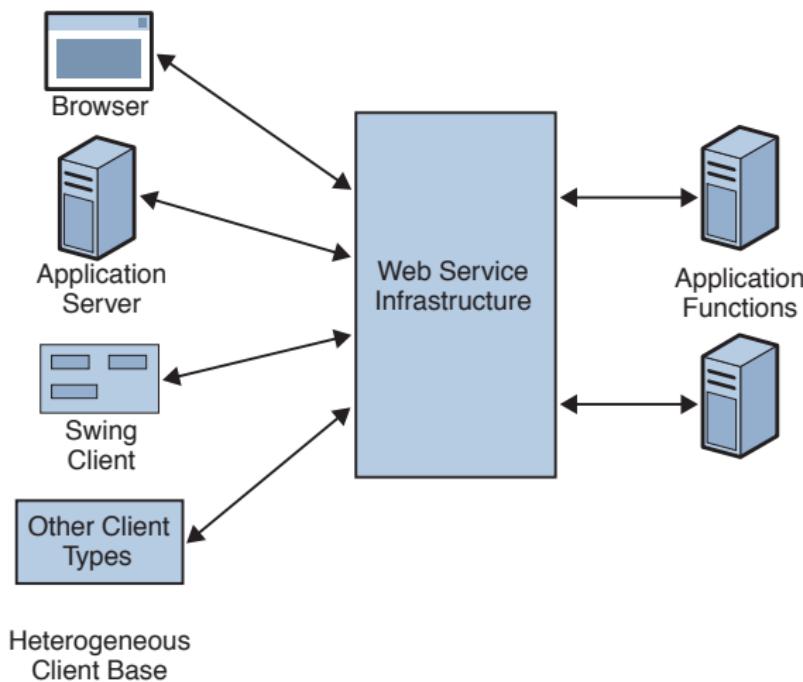
- Explore the need for web services
 - Define a web service and describe the motivation behind developing and using web services in business software
 - Describe the characteristics of a web service
 - Describe the two major approaches to developing web services
 - Describe the advantages of developing web services within a JavaEE container.

Web Services

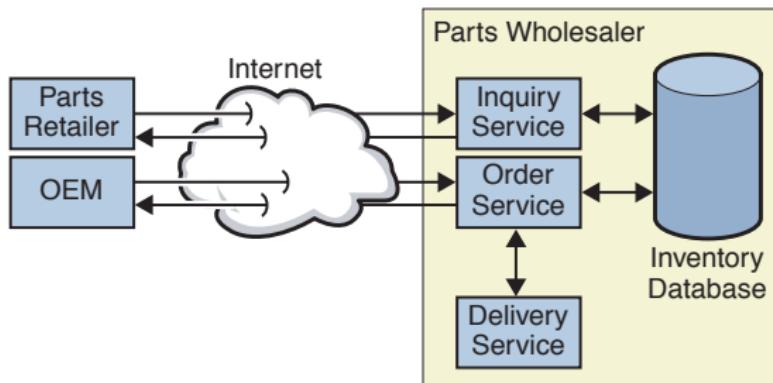
Web services introduce a suite of specifications and APIs for communicating between applications across the Internet.



Conceptual Model



Exposing Business Logic as Web Services



Characteristics of Web Services

- Loosely coupled
 - Self-describing
 - Published, located, and accessed over the Internet
 - Interoperable
 - Language and platform neutral
 - Standards based (Internet- and XML-based protocols)

Web Services vs Other Protocols

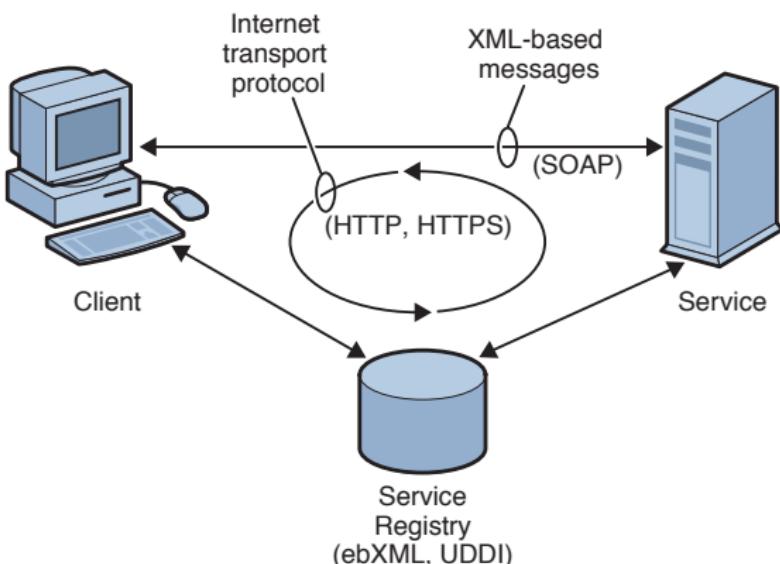
Traditional RPC

- Within enterprise
 - Tied to programming languages
 - Procedural
 - Specific transport
 - Tightly coupled
 - Firewall unfriendly
 - Efficient

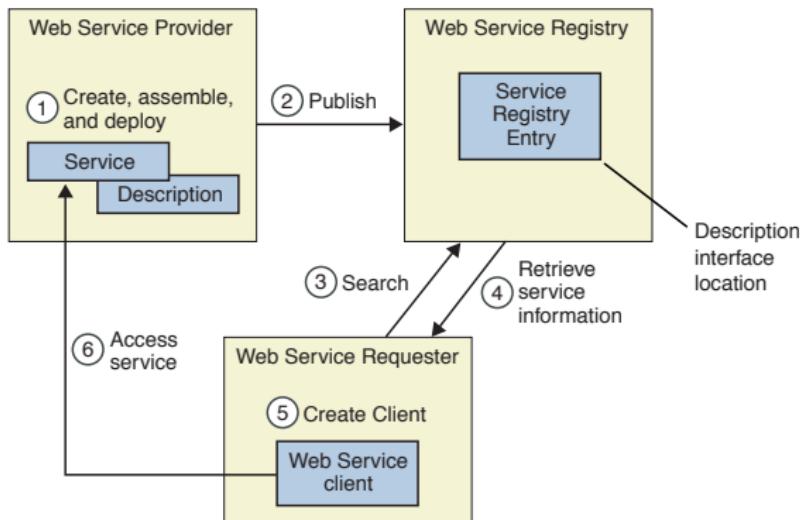
Web Service

- Public
 - Programming language independent
 - Message driven
 - Variable transports
 - Loosely coupled
 - Firewall friendly
 - Inefficient

Web Service Elements

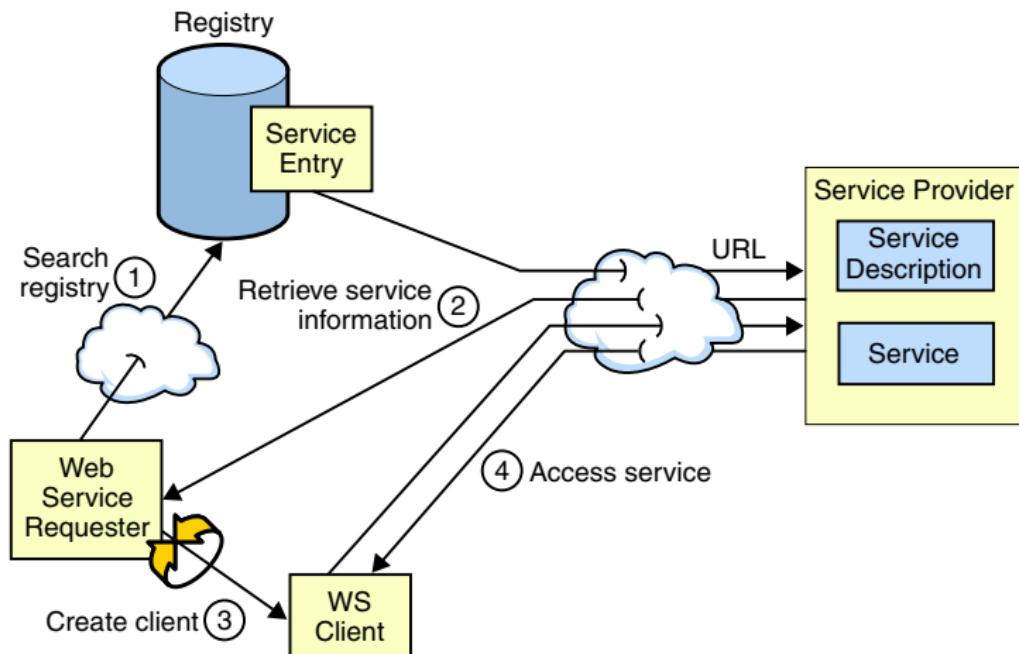


Web Service Life Cycle



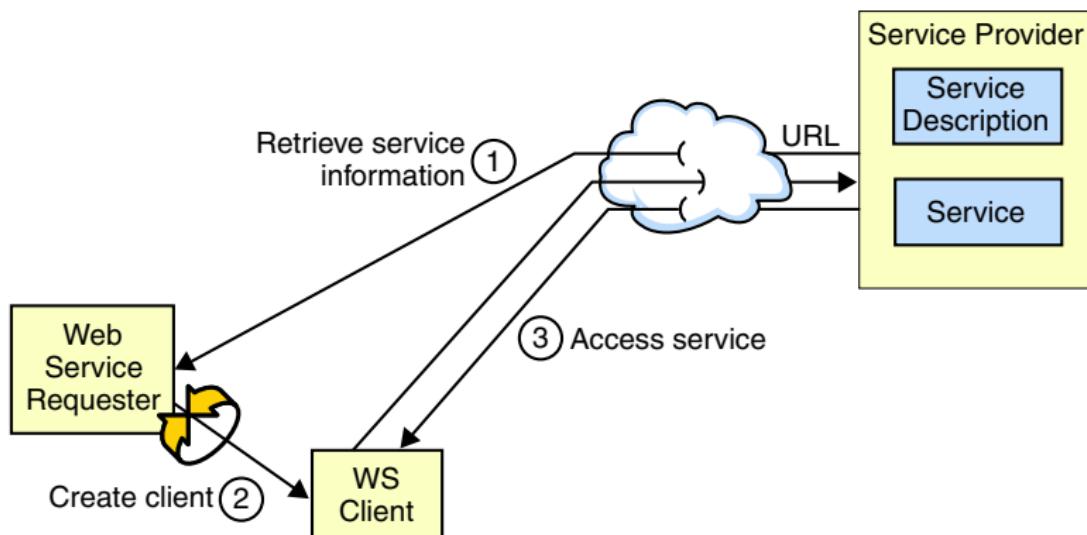
Interacting with a Web Service

Via Registry



Interacting with a Web Service

Direct



Major Web Services Models

There are two models for web service architecture:

- Simple Object Access Protocol (SOAP)-based
- Representational State Transfer (REST)-based
(also known as RESTful)

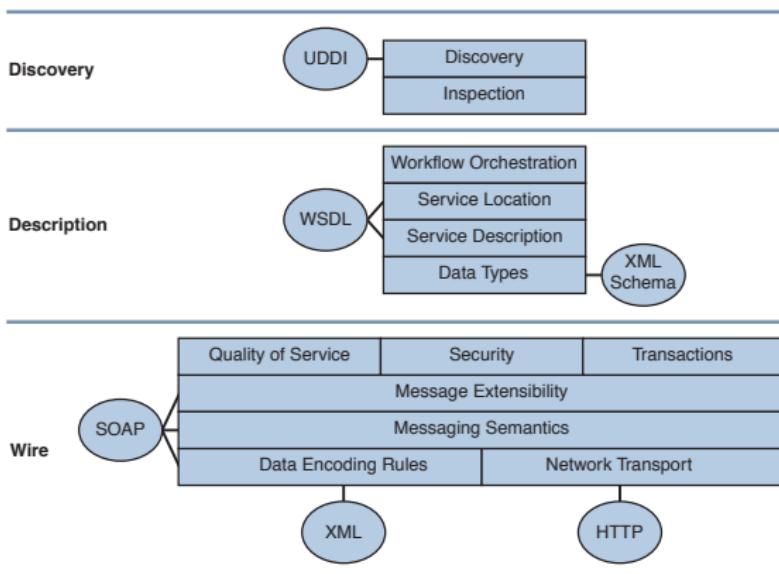
Common characteristics of the two models are:

- Use XML
- Rely on HTTP as a transport protocol

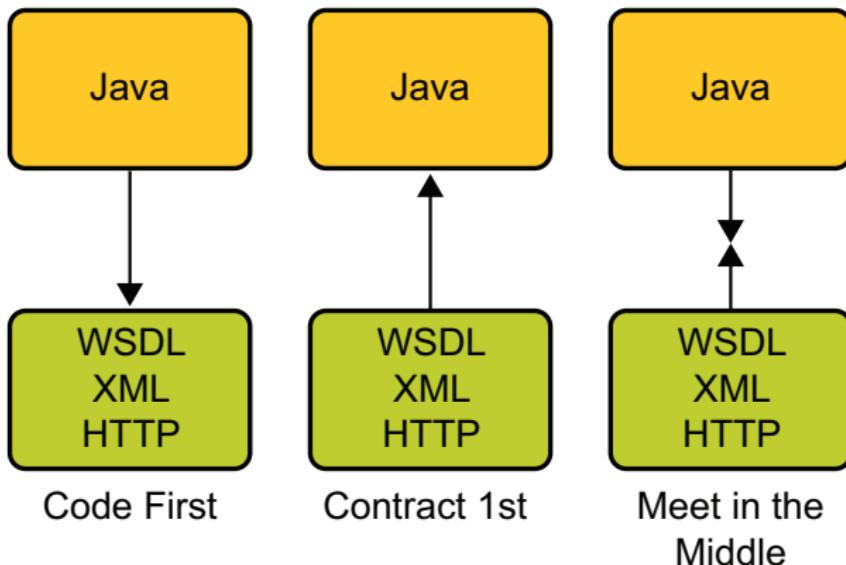
Key difference between the two models:

- SOAP-based web services have formal service definition (using WSDL)

An Interoperable Architecture



Development Approaches



Code First Approach

- Annotate your code.
- Deploy it in a container that supports JAX-WS, JAX-RS.
- The JAX-WS runtime will:
 - > Generate WSDL
 - > Translate SOAP request to a Java technology-based method invocation
 - > Translate method return into a SOAP response
- The JAX-RS runtime will:
 - > Translate HTTP request to a Java technology-base method invocation
 - > Translate method return into HTTP response

Contract First Approach

- “Compile” the WSDL for the service that you would like to deploy.
 - `wsimport` reads the WSDL and generates an interface for each `portType`.
- Create a class that implements each interface. The business logic of these classes implements your Web services.
- Deploy these Service Endpoint Implementation classes to a JAX-WS container.

Web Service Endpoints

Although almost any Java component could act as a web service, there are three primary models for a service endpoint as represented by a Java component:

- A stand-alone Java application can listen for SOAP messages and respond appropriately.
- The servlet service endpoint model – the most common endpoint type.
- The Enterprise JavaBeansTM(EJBTM) service endpoint model.

A Simplest Web Service

com.example.jaxws.server.AirportManager

```
1 @WebService
2 public class AirportManager {
3     public long
4         addAirport(String code, String name) {
5             return dao.add(null, code, name).getId();
6         }
7     private AirportDAO dao = new AirportDAO();
8 }
```

Simple POJO WS Client

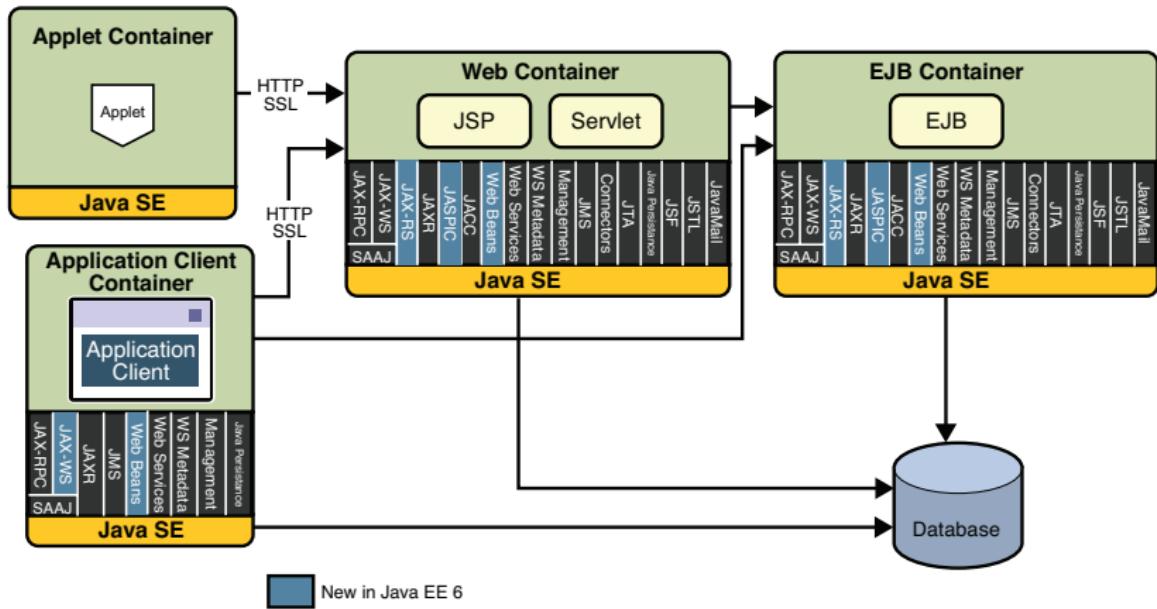
```
1  public class SimpleClient {  
2      public static void main(String[] args) {  
3          AirportManagerService service =  
4              new AirportManagerService();  
5          AirportManager port =  
6              service.getAirportManagerPort();  
7          java.lang.String code = "LGA";  
8          java.lang.String name = "New_York_LaGuardia";  
9          long result = port.addAirport(code, name);  
10         System.out.println("Result = "+result);  
11     }  
12 }
```

JavaEE Web Service Support

As of version 1.4, the Java Platform, Enterprise Edition (JavaEE) platform adds native support for web services:

- Both servlet and EJB containers can host web service components and service endpoints.
- All three Java 2 Platform, Enterprise Edition (J2EE™) containers offer client-side implementations of the web service APIs.
- JMS defines a standard for asynchronous messaging, which could play a role in SOAP interactions.

JavaEE 6 Specification



Session Beans as Service Endpoints

Web service endpoints implemented in session beans benefit from the EJB component model's support for enterprise features:

- Declarative, even implicit, transaction control
 - > transparent injection of JPA EntityManager and persistence context
 - Declarative Security Authentication and Authorization
 - Scalability through pooling
 - Availability through clustering



Developing Web Services with Java™ Technology

Module 2

Using JAX-WS

Objectives

On completion of this module, you should:

- Understand how to create web services using JAX-WS:
 - > Bottom-up, starting from Java classes
 - > Top-down, starting from WSDL descriptions
 - Understand how to deploy web services providers using JavaSE.
 - Understand how to create and deploy simple web services clients using JavaSE.

Overview of JAX-WS

Java API for XML Web Services (JAX-WS):

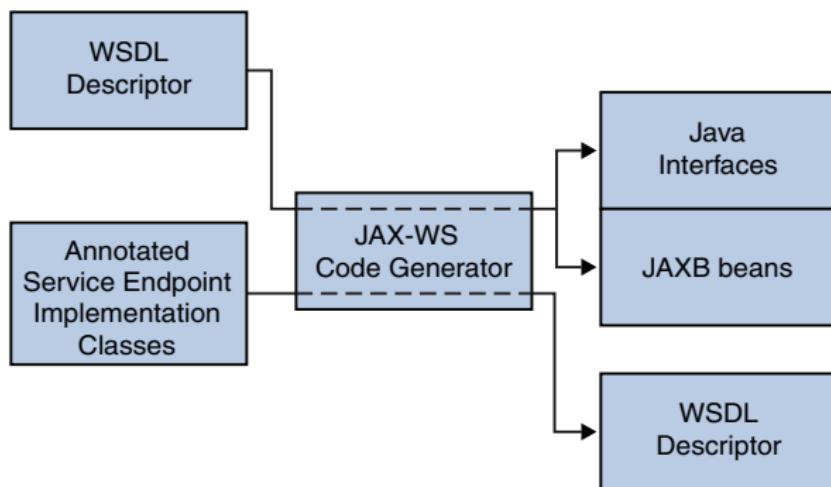
- Is a technology for building web services and clients that communicate using XML
 - Supports message-oriented and RPC-oriented web services
 - > even RESTful web services, with a bit of work...
 - Relies heavily on the use of annotations
 - Supports WS-I Basic Profile 1.1
 - Hides the complexity of SOAP interaction from the developer

JAX-WS Development Approaches

- Generate web service artifacts using the information contained in a WSDL file
- Create service endpoint interface (SEI) or value classes as Java source files, and then use them as inputs to generate the associated WSDL descriptor and other portable artifacts

In either case, JAX-WS generates the majority of the infrastructure code required by the service.

JAX-WS Artifacts



Advanced Features of JAX-WS

- Provides a flexible plug-in framework for message processing modules called handlers
 - Provides JAX-WS SOAP binding and JAX-WS XML/HTTP binding
 - Provides Dispatch API for those web service client applications which work at the XML message level
 - Provides support for asynchronous interactions
 - Provides support for stateful interactions

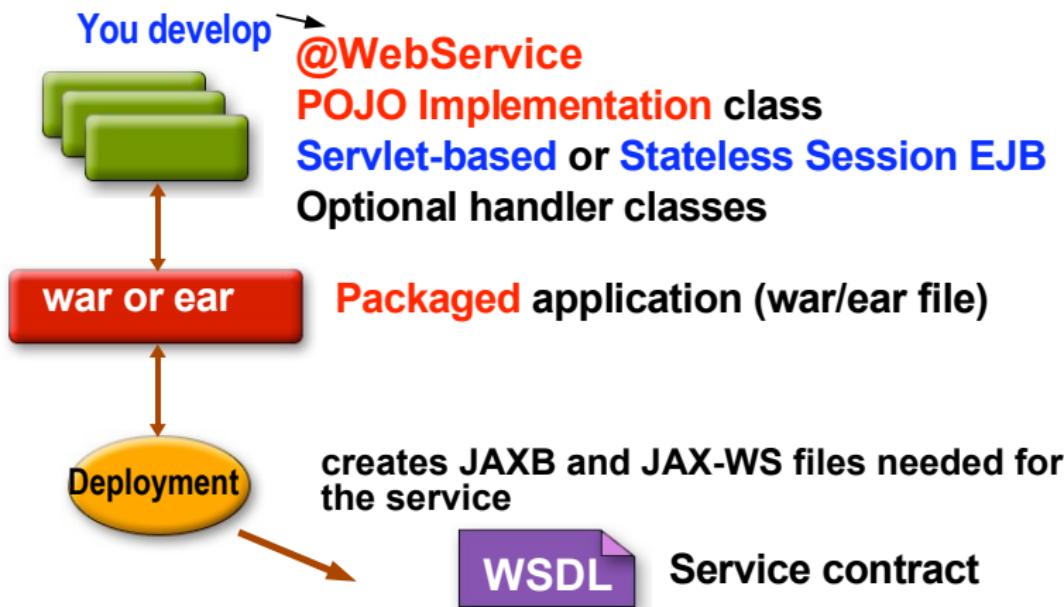
Creating a Web Service Using JAX-WS

Bottom-Up Approach

Starting from a Java class:

- For creating a web service that exposes the functionality of an existing application or creating from scratch.
 - When you start with a suite of application components, generate the necessary web service descriptive and infrastructure files based on the functionality provided by the existing components.

Starting From a Java Class



A Simple Service: AirportManager

com.example.standalone.AirportManager

```
1 public class AirportManager {  
2     public long  
3         addAirport(String code, String name) {  
4             return dao.add(null, code, name).getId();  
5         }  
6     private AirportDAO dao = new AirportDAO();  
7 }
```

A Simplest Web Service

com.example.jaxws.server.AirportManager

```
1 @WebService
2 public class AirportManager {
3     public long
4         addAirport(String code, String name) {
5             return dao.add(null, code, name).getId();
6         }
7     private AirportDAO dao = new AirportDAO();
8 }
```

JAX-WS Requirements

Requirements on the Java Class

- Must be annotated with `javax.ws.WebService`
 - Must not be declared **final**
 - Must not be **abstract**
 - Must have a default public constructor
 - Must not have a `finalize` method

JAX-WS Requirements

Requirements on Web Service Methods

- Must be **public**
By default, every public method in the class will be part of the web service.
 - must not be **static** or **final**
 - must have JAXB-compatible parameters and return types
 - > Parameters and return types must not implement the `java.rmi.Remote` interface.

Generating Portable Artifacts

WSDL Description and Supporting Files

- JAX-WS might need additional artifacts to support the web service provider:
 - > JAXB classes, to represent SOAP messages
 - > A WSDL file, for clients to access the web service
- These can be generated using `apt` or `wsgen`

```
apt [-d outputDir] sourceFile ...
```

```
wsgen [-d outputDir] classFile ...
```

- > JAX-WS can deliver WSDL descriptions dynamically:

```
http://host:port/path/to/service?WSDL
```

Generated WSDL

Default WSDL Description

```
1 <message name="addAirport">
2   <part name="parameters" element="tns:addAirport"/>
3 </message>
4 <message name="addAirportResponse">
5   <part name="parameters" element="tns:addAirportResponse"/>
6 </message>
7 <portType name="AirportManager">
8   <operation name="addAirport">
9     <input message="tns:addAirport"/>
10    <output message="tns:addAirportResponse"/>
11   </operation>
12 </portType>
```

Generated XML Schema

Default XML Schema

```
1 <xs:element name="addAirport" type="tns:addAirport"/>
2 <xs:element name="addAirportResponse"
3     type="tns:addAirportResponse"/>
4 <xs:complexType name="addAirport">
5     <xs:sequence>
6         <xs:element name="arg0" type="xs:string" minOccurs="0"/>
7         <xs:element name="arg1" type="xs:string" minOccurs="0"/>
8     </xs:sequence>
9 </xs:complexType>
10 <xs:complexType name="addAirportResponse">
11     <xs:sequence>
12         <xs:element name="return" type="xs:long"/>
13     </xs:sequence>
14 </xs:complexType>
```

Sample SOAP Request

```
1 <S:Envelope xmlns:ser="http://server.jaxws.example.com/"  
2   xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">  
3   <S:Header/>  
4   <S:Body>  
5     <ser:addAirport>  
6       <arg0>JFK</arg0>  
7       <arg1>New York Kennedy Airport</arg1>  
8     </ser:addAirport>  
9   </S:Body>  
10  </S:Envelope>
```

Sample SOAP Response

```
1 <S:Envelope
2   xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
3     <S:Body>
4       <ns2:addAirportResponse
5         xmlns:ns2="http://server.jaxws.example.com/">
6           <return>1</return>
7         </ns2:addAirportResponse>
8       </S:Body>
9     </S:Envelope>
```

Raw SOAP/HTTP Request

```
1 POST http://localhost:8080/airportManager HTTP/1.1
2 Accept-Encoding: gzip,deflate
3 Content-Type: text/xml; charset=UTF-8
4 SOAPAction: ""
5 Content-Length: 346
6
7 <S:Envelope xmlns:ser="http://server.jaxws.example.com/"
8   xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
9   <S:Header/>
10  <S:Body>
11    <ser:addAirport>
12      <arg0>JFK</arg0>
13      <arg1>New York Kennedy Airport</arg1>
14    </ser:addAirport>
15  </S:Body>
16 </S:Envelope>
```

Raw SOAP/HTTP Response

```
1 HTTP/1.1 200 OK
2 Transfer-encoding: chunked
3 Content-type: text/xml; charset=utf-8
4
5 <S:Envelope
6   xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
7   <S:Body>
8     <ns2:addAirportResponse
9       xmlns:ns2="http://server.jaxws.example.com/">
10      <return>1</return>
11      </ns2:addAirportResponse>
12    </S:Body>
13  </S:Envelope>
```

Custom WSDL Description

```
1 @WebService(portName="AirportMgr",
2             serviceName="Managers")
3 public class NamedAirportManager {
4     @WebMethod(operationName="add")
5     public long
6         addAirport(@WebParam(name="code") String code,
7                    @WebParam(name="name") String name) {
8             return dao.add(null, code, name).getId();
9         }
10    private AirportDAO dao = new AirportDAO();
11 }
```

Generated WSDL

Customized WSDL Description ...

```
1 <portType name="NamedAirportManager">
2   <operation name="add">
3     <input message="tns:add"/>
4     <output message="tns:addResponse"/>
5   </operation>
6 </portType>
7 <!-- ... -->
8 <service name="Managers">
9   <port name="AirportMgr" binding="tns:AirportMgrBinding">
10    <soap:address
11      location="http://localhost:8080/namedManager"/>
12    </port>
13 </service>
```

... and the Custom XML Schema

```
1 <xs:element name="add" type="tns:add"/>
2 <xs:element name="addResponse" type="tns:addResponse"/>
3 <xs:complexType name="add">
4   <xs:sequence>
5     <xs:element name="code" type="xs:string" minOccurs="0"/>
6     <xs:element name="name" type="xs:string" minOccurs="0"/>
7   </xs:sequence>
8 </xs:complexType>
9 <xs:complexType name="addResponse">
10   <xs:sequence>
11     <xs:element name="return" type="xs:long"/>
12   </xs:sequence>
13 </xs:complexType>
```

Custom WSDL Description

Overloaded Methods

Web services require unique names for each method:

```
1 @WebService
2 public class BetterAirportManager {
3     @WebMethod(operationName="removeById")
4     public void removeAirport(long id) {
5     }
6     @WebMethod(operationName="removeByCode")
7     public void removeAirport(String code) {
8     }
9     // ...
10 }
```

Generated WSDL

“Overloaded” Operations

```
1 <portType name="BetterAirportManager">
2   <!-- ... -->
3   <operation name="removeById">
4     <input message="tns:removeById"/>
5     <output message="tns:removeByIdResponse"/>
6   </operation>
7   <operation name="removeByCode">
8     <input message="tns:removeByCode"/>
9     <output message="tns:removeByCodeResponse"/>
10    </operation>
11 </portType>
```

Custom WSDL Description

Default XML Namespace

- default XML namespace is computed:

```
package com.example.security.server;
```

produces

```
1 <definitions  
2   targetNamespace=  
3     "http://server.security.example.com/"  
4   ... />
```

Custom WSDL Description

Custom XML Namespace

- Set targetNamespace attribute to @WebService

```
1 @WebService(  
2     targetNamespace="urn://com.example.managerNS")  
3 public class NamespacedAirportManager {
```

produces

```
1 <definitions  
2     targetNamespace="urn://com.example.managerNS/"  
3     ... />
```

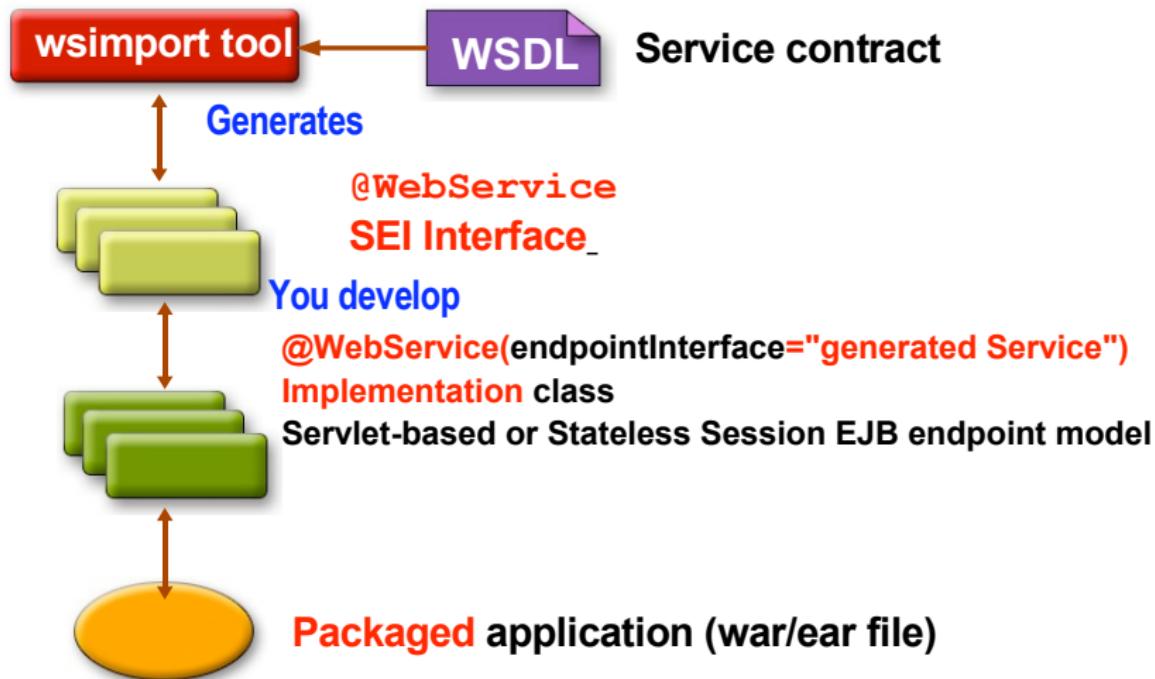
Creating a Web Service Using JAX-WS

Top-Down Approach

Use the WSDL-to-Java development approach when you create a web service that is not based on existing application components:

- Create or obtain a service description file that specifies the characteristics of the service
 - Generate a service endpoint interface
 - Implement the required web service application components

Starting From a WSDL Description



Structure of a WSDL file

<definitions>: Root WSDL Element

<types>: What data types will be transmitted?

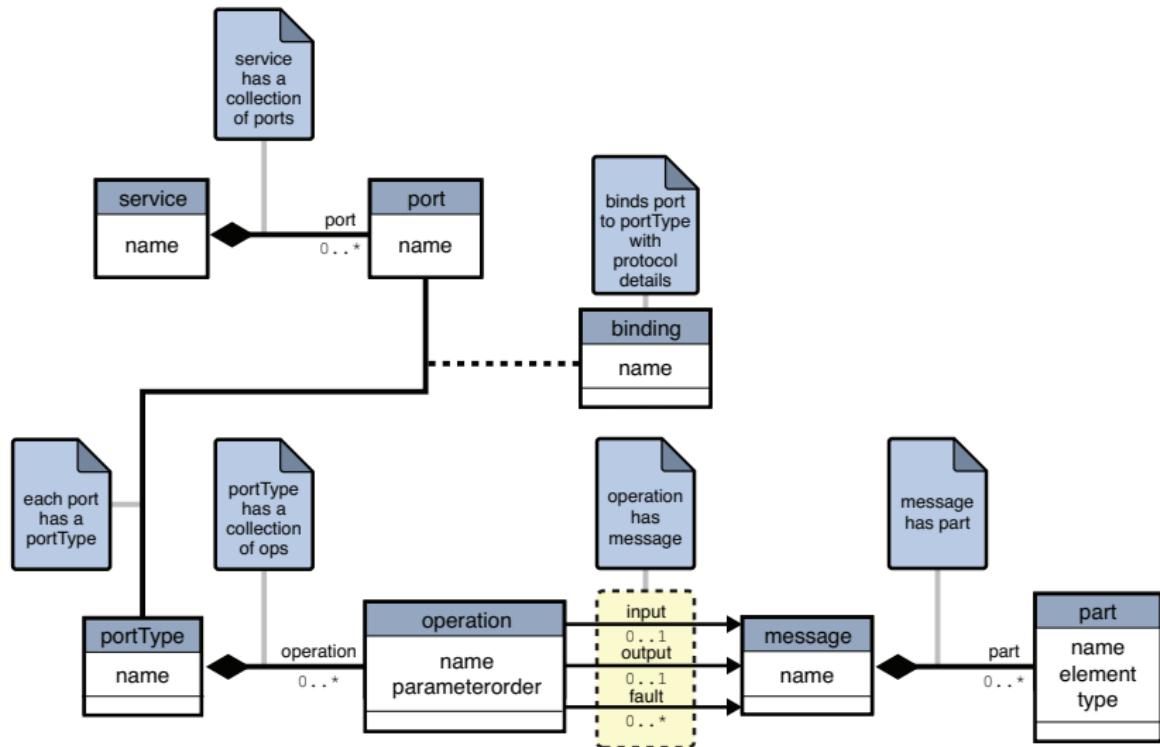
<message>: What exact information is expected?

<portType>: What operations (functions) will be supported?

<binding>: How will the messages be transmitted on the wire?
What SOAP-specific details are there?

<service>: Define the collection of ports that make up the service and where is the service located?

Definition of a Service Using WSDL



Creating WSDL

WSDL Description (1 of 2)

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <definitions name="PassengerManagerPort.wsdl"
3   xmlns="http://schemas.xmlsoap.org/wsdl/"
4   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
5   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
6   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
7   xmlns:tns="urn://Traveller/"
8   targetNamespace="urn://Traveller/">
9 <types>
10 <xsd:schema>
11   <xsd:import namespace="urn://Traveller/"
12     schemaLocation="PassengerManagerSchema.xsd"/>
13 </xsd:schema>
14 </types>
```

Creating WSDL

WSDL Description (2 of 2)

```
15 <message name="addPassengerRequest">
16   <part name="params" element="tns:addPassenger"/>
17 </message>
18 <message name="addPassengerResp">
19   <part name="params" element="tns:addPassengerResponse"/>
20 </message>
21 <portType name="PassengerManager">
22   <operation name="addPassenger">
23     <input name="in1" message="tns:addPassengerRequest"/>
24     <output name="out1" message="tns:addPassengerResp"/>
25   </operation>
26 </portType>
27 </definitions>
```

Creating WSDL

WSDL Schema

```
1 <xsd:schema id="PassengerManagerSchema.xsd"
2   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3   xmlns:tns="urn://Traveller/"
4   elementFormDefault="qualified"
5   targetNamespace="urn://Traveller/">
6 <xsd:element name="addPassenger">
7   <xsd:complexType>
8     <xsd:sequence>
9       <xsd:element name="firstName" type="xsd:string"/>
10      <xsd:element name="lastName" type="xsd:string"/>
11    </xsd:sequence>
12  </xsd:complexType>
13 </xsd:element>
```

Creating WSDL

WSDL Bindings

```
1 <binding name="binding" type="tns:PassengerManager">
2   <soap:binding style="document"
3     transport="http://schemas.xmlsoap.org/soap/http"/>
4   <operation name="addPassenger">
5     <soap:operation/>
6     <input><soap:body use="literal"/></input>
7     <output><soap:body use="literal"/></output>
8   </operation>
9 </binding>
10 <service name="PassengerManagerService">
11   <port name="PassengerManager" binding="tns:binding">
12     <soap:address
13       location="http://localhost:8080/passengerManager"/>
14   </port>
15 </service>
```

Generating Implementation Artifacts

The `wsimport` tool is a command-line tool to import a WSDL and to generate a SEI interface.

- The generated interface can be either:
 - > Implemented on the server to build a web service, or
 - > Used on the client to invoke the web service
- Some of the important command-line switches include:
 - > `-d` - location of generated class files
 - > `-s` - location of generated source files

```
wsimport http://host:port/path?wsdl  
wsimport PassengerManagerService.wsdl
```

Generated Java SEI

```
1 package com.example.generated;
2 @WebService(name = "PassengerManager",
3             targetNamespace = "urn://Traveller/")
4 public interface PassengerManager {
5     @WebMethod
6     public long addPassenger(
7         @WebParam(name = "firstName",
8                 targetNamespace = "urn://Traveller/")
9         String firstName,
10        @WebParam(name = "lastName",
11                  targetNamespace = "urn://Traveller/")
12        String lastName);
13 }
```

Service Implementation Class

```
1 @WebService(  
2     endpointInterface=  
3     "com.example.generated.PassengerManager")  
4 public class PassengerManager  
5 implements com.example.generated.PassengerManager {  
6     public long  
7         addPassenger(String firstName, String lastName) {  
8             Passenger newPassenger =  
9                 new Passenger(firstName, lastName, null, null);  
10            return dao.add(null, newPassenger).getId();  
11        }  
12        private PassengerDAO dao = new PassengerDAO();  
13    }
```

Customizing Generated Code

- Just as in the Java-to-WSDL scenario, one can customize:
 - the package name where generated classes will be placed
 - the name for the type generated for a WS port
 - the name for the method generated for a WS operation
 - developer can introduce overloading in the generated code
- There are two ways to specify this customization detail:
 - embedded declarations within the WSDL file
 - separate customization XML file

Embedded Customization

Custom Package Name

```
1 <definitions name="CustomPassengerManagerService.wsdl"
2   xmlns="http://schemas.xmlsoap.org/wsdl/">
3   <jaxws:bindings
4     xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
5     <jaxws:package name="com.example.custom"/>
6   </jaxws:bindings>
7   <service name="CustomPassengerManagerService">
8     <port name="CustomPassengerManager"
9       binding="tns:CustomPassengerManagerBinding">
10      <soap:address
11        location="http://localhost:8080/customManager"/>
12    </port>
13  </service>
```

Embedded Customization

Custom Class Name

```
1 <definitions name="CustomPassengerManagerPort.wsdl"
2     xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
3     xmlns="http://schemas.xmlsoap.org/wsdl/">
4     <!-- types -->
5     <portType name="CustomPassengerManager">
6         <jaxws:bindings>
7             <jaxws:class name="CustomManager"/>
8         </jaxws:bindings>
9         <!-- operations ... -->
10        </portType>
11    </definitions>
```

Embedded Customization

Custom Method Name

```
1 <definitions name="CustomPassengerManagerPort.wsdl"
2   xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
3   xmlns="http://schemas.xmlsoap.org/wsdl/">
4   <!-- types -->
5   <portType name="CustomPassengerManager">
6     <operation name="addPassenger">
7       <jaxws:bindings>
8         <jaxws:method name="add"/>
9       </jaxws:bindings>
10      <!-- messages ... -->
11      </operation>
12    </portType>
13  </definitions>
```

Generated Code

From Customized WSDL

```
1 package com.example.custom;
2 @WebService(name = "CustomPassengerManager",
3             targetNamespace = "urn://Traveller")
4 public interface CustomManager {
5     @WebMethod(operationName = "addPassenger")
6     public long add(
7         @WebParam(name = "firstName",
8                 targetNamespace = "urn://Traveller")
9         String firstName,
10        @WebParam(name = "lastName",
11                  targetNamespace = "urn://Traveller")
12        String lastName);
13 }
```

Schema Validation

- JAX-WS won't validate web service calls against XML schema constraints unless requested:

```
1  @SchemaValidation
2  @WebService(
3      endpointInterface=
4          "com.example.generated.PassengerManager")
5  public class PassengerManager
6  implements com.example.generated.PassengerManager {
7      public long
8          addPassenger(String firstName, String lastName) {
9              Passenger newPassenger =
10                 new Passenger(firstName, lastName, null, null);
11                 return dao.add(null, newPassenger).getId();
12 }
```

Comparing Development Approaches

- Each of the development approaches has benefits and costs, and is best suited to specific scenarios.
 - You should decide on a specific development approach based on the needs of the development effort.

Strong Typing for Web Services

A benefit of the top-down approach

Fully worked WSDL can specify data and message types to fine granularity, using the full power of XML schema and allowing clients as well as servers to validate message content locally as follows:

- When you generate Java from WSDL, you can map strong XML types to Java and validate those types using generated code.
 - When you generate WSDL from Java, the generated types are weaker.

Benefits of a Bottom-Up Approach

When you provide a web service interface to an existing enterprise application, you should work from Java code:

- You can use the quickest development path.
 - It is a natural approach, especially when business logic has already been implemented.
 - You can map existing domain models directly to WSDL with little effort.
 - You can re-use a service facade as a mediator to the domain logic for other types of applications and clients.

Benefits of a Top-Down Approach

Working from WSDL provides the following advantages:

- WSDL is better suited when the service developer is not also the author of the service.
- The strong typing is shared using the WSDL file itself.
- Interoperability is easier to achieve.
- Serializable types are expressed first in the XML schema and then mapped to one or more program languages and object models.
- Client-side validation is easier to implement.

Deploying POJO WS Providers

There are two ways to deploy a POJO Web Service Provider:

- Use the Endpoint machinery in JAX-WS on JavaSE
 - Rely on a web container that includes support for JAX-WS

A Simplest Web Service (Recap)

com.example.jaxws.server.AirportManager

```
1 @WebService
2 public class AirportManager {
3     public long
4         addAirport(String code, String name) {
5             return dao.add(null, code, name).getId();
6         }
7     private AirportDAO dao = new AirportDAO();
8 }
```

Simple Standalone Server

```
1 public class AirportManager {  
2     // ...  
3     static public void main(String[] args) {  
4         String url =  
5             "http://localhost:8080/airportManager";  
6         if (args.length > 0)  
7             url = args[1];  
8         AirportManager manager = new AirportManager();  
9         Endpoint endpoint =  
10            Endpoint.publish(url, manager);  
11    }  
12 }
```

Finer Control over Standalone Server

```
1 public class FancyServer {  
2     public static  
3         void main( String[] args ) throws Exception {  
4             HttpServer server =  
5                 HttpServer.create(new InetSocketAddress(8080),10);  
6             Executor executor = Executors.newFixedThreadPool(10);  
7             server.setExecutor(executor);  
8             HttpContext context =  
9                 server.createContext("/fancyServer");  
10            AirportManager manager = new AirportManager();  
11            Endpoint endpoint = Endpoint.create(manager);  
12            endpoint.publish(context);  
13            server.start();  
14        }  
15    }
```

Viewing SOAP Messages

Sometimes, it helps to dump all SOAP messages exchanged by the server to standard out:

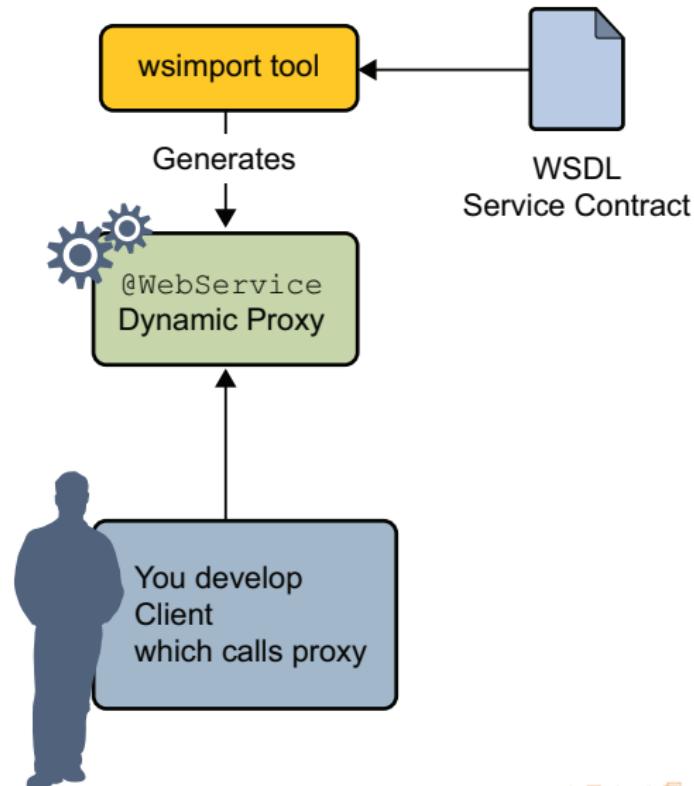
- JAX-WS includes a generic framework that can be used for this: JAX-WS Handlers.
 - The system property

`com.sun.xml.ws.transport.http.HttpAdapter.dump`
when set to **true** will prompt JAX-WS to do so.

Testing a Web Service

- In Glassfish,
`http://hostname/webService?Tester` generates a web page that can be used to test the web service specified by the URI.
 - This only works for operations whose arguments are simple types.
- the SoapUI plugin for NetBeans/Eclipse will generate test suites for all operations in a web service
- a custom web services client

Developing Web Service Clients



Simple POJO WS Client

```
1  public class SimpleClient {  
2      public static void main(String[] args) {  
3          AirportManagerService service =  
4              new AirportManagerService();  
5          AirportManager port =  
6              service.getAirportManagerPort();  
7          java.lang.String code = "LGA";  
8          java.lang.String name = "New_York_LaGuardia";  
9          long result = port.addAirport(code, name);  
10         System.out.println("Result = "+result);  
11     }  
12 }
```



Developing Web Services with Java™ Technology

Module 3

SOAP and WSDL

Objectives

On completion of this module, you should be able to:

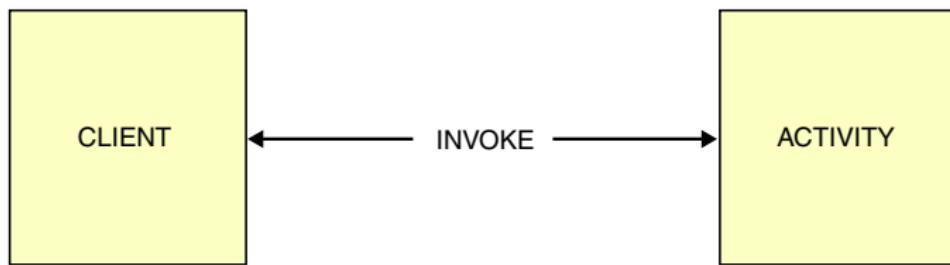
- Understand the basic structure of a SOAP message, and how it is encapsulated by transports
 - Understand how WSDL defines a web service, including its message representation and transport mechanism
 - Understand the different styles of SOAP messages that a web service can use, and their trade-offs
 - Customize a web service to control the style of SOAP message that that web service will use

Why Use SOAP?

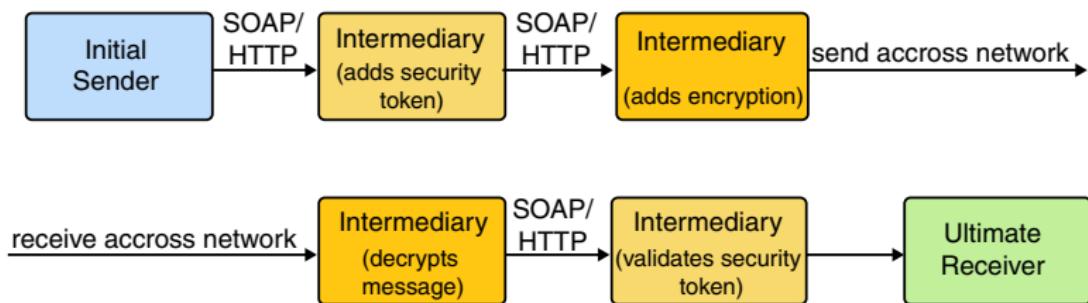
The web services specification sets out to define an interoperable, platform-independent means for component interaction. Among their requirements:

- decouple message representation from transport mechanisms
 - support extensible frameworks

A Simple Web Service Interaction

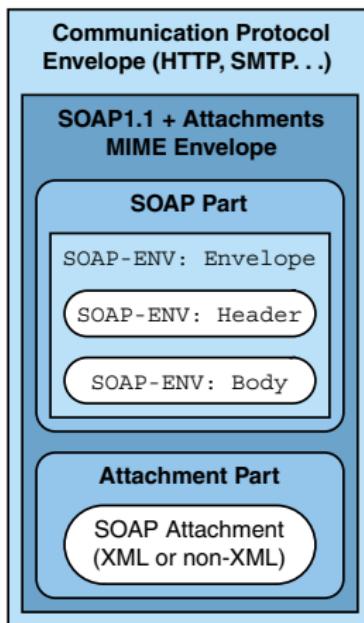


Extensible Web Service Interactions



Extensible Message Representation

Simple Object Access Protocol



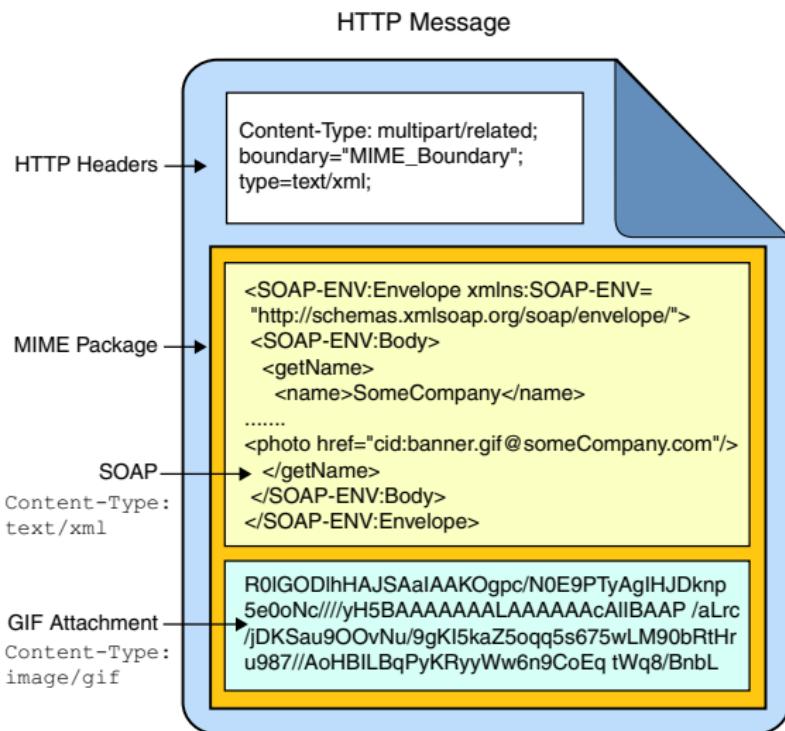
Sample SOAP Request

```
1 <S:Envelope xmlns:ser="http://server.jaxws.example.com/">
2   xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
3     <S:Header/>
4     <S:Body>
5       <ser:addAirport>
6         <arg0>JFK</arg0>
7         <arg1>New York Kennedy Airport</arg1>
8       </ser:addAirport>
9     </S:Body>
10    </S:Envelope>
```

Sample SOAP Response

```
1 <S:Envelope  
2   xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">  
3     <S:Body>  
4       <ns2:addAirportResponse  
5         xmlns:ns2="http://server.jaxws.example.com/">  
6           <return>1</return>  
7           </ns2:addAirportResponse>  
8     </S:Body>  
9   </S:Envelope>
```

SOAP over HTTP



Raw SOAP/HTTP Request

```
1 POST http://localhost:8080/airportManager HTTP/1.1
2 Accept-Encoding: gzip,deflate
3 Content-Type: text/xml; charset=UTF-8
4 SOAPAction: ""
5 Content-Length: 346
6
7 <S:Envelope xmlns:ser="http://server.jaxws.example.com/""
8   xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
9     <S:Header/>
10    <S:Body>
11      <ser:addAirport>
12        <arg0>JFK</arg0>
13        <arg1>New York Kennedy Airport</arg1>
14      </ser:addAirport>
15    </S:Body>
16  </S:Envelope>
```

Raw SOAP/HTTP Response

```
1 HTTP/1.1 200 OK
2 Transfer-encoding: chunked
3 Content-type: text/xml; charset=utf-8
4
5 <S:Envelope
6   xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
7   <S:Body>
8     <ns2:addAirportResponse
9       xmlns:ns2="http://server.jaxws.example.com/">
10      <return>1</return>
11      </ns2:addAirportResponse>
12    </S:Body>
13  </S:Envelope>
```

Structure of a WSDL file

<definitions>: Root WSDL Element

<types>: What data types will be transmitted?

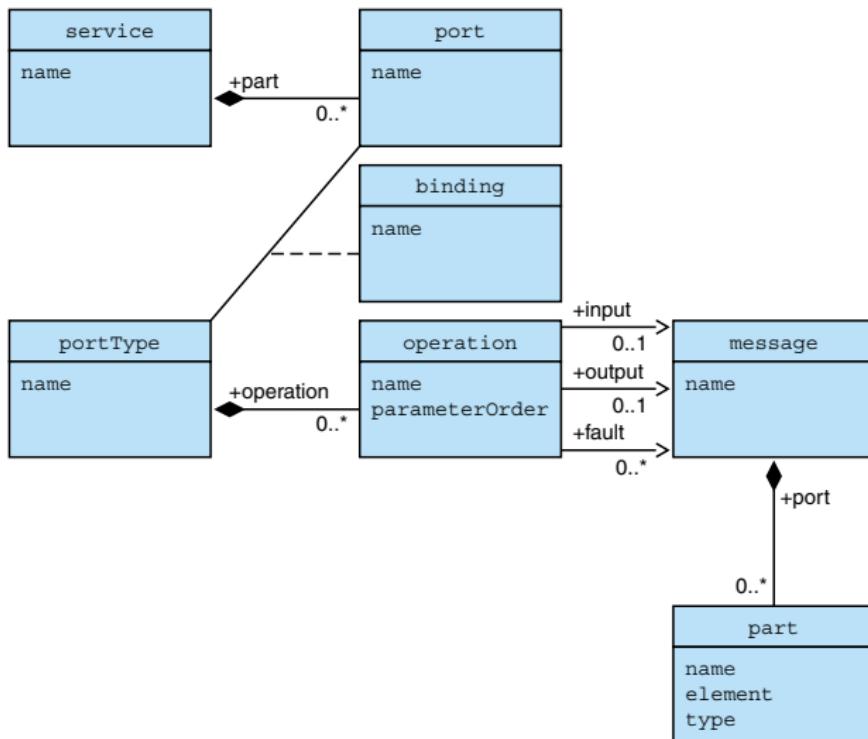
<message>: What exact information is expected?

<portType>: What operations (functions) will be supported?

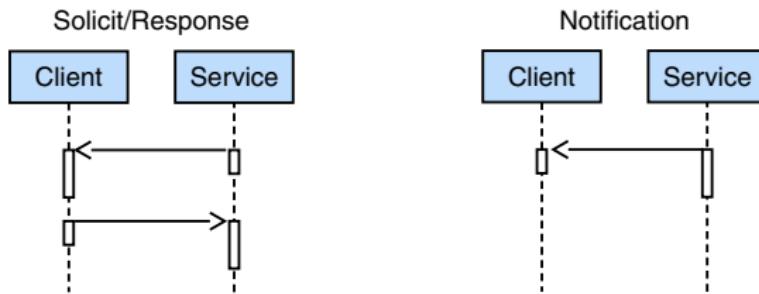
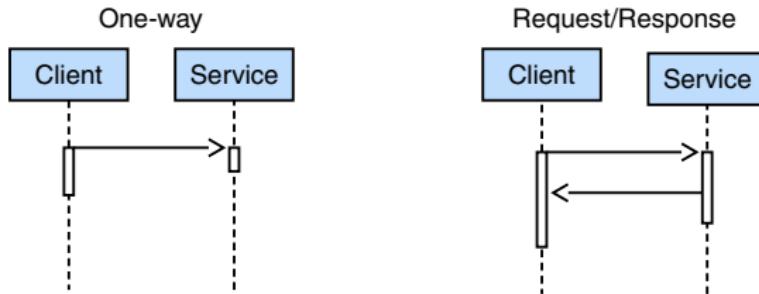
<binding>: How will the messages be transmitted on the wire?
What SOAP-specific details are there?

<service>: Define the collection of ports that make up the service and where
is the service located?

Defining a Web Service in WSDL



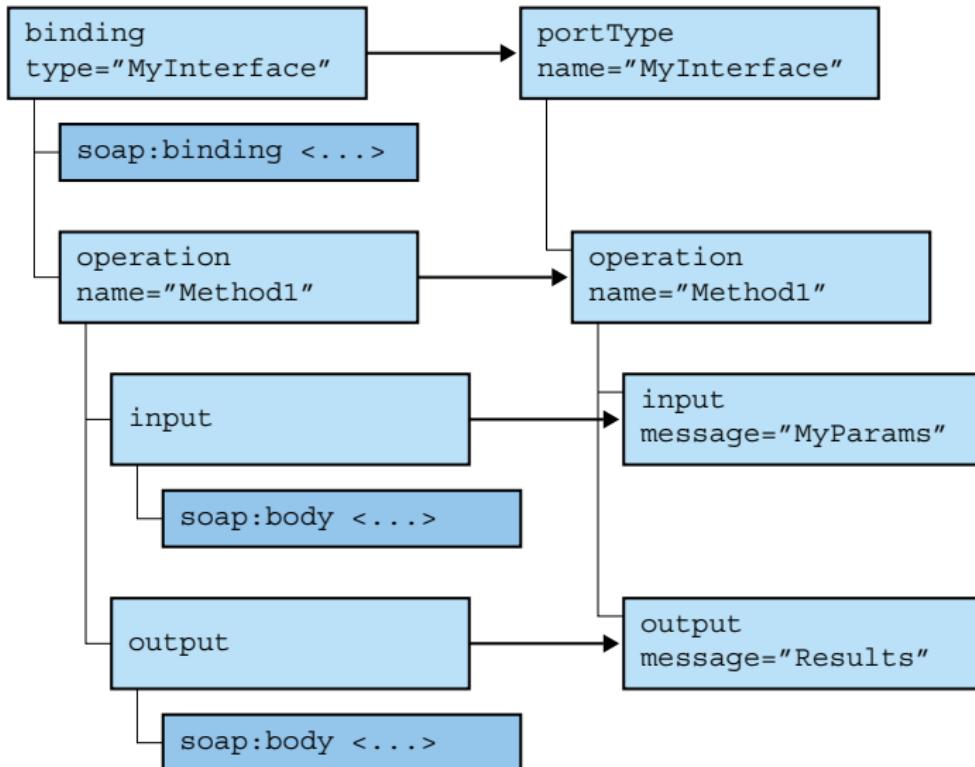
WSDL Interaction Scenarios



Logical vs Implementation Descriptions

- <portType>, <operation> and <message> represent the *logical* description of a web service: what the service can do.
 - WSDL files also provide some implementation guidance:
 - > XML schemas define the representation of the data embedded in messages
 - > WSDL bindings provide additional guidance:
 - Style of WSDL to use

WSDL Bindings



Describing SOAP Messages

```
1 <S:Envelope  
2     xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"  
3     xmlns:ser="http://server.jaxws.example.com/">  
4     <S:Header/>  
5     <S:Body>  
6         <ser:addAirport>  
7             <code>JFK</code>  
8             <name>New York John F. Kennedy Airport</name>  
9         </ser:addAirport>  
10    </S:Body>  
11 </S:Envelope>
```

- How many ways to construct this request?
 - How many ways to describe a given structure?

Variations of WSDL

Designers have to make two choices, when writing WSDL descriptions for web services:

- What “style” of WSDL definition to use
 - > RPC-style
 - > Document-style
- What data representation to use
 - > literal
 - > SOAP/RPC encoded (discouraged)

RPC Style

- Every input message results in a SOAP body with a single element that represents the function to be called, and which contains its parts
- Messages can have multiple parts:
 - > Every part of an input message is a parameter to the function call
 - > every part of an output message is a return value from the function call
- Every part is described by a builtin XML schema type, or by a complex type defined by the WSDL file

RPC Style

<portType>

```
1 <portType name="RPCLiteralAirportManager">
2   <operation name="addAirport"
3     parameterOrder="code_name">
4     <input message="tns:addAirport"/>
5     <output message="tns:addAirportResponse"/>
6   </operation>
7   <operation name="findNeighbors">
8     <input message="tns:findNeighbors"/>
9     <output message="tns:findNeighborsResponse"/>
10    </operation>
11 </portType>
```

rpcLiteralManager.wsdl

RPC Style

<message>

```
1 <message name="addAirport">
2   <part name="code" type="xsd:string"/>
3   <part name="name" type="xsd:string"/>
4 </message>
5 <message name="addAirportResponse">
6   <part name="return" type="xsd:long"/>
7 </message>
```

rpcLiteralManager.wsdl

RPC Style

<binding>

```
1 <binding name="AirportMgrBinding"
2     type="tns:RPCLiteralAirportManager">
3     <soap:binding style="rpc"
4         transport="http://schemas.xmlsoap.org/soap/http"/>
5     <operation name="addAirport">
6         <soap:operation soapAction=""></soap:operation>
7         <input>
8             <soap:body use="literal"
9                 namespace="http://server.jaxws.example.com/">
10            </input>
```

rpcLiteralManager.wsdl

RPC Style

Sample Message

```
1 <S:Envelope  
2     xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"  
3     xmlns:ser="http://server.jaxws.example.com/">  
4     <S:Header/>  
5     <S:Body>  
6         <ser:addAirport>  
7             <code>JFK</code>  
8             <name>New York John F. Kennedy Airport</name>  
9         </ser:addAirport>  
10    </S:Body>  
11 </S:Envelope>
```

RPC Style

<message>

```
1 <message name="findNeighbors">
2   <part name="code" type="xsd:string"/>
3 </message>
4 <message name="findNeighborsResponse">
5   <part xmlns:ns1="http://jaxb.dev.java.net/array"
6       name="return" type="ns1:stringArray"/>
7 </message>
```

rpcLiteralManager.wsdl

RPC Style

<schema>

```
1 <xs:complexType name="stringArray" final="#all">
2   <xs:sequence>
3     <xs:element name="item"
4       minOccurs="0" maxOccurs="unbounded"
5       nillable="true" type="xs:string" />
6   </xs:sequence>
7 </xs:complexType>
```

rpcLiteralManager.xsd

RPC Style

Sample Response

```
1 <S:Envelope  
2     xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">  
3     <S:Body>  
4         <ns2:findNeighborsResponse  
5             xmlns:ns2="http://server.jaxws.example.com/">  
6             <return>  
7                 <item>JFK</item>  
8                 <item>SFO</item>  
9             </return>  
10            </ns2:findNeighborsResponse>  
11        </S:Body>  
12    </S:Envelope>
```

RPC/literal Style

Java-to-WSDL Specification

```
1 @WebService(portName="AirportMgr",
2             serviceName="ManagersRPCLit")
3 @SOAPBinding(style=Style.RPC,
4             parameterStyle=ParameterStyle.WRAPPED,
5             use=Use.LITERAL)
6 public class RPCLiteralAirportManager {
7     @WebMethod
8     public long
9         addAirport (@WebParam(name="code") String code,
10                  @WebParam(name="name") String name) {
```

RPC/literal Style

Pros and Cons

- Advantages:
 - > Simple, readable WSDL
 - > SOAP message encodes operation and its parameters
 - > no redundant type information
 - > WS-I compliant
- Disadvantages:
 - > SOAP messages cannot be validated against schemas

Document Style

- Every message should contain a single part
- Every part is defined by an element in the XML schema defined within the WSDL file:
 - Since there is a single element for a single part for each input message, function calls end up modelled as requiring a single (object) parameter.
- No predefined or expected structure to the element associated with each message.

Document Style

<portType>

```
1 <portType name="DocumentLiteralAirportManager">
2   <operation name="addAirport">
3     <input message="tns:addAirport"/></input>
4     <output message="tns:addAirportResponse"/>
5   </operation>
6   <operation name="findNeighbors">
7     <input message="tns:findNeighbors/">
8     <output message="tns:findNeighborsResponse"/>
9   </operation>
10 </portType>
```

rpcLiteralManager.wsdl

Document Style

<message>

```
1 <message name="addAirport">
2   <part name="airport" element="tns:airport"/>
3 </message>
4 <message name="addAirportResponse">
5   <part name="addAirportResponse"
6       element="tns:addAirportResponse"/>
7 </message>
8 <message name="findNeighbors">
9   <part name="code" element="tns:code"/>
10 </message>
11 <message name="findNeighborsResponse">
12   <part name="findNeighborsResponse"
13       element="tns:findNeighborsResponse"/>
14 </message>
```

Document Style

<schema>

```
1 <xs:element name="addAirportResponse" type="xs:long"/>
2 <xs:element name="airport" nillable="true"
3     type="tns:airport"/>
4 <xs:element name="code" nillable="true" type="xs:string"/>
5 <xs:element name="findNeighborsResponse" nillable="true"
6     type="ns1:stringArray"/>
7 <xs:complexType name="airport">
8     <xs:complexContent>
9         <xs:sequence>
10            <xs:element name="code" type="xs:string" minOccurs="0"/>
11            <xs:element name="name" type="xs:string" minOccurs="0"/>
12        </xs:sequence>
13    </xs:complexContent>
14 </xs:complexType>
```

Document Style

<binding>

```
1 <binding name="AirportMgrBinding"
2     type="tns:DocumentLiteralAirportManager">
3     <soap:binding style="document"
4         transport="http://schemas.xmlsoap.org/soap/http"/>
5     <operation name="addAirport">
6         <soap:operation soapAction="" />
7         <input><soap:body use="literal" /></input>
8         <output><soap:body use="literal" /></output>
9     </operation>
10    <operation name="findNeighbors">
11        <soap:operation soapAction="" />
12        <input><soap:body use="literal" /></input>
13        <output><soap:body use="literal" /></output>
14    </operation>
15 </binding>
```

Document Style

Sample Message

```
1 <S:Envelope xmlns:ser="http://server.jaxws.example.com/"  
2     xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">  
3     <S:Header/>  
4     <S:Body>  
5         <ser:airport>  
6             <code>LGA</code>  
7             <name>New York La Guardia Airport</name>  
8         </ser:airport>  
9     </S:Body>  
10    </S:Envelope>
```

- What is the request?

Document/literal Style

Java-to-WSDL Specification

```
1 @WebService(portName="DocLiteralAirportMgr",
2             serviceName="ManagersDoc")
3 @SOAPBinding(style=Style.DOCUMENT,
4             parameterStyle=ParameterStyle.BARE,
5             use=Use.LITERAL)
6 public class DocumentLiteralAirportManager {
7     @WebMethod
8     public long
9         addAirport(@WebParam(name="airport")
10                 Airport airport) {
```

Document/literal Style

Pros and Cons

- Advantages:
 - > Validate SOAP messages against XML schema
 - > No redundant type information
 - > Mostly WS-I compliant
- Disadvantages:
 - > Somewhat complex WSDL
 - > SOAP message does not encode operation, which makes dispatching harder
 - > Multiple children to SOAP body violates WS-I

Document “Wrapped” Style

- Every message must contain a single part
- Each part is defined by an element in the XML schema defined within the WSDL file:
 - the root element associated with the input message is named to match the function to be called.
 - parameters to the function call are mapped to children of the root element associated with the input message.
 - return values from the function call are mapped to children of the root element associated with the output message
- This style explicitly builds a representation similar to that used by the RPC/literal style.

Using the Document “Wrapped” Style

- <portType> and <message> definitions look just like those for the Document/literal style
- Schema structure is richer:
 - > root element for message corresponds to function call
 - > parameters to call are explicitly listed as children of the root function call node.

Document “Wrapped” Style

<message>

```
1 <message name="addAirport">
2   <part name="parameters"
3     element="tns:addAirport"/>
4 </message>
5 <message name="addAirportResponse">
6   <part name="parameters"
7     element="tns:addAirportResponse"/>
8 </message>
```

Document “Wrapped” Style

<schema>

```
1 <xs:element name="addAirport" type="tns:addAirport"/>
2 <xs:element name="addAirportResponse"
3   type="tns:addAirportResponse"/>
4 <xs:complexType name="addAirport">
5   <xs:sequence>
6     <xs:element name="code" type="xs:string" minOccurs="0"/>
7     <xs:element name="name" type="xs:string" minOccurs="0"/>
8   </xs:sequence>
9 </xs:complexType>
10 <xs:complexType name="addAirportResponse">
11   <xs:sequence>
12     <xs:element name="return" type="xs:long"/>
13   </xs:sequence>
14 </xs:complexType>
```

Document “Wrapped” Style

<binding>

```
1 <binding name="WrappedAirportManagerPortBinding"
2   type="tns:AirportManager">
3     <soap:binding style="document"
4       transport="http://schemas.xmlsoap.org/soap/http"/>
5     <jaxws:bindings
6       xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
7       <jaxws:enableWrapperStyle>true</jaxws:enableWrapperStyle>
8     </jaxws:bindings>
9     <operation name="addAirport">
10      <soap:operation soapAction="" />
11      <input><soap:body use="literal" /></input>
12      <output><soap:body use="literal" /></output>
13    </operation>
```

Document “Wrapped” Style

Sample Message

```
1 <S:Envelope xmlns:ser="http://server.jaxws.example.com/"  
2   xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">  
3     <S:Header/>  
4     <S:Body>  
5       <ser:addAirport>  
6         <code>JFK</code>  
7         <name>New York Kennedy Airport</name>  
8       </ser:addAirport>  
9     </S:Body>  
10    </S:Envelope>
```

Document/literal “Wrapped” Style

Java-to-WSDL Specification

```
1 @WebService
2 public class AirportManager {
3     public long
4         addAirport(String code, String name) {
5             return dao.add(null, code, name).getId();
6         }
7     private AirportDAO dao = new AirportDAO();
8     // ...
```

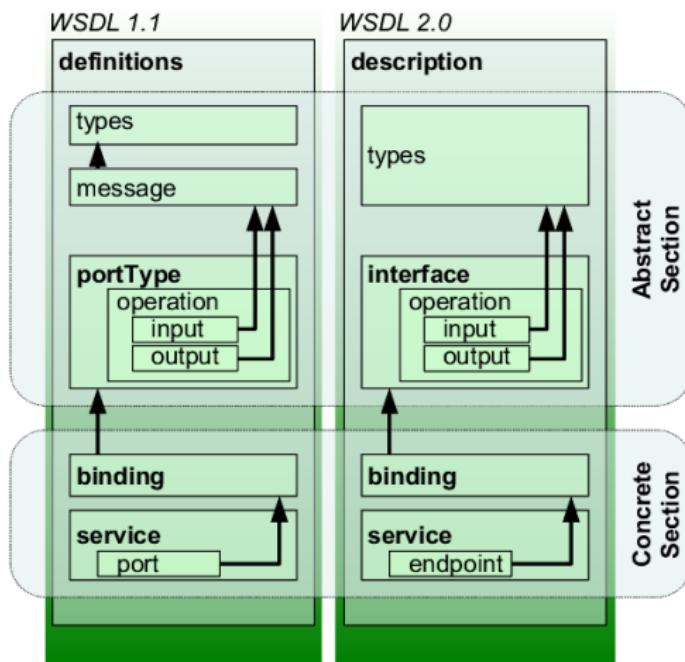
It is the default style!

Document/literal “Wrapped” Style

Pros and Cons

- Advantages:
 - > SOAP message encodes operation and its parameters
 - > Validate SOAP messages against XML schema
 - > No redundant type information
 - > WS-I compliant
 - Disadvantages:
 - > Very complex WSDL

WSDL 1.0 vs WSDL 2.0



Changes in WSDL 2.0

- Adding further semantics to the description language
- Removal of message constructs
- No support for operator overloading
- portType renamed to **interface**
- Port renamed to endpoint



Developing Web Services with Java™ Technology

Module 4

JAX-WS and JavaEE

Objectives

On completion of this module, you should:

- Understand how to deploy POJO web services to a web container
 - Understand how to define a web service in terms of an Enterprise Java Bean
 - Understand how to deploy an EJB web service to an EJB container
 - Describe the benefits associated with implementing a web service as an EJB

A Simple POJO Web Service

com.example.jaxws.server.AirportManager

```
1 @WebService
2 public class AirportManager {
3     public long
4         addAirport(String code, String name) {
5             return dao.add(null, code, name).getId();
6         }
7     private AirportDAO dao = new AirportDAO();
8 }
```

Deploying a Web Service on JavaSE

```
1 public class AirportManager {  
2     // ...  
3     static public void main(String[] args) {  
4         String url =  
5             "http://localhost:8080/airportManager";  
6         if (args.length > 0)  
7             url = args[1];  
8         AirportManager manager = new AirportManager();  
9         Endpoint endpoint =  
10            Endpoint.publish(url, manager);  
11    }  
12 }
```

Limitations of JavaSE Deployment

- HTTP Server built into JavaSE not designed for production use:
 - > Limited support for scalability
 - Thread pooling.
 - HTTP sessions in memory only.
 - > Limited infrastructure for security
 - Support for encryption HTTPS.
 - No builtin support for authentication and authorization.
 - Is there an alternative?

Deploying to a Web Container

- JAX-WS Web services can leverage a web container's infrastructure to process HTTP messages:
 - > The web container hands incoming HTTP requests to a servlet that feeds the JAX-WS runtime within the web container.
 - > As a convenience to the developer, JAX-WS in Glassfish will allow the developer to ignore the actual servlet responsible for initial processing of incoming HTTP requests.
 - This servlet is provided by JAX-WS.

Deploying to a Web Container

```
1 @WebService(serviceName="AirportManagerWS")
2 public class AirportManager {
3     public long
4         addAirport(String code, String name) {
5             return dao.add(null, code, name).getId();
6         }
7     private AirportDAO dao = new AirportDAO();
8 }
```

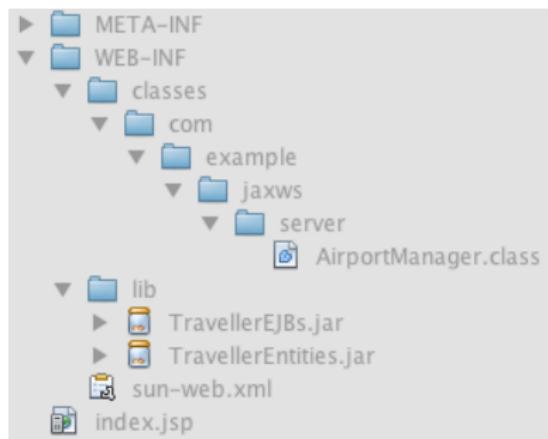
- No web.xml !
 - The web service will be available at the URL:

<http://context/AirportManagerWS>

Deploying to a Web Container

Packaging and Deployment

- Package into the WAR file the web service implementation class and all its supporting structure as if it were a servlet.



- Deploy WAR file to web container as usual

Deploying to a Web Container

- Advantages:
 - > Security infrastructure built in
 - > HTTP session management built in
 - > Support for scalability and availability built in
- Disadvantages:
 - > Transaction management still up to the application
 - > More complex deployment
 - > Requires a web container

Incorporating Security Constraints

- Best practices for security address these security principles:
 - > Maintaining Corporate Security Policies
 - > Self-Preservation
 - > Defense in Depth
 - > Least Privilege
 - > Compartmentalization
 - > Proportionality
 - Least Privilege requires authentication and authorization constraints on services.

Specifying Security Constraints

Declarative Authentication and Authorization

JavaEE annotations:

@DeclareRoles	Specify the set of roles relevant for a particular component
@RolesAllowed	Specify the set of roles required in order to call a method
@PermitAll	Allow anyone to call a method
@DenyAll	Allow no one to call a method

Specifying Security Constraints

A Secured Web Service

```
1 @WebService(serviceName="SecureManagerWS")
2 @DeclareRoles({"client", "administrator"})
3 public class SecureAirportManager {
4     @WebMethod @RolesAllowed("administrator")
5     public long addAirport(String code, String name) {
6         @WebMethod @PermitAll
7         public String getNameByCode(String code) {
8             return dao.findByCode(null, code).getName();
9         }
10        // ...
11        private AirportDAO dao = new AirportDAO();
12        @Resource WebServiceContext context;
13    }
```

Configuring the Web Application

Unfortunately, web containers need a little help to get our SecureAirportManager to work as intended:

- The `web.xml` file has to specify:
 - > that there are protected URLs to be considered
 - > which authentication protocol to use
 - > which roles to expect
- The server-specific configuration has to specify:
 - > the mapping from roles expected to principals that can assume that role
 - > the set of principals known.

Configuring web.xml

Securing Web Service URLs

```
1 <security-constraint>
2   <display-name>SecureManagerWS</display-name>
3   <web-resource-collection>
4     <web-resource-name>
5       SecureManagerWS
6     </web-resource-name>
7     <url-pattern>/SecureManagerWS</url-pattern>
8     <http-method>POST</http-method>
9   </web-resource-collection>
10  <auth-constraint>
11    <role-name>administrator</role-name>
12    <role-name>client</role-name>
13  </auth-constraint>
14 </security-constraint>
```

Configuring web.xml

Configuring Login

```
15 <login-config>
16     <auth-method>BASIC</auth-method>
17     <realm-name>file</realm-name>
18 </login-config>
19 <security-role>
20     <role-name>client</role-name>
21 </security-role>
22 <security-role>
23     <role-name>administrator</role-name>
24 </security-role>
```

Configuring sun-web.xml

```
1 <sun-web-app error-url="">
2   <context-root>/JavaEEJAXWS</context-root>
3   <security-role-mapping>
4     <role-name>client</role-name>
5     <principal-name>tracy</principal-name>
6   </security-role-mapping>
7   <security-role-mapping>
8     <role-name>administrator</role-name>
9     <principal-name>kelly</principal-name>
10  </security-role-mapping>
```

- We could have mapped a role-name to a *group*, instead of listing the individual principals here...

Retrieving Security Information

- Sometimes, the simple role-based access control strategy is not enough.
- Applications can add further authorization constraints *programmatically*:
 - it may be necessary to obtain the specific identity of the caller, and not just the role (or group) they belong to:
 - via API
 - via *injection*

Retrieving Security Information

A Servlet

```
1 @WebServlet(name="SecureServlet",
2             urlPatterns={"/SecureServlet"})
3 public class SecureServlet extends HttpServlet {
4     protected void
5         processRequest(HttpServletRequest request,
6                         HttpServletResponse response)
7     throws ServletException, IOException {
8         response.setContentType("text/html;charset=UTF-8");
9         PrintWriter out = response.getWriter();
10        String user = request.getRemoteUser();
11        Principal principal = request.getUserPrincipal();
```

Retrieving Security Information

- The API that allows a Servlet to retrieve security information is built into one of the parameters to its `service()` call
 - > they are always available
- The only arguments provided to a web service method are those required by its application-level service definition. How to match the API given to a Servlet?
 - > Context objects provide useful services to components
 - > How can a context object be provided to a web service?

Retrieving Security Information

Dependency Injection

```
1 @WebService(serviceName="SecureManagerWS")
2 public class SecureAirportManager {
3     @WebMethod @RolesAllowed("administrator")
4     public long addAirport(String code, String name) {
5         String user = context.getUserPrincipal().getName();
6         return dao.add(null, code, name).getId();
7     }
8     // ...
9     @Resource WebServiceContext context;
10 }
```

- WebServiceContext also offers access to the underlying MessageContext

Retrieving Security Information

Logging Callers

```
1 @WebService(serviceName="SecureManagerWS")
2 public class SecureAirportManager {
3     @WebMethod @RolesAllowed("administrator")
4     public long addAirport(String code, String name) {
5         String user = context.getUserPrincipal().getName();
6         MessageContext msgContext = context.getMessageContext();
7         ServletContext webContext = (ServletContext)
8             msgContext.get(MessageContext.SERVLET_CONTEXT);
9         webContext.log("add_requested_by:_ " + user);
10        return dao.add(null, code, name).getId();
11    }
12    // ...
13    @Resource WebServiceContext context;
14 }
```

Accessing the Web Infrastructure

- WebServiceContext
 - > Offers `getUserPrincipal()` and `isUserInRole()`
 - > Offers access to the underlying MessageContext
- MessageContext
 - > Offers information from the underlying web infrastructure:

SERVLET_CONTEXT	HTTP_REQUEST_METHOD
SERVLET_REQUEST	HTTP_REQUEST_HEADERS
SERVLET_RESPONSE	HTTP_REQUEST_ATTACHMENTS
QUERY_STRING	PATH_INFO
WSDL_DESCRIPTION	WSDL_OPERATION

...

Authenticating POJO WS Client

```
1  public class AuthSimpleClient {  
2      public static void main(String[] args) {  
3          AirportManagerService service =  
4              new AirportManagerService();  
5          AirportManager port = service.getAirportManagerPort();  
6          Map<String, Object> reqCtx =  
7              ((BindingProvider) port).getRequestContext();  
8          reqCtx.put(BindingProvider.USERNAME_PROPERTY,  
9                  "tracy");  
10         reqCtx.put(BindingProvider.PASSWORD_PROPERTY,  
11                  "password");  
12         java.lang.String code = "LGA";  
13         java.lang.String name = "New_York_LaGuardia";  
14         long result = port.addAirport(code, name);  
15         System.out.println("Result_=_" + result);
```

A POJO Web Service Using a DAO

```
1 @WebService
2 public class AirportManager {
3     public long
4     addAirport(String code, String name) {
5         return dao.add(null, code, name).getId();
6     }
7     private AirportDAO dao = new AirportDAO();
8 }
```

- Ideally, concerns related to transaction management and persistence contexts should be hidden within the DAO...
- ... but is it always feasible?

A POJO Data Access Object

```
1  class GenericDAO<PersistentClass extends DomainEntity> {
2      public PersistentClass add(EntityManager em,
3                                  PersistentClass newObj) {
4          EntityTransaction tx = null;
5          if (em == null) {
6              em = getEMF().createEntityManager();
7              tx = em.getTransaction();
8          }
9          if (tx != null) tx.begin();
10         em.persist( newObj );
11         if (tx != null) {
12             tx.commit();
13             em.close();
14         }
15         return newObj;
16     }
```

A POJO Data Access Object

```
1  public class AirportDAO extends GenericDAO<Airport> {  
2      public  
3          Airport add(EntityManager em, String code, String name) {  
4              return add( em, new Airport( code, name ) );  
5      }  
6      public void removeByCode(EntityManager em, String code) {  
7          EntityTransaction tx = null;  
8          if (em == null) {  
9              em = getEMF().createEntityManager();  
10             tx = em.getTransaction();  
11         }  
12         if (tx != null) tx.begin();  
13         remove( em, findByCode(em, code));
```

- removeByCode() is one transaction...

A More Complex POJO Web Service

```
1  @WebService
2  public class BetterAirportManager {
3      @WebMethod(operationName="removeByCode")
4      public void removeAirport(String code) {
5          EntityManager em =
6              GenericDAO.getEMF().createEntityManager();
7          EntityTransaction tx = em.getTransaction();
8          tx.begin();
9          dao.remove(em, dao.findByName(em, code));
10         tx.commit();
11         em.close();
12     }
```

- This had to be one transaction...

Limitations of a POJO Web Service

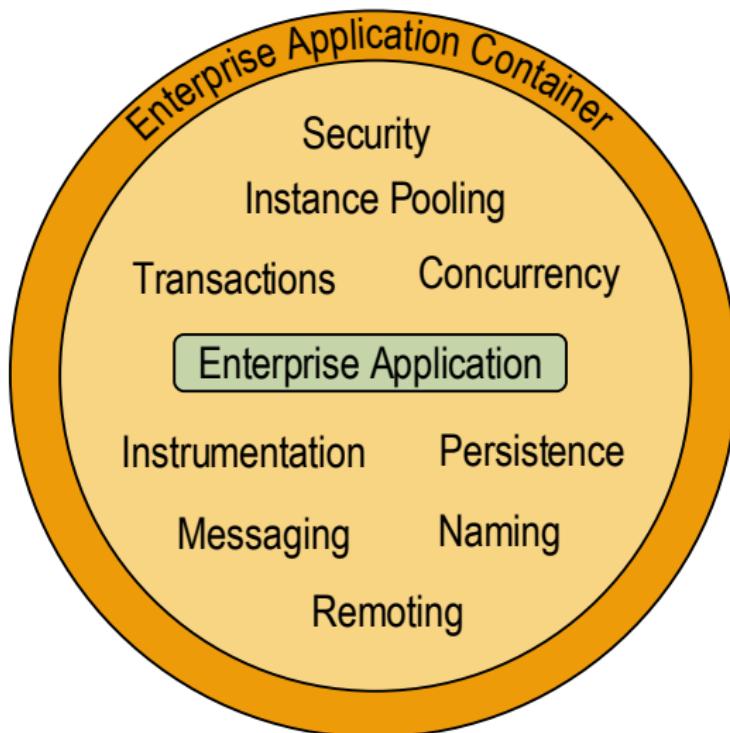
- A POJO Web Service endpoint is “just a POJO” – and so shares some limitations of all POJOs:
 - > explicit transaction management
 - > explicit persistence context management
- By default, POJO web services are stateless, and a new instance is allocated for each call:
 - > this is an advantage from the point of view of concurrency, as each instance is independent
 - > this is a disadvantage from the point of view of scalability, since the number of instances could explode, as well as the garbage-collection overhead.

Enterprise Java Beans

Goals of the Enterprise Java Beans specification include:

- The Enterprise JavaBeans architecture will support the development, deployment, and use of web services.
- The Enterprise JavaBeans architecture will make it easy to write applications: application developers will not have to understand low-level transaction and state management details, multi-threading, connection pooling, or other complex low-level APIs.

EJB Container Services



Advantages of EJBs

EJBs are “just” *smart* POJOs:

- Approaches to improve scalability can be declarative:
 - > Service instance lifecycle choice: shared stateless pool, shared singleton, per-client.
 - > Concurrency management built-in: stateless and stateful (per-client) instances guarantee safe concurrent execution, while singletons support declarative concurrency control.
 - Transaction management can be declarative
 - > Persistence contexts can be managed transparently (one context per transaction), or programmatically.

An EJB Data Access Object

Generic Base Class

```
1 @TransactionAttribute(TransactionAttributeType.REQUIRED)
2 public abstract
3 class GenericDAO<PersistentClass extends DomainEntity> {
4     @PersistenceContext(unitName="TravellerEJBsPU")
5     protected EntityManager em;
6     public PersistentClass add(PersistentClass newInstance) {
7         em.persist(newInstance);
8         return newInstance;
9     }
10    public PersistentClass update(PersistentClass instance) {
11        PersistentClass result = em.merge(instance);
12        return result;
13    }
```

An EJB Data Access Object

```
1 @Stateless
2 public class AirportDAO extends GenericDAO<Airport> {
3     public Airport add( String code, String name ) {
4         return add( new Airport( code, name ) );
5     }
6     public Airport findByCode( String code ) {
7         String sqlQuery =
8             "SELECT _a_ from _Airport_a_ WHERE _a_.code=_:code";
9         TypedQuery<Airport> query =
10            em.createQuery(sqlQuery,Airport.class);
11         query.setParameter( "code", code );
12         List<Airport> results = query.getResultList();
13         if (results.size() != 1)
14             throw new QueryException();
15         return results.get(0);
16     }
```

Creating Web Services from EJBs

Support for deploying EJBs as Web Services is built into Java EE:

```
1 @WebService(serviceName="AirportManagerEJBWS")
2 @Stateless
3 public class AirportManager {
4     public long
5         addAirport(String code, String name) {
6             return dao.add(code, name).getId();
7         }
8         @EJB private AirportDAO dao;
9 }
```

Injecting EJBs

```
1 @WebService(serviceName="AirportManagerEJBWS")
2 @Stateless
3 public class AirportManager {
4     public long
5         addAirport(String code, String name) {
6             return dao.add(code, name).getId();
7         }
8     @EJB private AirportDAO dao;
9 }
```

- AirportManager obtains its DAO via *injection*
 - > whether/how that AirportDAO is shared is transparent!

A More Complex EJB Web Service

```
1 @WebService(serviceName="BetterManagerEJBWS")
2 @Stateless
3 public class BetterAirportManager {
4     @WebMethod
5     public long
6         addAirport(String code, String name) {
7             return dao.add(code, name).getId();
8         }
9     @WebMethod(operationName="removeByCode")
10    public void removeAirport(String code) {
11        dao.remove(dao.findByCode(code));
12    }
13    @EJB private AirportDAO dao;
14 }
```

Types of EJBs

There are three types of EJBs that can also be web service endpoints:

- Stateless
- Singleton
- Stateful – but not interoperable

There is fourth type of EJB – Message-Driven Beans.

Stateless Session Beans

Stateless session beans are pools of shared objects:

- any one instance can only be used by one client at a time
- instances return to the pool after each use

Pros and Con: Stateless EJBs

- Advantages:
 - > Improved scalability through shared pool of instances
 - > Transparent transaction management
- Disadvantages:
 - > Increased network overhead when engaging in conversations

Singleton EJBs

Singleton beans are single instances per JVM:

- all clients share the same instance
- a concurrency control policy must be specified

Singleton Web Service EJBs

```
1 @WebService(serviceName="SingletonManagerEJBWS")
2 @Singleton
3 public class SingletonAirportManager {
4     @WebMethod
5     public long addAirport(String code, String name) {
6         return dao.add(code, name).getId();
7     }
8     @WebMethod(operationName="removeByCode")
9     public void removeAirport(String code) {
10        dao.remove(dao.findByCode(code));
11    }
12    @WebMethod
13    public String getNameByCode(String code) {
14        return dao.findByCode(code).getName();
15    }
```

Concurrency Management

Two strategies are available:

- Container-managed concurrency
 - > Container will use read/write locks to control concurrent access to the singleton
 - > Developer describes read/write behavior of each method declaratively
- Bean-managed concurrency
 - > Developer uses Java mechanisms (such as **synchronized** blocks) to control concurrent access to the singleton.

Singleton Web Service EJBs

```
1  @WebService(serviceName="SingletonManagerEJBWS")
2  @Singleton
3  @Lock(LockType.WRITE)
4  public class SingletonAirportManager {
5      @WebMethod
6      public long addAirport(String code, String name) {
7          return dao.add(code, name).getId();
8      }
9      @WebMethod(operationName="removeByCode")
10     public void removeAirport(String code) {
11         dao.remove(dao.findByCode(code));
12     }
13     @WebMethod
14     @Lock(LockType.READ)
15     public String getNameByCode(String code) {
16         return dao.findByCode(code).getName();
17     }
```

Pros and Cons: Singletons

- Advantages:
 - > Convenient caching of shared state
 - > Transparent transaction management
- Disadvantages:
 - > Increased network overhead when engaging in conversations
 - > Possible bottleneck due to concurrency control policy



Developing Web Services with Java™ Technology

Module 5

Implementing More Complex Services Using JAX-WS

Objectives

On completion of this module, you should:

- Apply JAXB to pass complex objects to and from a web service
 - Understand how to map Java exceptions from a web service provider to SOAP faults
 - Inject attributes into JAX-WS web service endpoints
 - Describe JAX-WS artifacts that can be injected and how to use them

Complex Arguments

- JAX-WS supports the use of complex objects as arguments or return types.
 - > Pretty much any serializable class can be used
 - JAX-WS relies on JAXB to provide support for such interactions.
 - > Java classes to be used so need to be annotated with the proper JAXB annotation(s).
 - RMI Remote objects may not be used as parameters or return types.

Complex Arguments

Service Class with Complex Return Type

```
1 @WebService  
2 public class FlightManager {  
3     @WebMethod(operationName="simpleAddFlight")  
4     public Flight  
5         addFlight(String airline, String number,  
6                     String departsPort, String arrivesPort) {  
7             Date now = new Date();  
8             Flight flight = new  
9                 Flight(airline, number,  
10                     airportDAO.findByName(departsPort),  
11                     airportDAO.findByName(arrivesPort),  
12                     now, now, 150);  
13             return flightDao.add(flight);  
14 }
```

Complex Arguments

Class with JAXB Annotations

```
1 @XmlRootElement
2 public class Airport
3     implements Serializable {
4     private String code;
5     private String name;
6     public Airport() {}
7     public Airport( String code, String name ) {
8         this.code = code; this.name = name;
9     }
```

Complex JAXB Mappings

Elements and Attributes

```
1 public class DomainEntity {  
2     @XmlAttribute  
3     protected long id;
```

```
1 @XmlRootElement  
2 public class Airport  
3     extends DomainEntity  
4     implements Serializable {  
5     private String code;  
6     private String name;
```

XML Representation

Using Elements and Attributes

```
<airport id="5">
  <code>MCO</code>
  <name>Orlando International Airport</name>
</airport>
```

- both inherited and immediate properties are included in the marshall/unmarshall logic used by JAXB.

Enums in XML

```
1 @XmlType
2 public class Payment extends DomainEntity
3     implements Serializable {
4     @XmlEnum(String.class)
5     public static enum Status {
6         pending, processing, accepted, rejected
7     };
8     private String creditCardNum;
9     private Date expirationDate;
10    private Status status = Status.pending;
```

FlightManager Web Service

Generated XML Schema – Inheritance

```
1 <xs:complexType name="domainEntity">
2   <xs:sequence/>
3     <xs:attribute name="id" type="xs:long" use="required"/>
4     <xs:attribute name="version" type="xs:int"/>
5   </xs:complexType>
6   <xs:complexType name="payment">
7     <xs:complexContent>
8       <xs:extension base="tns:domainEntity">
9         <xs:sequence>
10           <xs:element name="ticket" type="tns:ticket"/>
11           <xs:element name="creditCardNum" type="xs:string"/>
12           <xs:element name="bankName" type="xs:string"/>
13           <xs:element name="expirationDate" type="xs:dateTime"/>
14           <xs:element name="status" type="tns:status"/>
15         </xs:sequence>
16       </xs:extension>
17     </xs:complexContent>
18   </xs:complexType>
```

FlightManager Web Service

Generated XML Schema – Enums

```
1 <xssimpleType name="status">
2   <xsrrestriction base="xs:string">
3     <xsenumeration value="pending"/>
4     <xsenumeration value="processing"/>
5     <xsenumeration value="accepted"/>
6     <xsenumeration value="rejected"/>
7   </xsrrestriction>
8 </xssimpleType>
```

Exceptions in Web Services

Exceptions result from error conditions or unexpected behavior in a running program. Exceptions in a web service can occur due to:

- Invalid input received by the web service from the client
- Unavailability of resources needed by the web service

Sometimes, the web service must generate an exception and deliver it to the client.

Exceptions in Top-Down Development

- The WSDL file describes at least one possible failure condition for a given operation, represented as a fault message.
- The fault message is described in terms of a message element, and XML schema types.
- `wsimport` generates both a Java Exception class and a Java Bean class to embed within the Exception, representing the data associated with the exception.
- Application-level logic throws and catches this Exception type as appropriate.

WSDL Code Fragment

fault Message

```
1 <message name="addPassengerFault">
2   <part name="parameters" element="tns:addPassengerFault"/>
3 </message>
4 <portType name="SaferPassengerManager">
5   <operation name="addPassenger">
6     <input name="input1" message="tns:addPassengerRequest"/>
7     <output name="out1" message="tns:addPassengerResponse"/>
8     <fault name="fault1" message="tns:addPassengerFault"/>
9   </operation>
10 </portType>
```

XML Schema Type Fragment

fault Message

```
1 <xsd:element name="addPassengerFault">
2   <xsd:complexType>
3     <xsd:sequence>
4       <xsd:element name="duplicateCode" type="xsd:string"/>
5       <xsd:element name="duplicateName" type="xsd:string"/>
6     </xsd:sequence>
7   </xsd:complexType>
8 </xsd:element>
```

Fault Type Generated by JAX-WS

```
1  @XmlType(name = "", propOrder = {  
2      "duplicateCode",  
3      "duplicateName"  
4  })  
5  @XmlRootElement(name = "addPassengerFault")  
6  public class AddPassengerFault {  
7      @XmlElement(required = true)  
8      protected String duplicateCode;  
9      @XmlElement(required = true)  
10     protected String duplicateName;
```

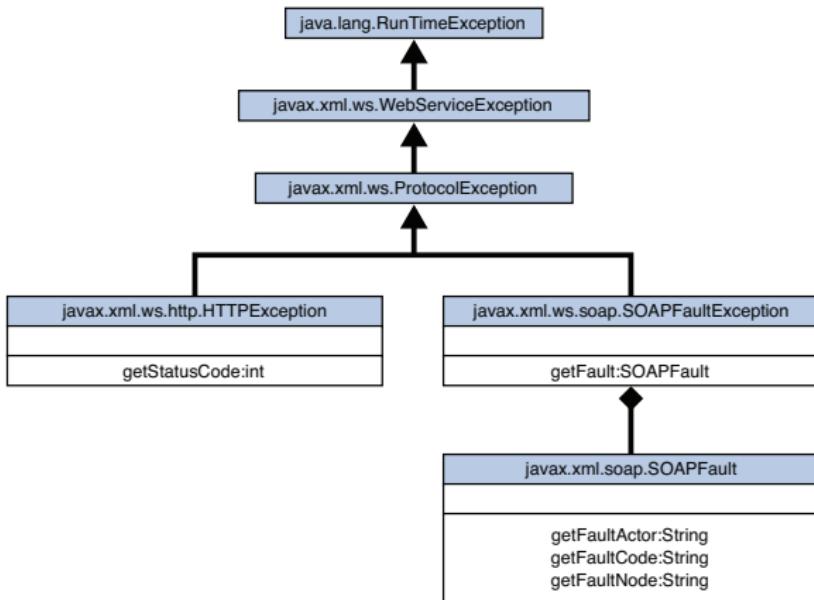
Exception Type Generated by JAX-WS

```
1 @WebFault(name = "addPassengerFault",
2             targetNamespace = "urn://Traveller/")
3 public class AddPassengerFault_Exception
4     extends Exception {
5     private AddPassengerFault faultInfo;
6     public AddPassengerFault_Exception(String message,
7             AddPassengerFault faultInfo) {
8         super(message);
9         this.faultInfo = faultInfo;
10    }
11    public AddPassengerFault_Exception(String message,
12             AddPassengerFault faultInfo, Throwable cause) {
13        super(message, cause);
14        this.faultInfo = faultInfo;
15    }
```

Exceptions in Bottom-Up Development

- The Java code defines at least one Exception class that is thrown and caught as appropriate, at application level.
- `apt` or `wsgen` generate a description for the operation in question that includes a fault message to represent the potential Exception, and its XML schema types.
- Java Exception class is mapped to a single XML schema type.

JAX-WS Exception Class Hierarchy



JAX-WS Resources

- JAX-WS offers web service endpoints access to the JAX-WS context around each request via a `WebServiceContext` instance:
 - Obtained via injection, by annotating `WebServiceContext` attribute of endpoint with standard `@Resource`
 - Offers `getUserPrincipal()` and `isUserInRole()`
 - Offers access to the underlying `MessageContext`
- `MessageContext` offers more detailed information about the current web service call.

Accessing the Web Infrastructure

MessageContext provides access to these servlet resources:

- SERVLET_CONTEXT
- SERVLET_REQUEST
- SERVLET_RESPONSE

Accessing the Web Infrastructure

MessageContext provides access to these HTTPResources:

- HTTP_REQUEST_METHOD
- HTTP_REQUEST_HEADERS
- HTTP_REQUEST_ATTACHMENTS
- HTTP_RESPONSE_CODE
- HTTP_RESPONSE_HEADERS
- QUERY_STRING
- PATH_INFO

Accessing the Web Infrastructure

MessageContext provides access to these JAX-WS resources:

- WSDL_DESCRIPTION
- WSDL_INTERFACE
- WSDL_OPERATION
- WSDL_PORT
- WSDL_SERVICE
- REFERENCE_PARAMETERS



Developing Web Services with Java™ Technology

Module 6

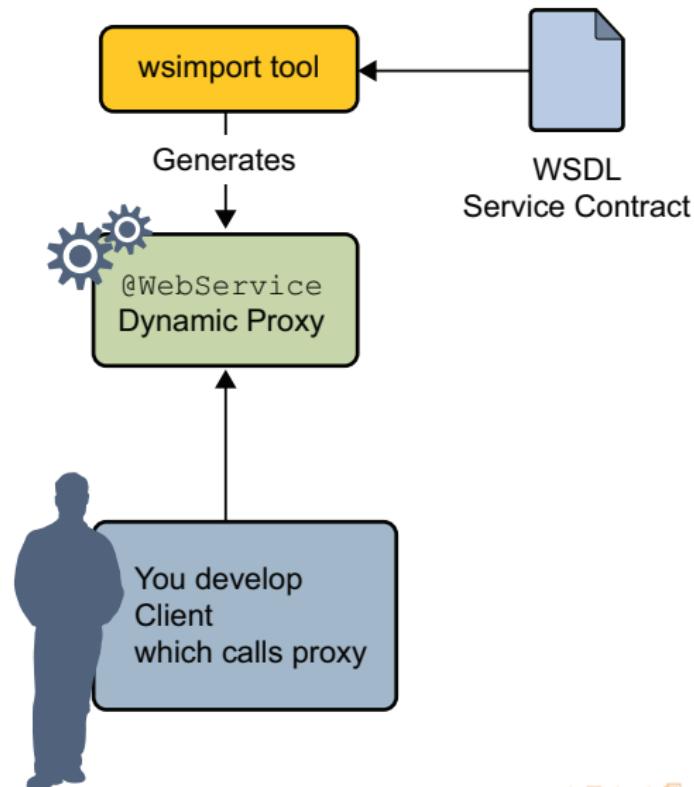
JAX-WS Web Service Clients

Objectives

On completion of this module, you should be able to:

- Understand how to create web service clients using JAX-WS
 - > Create simple JavaSE web service clients
 - > Create web service clients in a JavaEE container
 - Understand how to create web service clients using JAX-WS that support asynchronous interactions:
 - > One-way interactions
 - > Asynchronous pull-based interactions
 - > Asynchronous push-based interactions

Developing Web Service Clients



Sample wsimport Ant Task

```
1 <taskdef name="wsimport"
2     classname="com.sun.tools.ws.ant.WsImport">
3     <classpath path="${libs.JAX-WS-22_RI.classpath}"/>
4 </taskdef>
5 <target name="-pre-compile" depends="-post-init">
6     <wsimport
7         wsdl="${basedir}/${src.dir}/xml/service.wsdl"
8         destdir="build"
9         sourcedestdir="generated/src"
10        endorsed="true"
11        package="com.example.generated">
12     </wsimport>
```

Simple POJO WS Client

```
1 public class SimpleClient {
2     public static void main(String[] args) {
3         AirportManagerService service =
4             new AirportManagerService();
5         AirportManager port =
6             service.getAirportManagerPort();
7         java.lang.String code = "LGA";
8         java.lang.String name = "New_York_LaGuardia";
9         long result = port.addAirport(code, name);
10        System.out.println("Result = "+result);
11    }
12 }
```

Restricting the Schema

```
1 <xs:element name="addAirport" type="tns:addAirport"/>
2 <xs:complexType name="addAirport">
3     <xs:sequence>
4         <xs:element name="arg0" type="tns:code" minOccurs="0"/>
5         <xs:element name="arg1" type="xs:string" minOccurs="0"/>
6     </xs:sequence>
7 </xs:complexType>
8 <xs:simpleType name="code">
9     <xs:restriction base="xs:string">
10        <xs:pattern value="\w{3}"/>
11    </xs:restriction>
12 </xs:simpleType>
```

```
1 <message name="addAirport">
2   <part name="parameters" element="tns:addAirport"/>
3 </message>
```

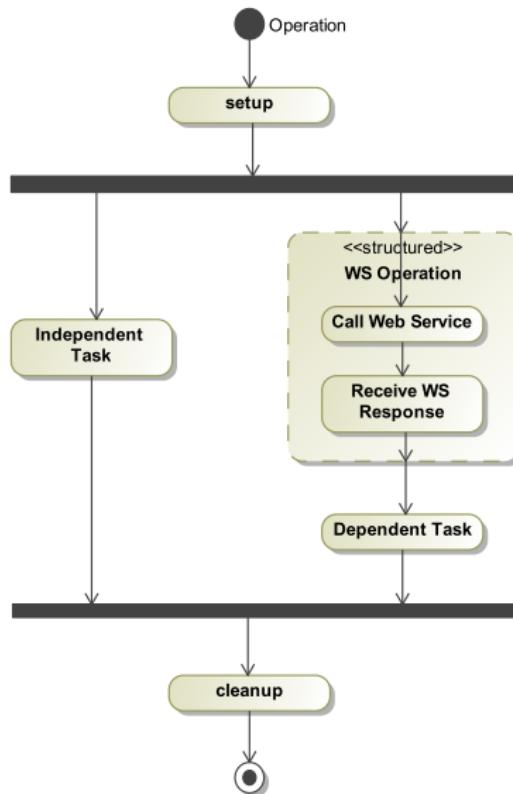
Validating POJO WS Client

```
1 public class StricterClient {  
2     public static void main(String[] args) {  
3         StricterAirportManagerService service =  
4             new StricterAirportManagerService();  
5         WebServiceFeature validation =  
6             new SchemaValidationFeature();  
7         StricterAirportManager port =  
8             service.getAirportManagerPort(validation);  
9         java.lang.String code =  
10            (args.length >= 1) ? args[0] : "NYLGA";  
11         java.lang.String name =  
12            (args.length >= 2) ? args[1] : "New_York_LaGuardia";  
13         long result = port.addAirport(code, name);  
14         System.out.println("Result_=_" + result);  
15     }  
16 }
```

JavaEE Web Service Clients

```
1  @WebService @Stateless
2  public class Planner {
3      @WebServiceRef(wsdlLocation =
4          "http://localhost:8081/airportManager.wsdl")
5      private SaferAirportManagerService service;
6      public List<String> getNeighbors(String code) {
7          SaferAirportManager airportWS =
8              service.getSaferAirportManagerPort();
9          List<Airport> candidates = airportWS.findNeighbors(code);
10         List<String> result = new ArrayList<String>();
11         for (Airport a : candidates)
12             result.add(a.getCode());
13         return result;
14     }
15 }
```

A Prototypical Operation



Optimizing the Operation

- Sequential execution would lead to needless delay while the client waits for the web service call to return.
 - Common strategies to optimize such an operation include:
 - > execute both branches in different threads
 - > delegate the expensive operation (and execution of DependentTask)
 - Someone else would wait...
 - Is there another way?

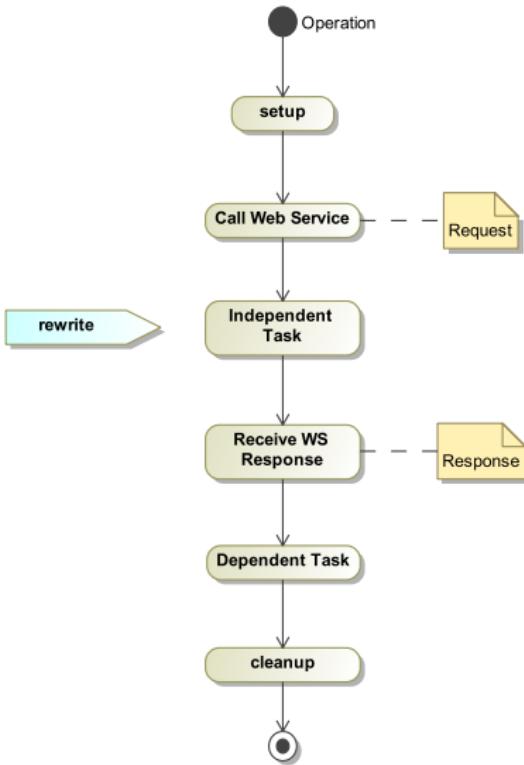
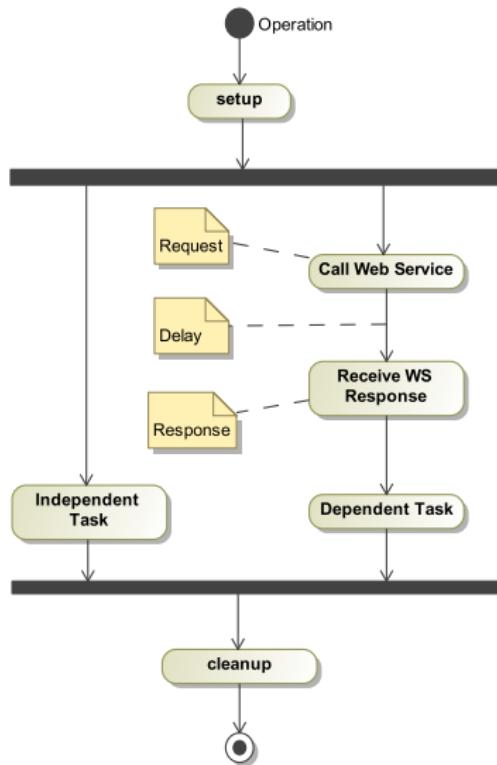
One-Way Operations

WSDL descriptions can capture a *one-way* message exchange pattern:

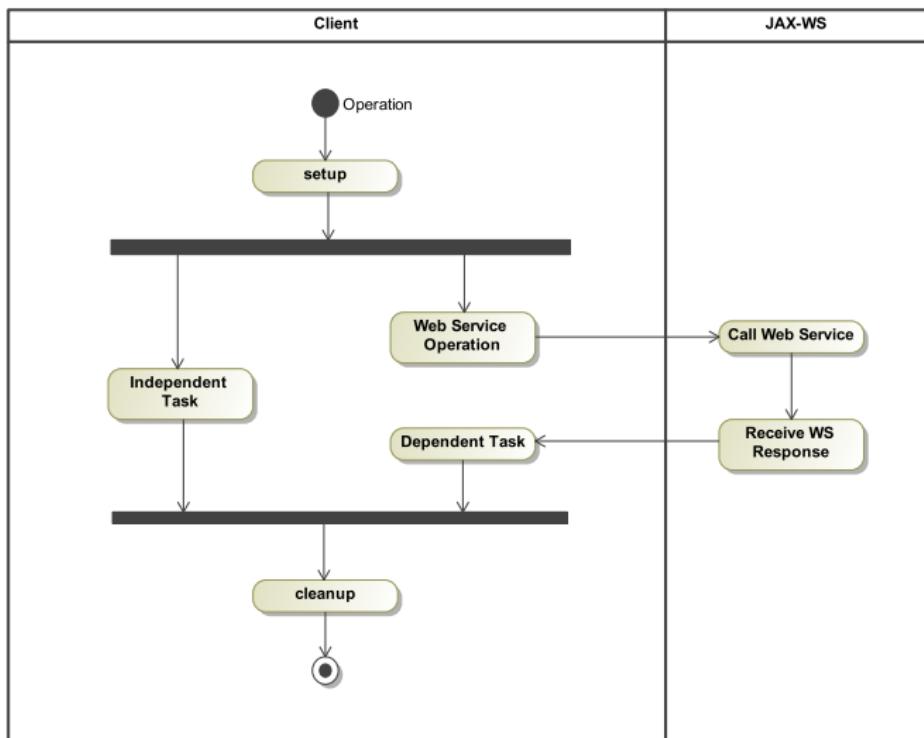
```
1 @WebService
2 public class OneWayPassengerManager {
3     @Oneway
4     public void removePassenger(long id) {
5         dao.remove(null, id);
6     }
7     private PassengerDAO dao = new PassengerDAO();
```

- But what about return values?

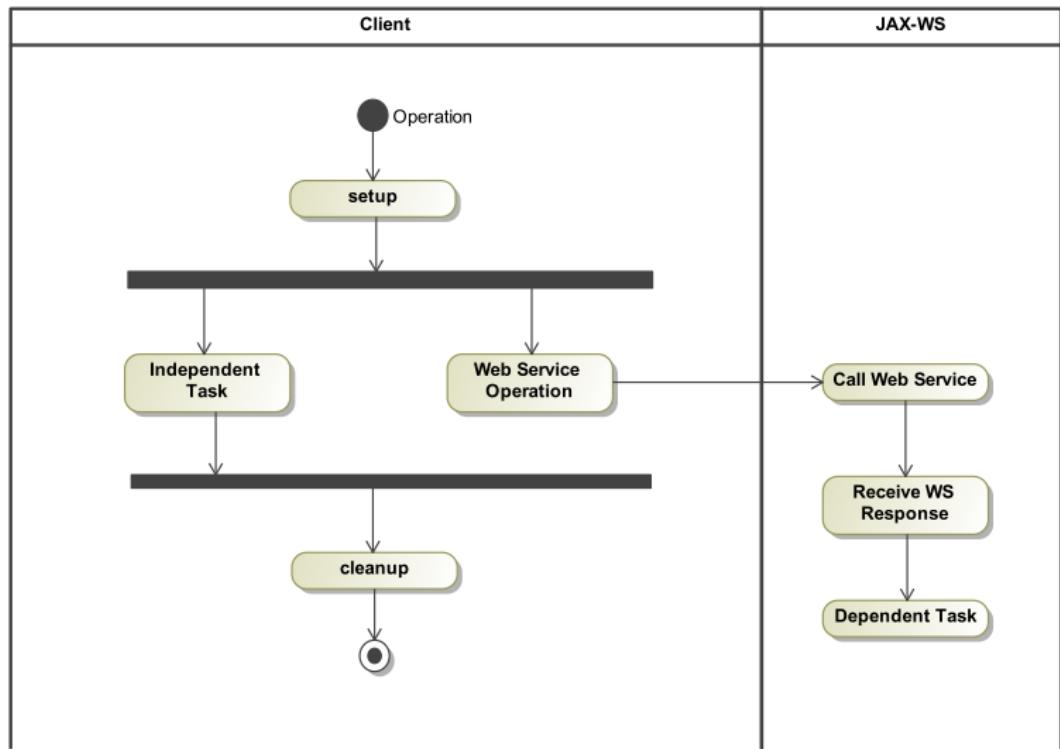
An Asynchronous Request



A Prototypical Operation - Revisited



Async Requests – Another Approach



Requesting Asynchronous Support

Support must be requested:

```
1 <portType name="AirportManager">
2   <jaxws:bindings
3     xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
4     <jaxws:enableAsyncMapping>
5       true
6     </jaxws:enableAsyncMapping>
7   </jaxws:bindings>
8   <operation name="addAirport">
```

- The actual interaction between client and server is not affected – only the client's control flow is.

Asynchronous Operations

```
1 @WebService(name = "AirportManager",
2     targetNamespace = "http://server.jaxws.example.com/")
3 public interface AsyncAirportManager {
4     @WebMethod(operationName = "addAirport")
5     public Response<AddAirportResponse>
6         addAirportAsync(String code, String name);
7     @WebMethod(operationName = "addAirport")
8     public Future<?>
9         addAirportAsync(String code, String name,
10                         @WebParam(name = "asyncHandler")
11                         AsyncHandler<AddAirportResponse> h);
12     @WebMethod
13     public long addAirport(String code, String name);
14 }
```

Client Using Response<>

```
1  public class AsyncClientViaResponse {
2      public static void main(String[] args)
3          throws Exception {
4              clients.async.AirportManagerService service =
5                  new clients.async.AirportManagerService();
6              AsyncAirportManager port =
7                  service.getAirportManagerPort();
8              java.lang.String code = "LGA";
9              java.lang.String name = "New_York_LaGuardia";
10             Response<clients.async.AddAirportResponse> holder =
11                 port.addAirportAsync(code, name);
12             // some other work goes here...
13             long result = holder.get().getReturn();
14             System.out.println("Result_=_" + result);
15         }
16     }
```

Client Using AsyncHandler

AsyncHandler<AddAirportResponse>

```
1 class AsyncClientHandler implements AsyncHandler {  
2     public void handleResponse( Response res ) {  
3         Response<clients.async.AddAirportResponse> response =  
4             (Response<clients.async.AddAirportResponse>) res;  
5         try { System.out.println(response.get().getReturn()); }  
6         catch( Exception ex ) {  
7             System.out.println( ex.getMessage() );  
8         }  
9     }  
10 }
```

Client Using AsyncHandler

```
11 public class AsyncClientViaHandler {  
12     public static void main(String[] args)  
13         throws Exception {  
14         clients.async.AirportManagerService service =  
15             new clients.async.AirportManagerService();  
16         AsyncAirportManager port =  
17             service.getAirportManagerPort();  
18         java.lang.String code = "BOS";  
19         java.lang.String name = "Boston_Logan";  
20         AsyncHandler handler = new AsyncClientHandler();  
21         Future<?> holder =  
22             port.addAirportAsync(code, name, handler);  
23         // now we can go do anything else we need...  
24         Thread.sleep(30 * 1000);  
25     }  
26 }
```



**Developing Web Services with Java™
Technology**

Module 7

Introduction to RESTful Web Services

Objectives

On completion of this module, you should:

- Understand what RESTful Web Services are
 - Understand the five principles behind RESTful Web Services
 - Understand the advantages and disadvantages of a RESTful approach.

REST in Five Steps*

A Short Primer

- ① Give everything an ID
- ② Link things together
- ③ Use standard HTTP methods
- ④ Support multiple representations
- ⑤ Use stateless communications

* Inspired by Stefan Tilkov: <http://www.innog.com/blog/st/presentations/2008/2008-03-13-REST-Intro-QCon-London.pdf>

REST in Five Steps

Give Everything an ID

- Use URIs for entity IDs

```
http://example.com/userDirectory/user/johnDoe  
http://example.com/userDirectory/user/{login}
```

`http://example.com/userDirectory/users`

REST in Five Steps

Link Things Together

- **Query**

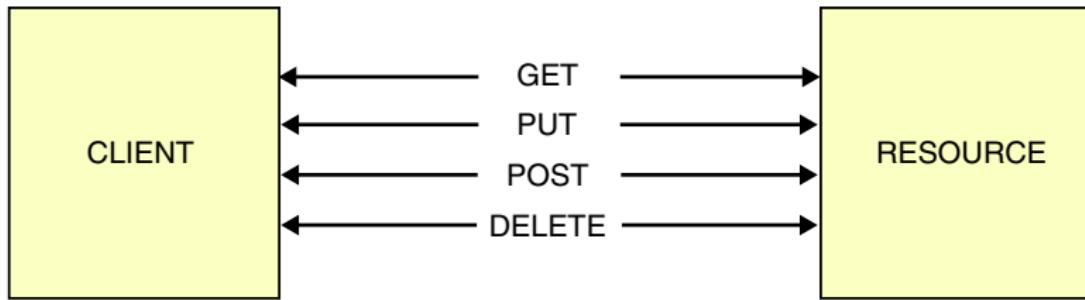
`http://example.com/userDirectory/users`

- Response

```
<customers>
  <customer ref="/userDirectory/user/johnDoe"/>
  <customer ref="/userDirectory/user/janeDoe"/>
  <customer ref="/userDirectory/user/jimKirk"/>
</customers>
```

REST in Five Steps

Use Standard HTTP Methods



REST in Five Steps

Use Standard HTTP Methods

- Leverage standard semantics for HTTP methods

Method	Purpose
GET	Read, possibly cached
POST	Update or create without a known ID
PUT	Update or create with a known ID
DELETE	Remove

REST in Five Steps

Support Multiple Representations

- Offer data in a variety of formats
 - > XML
 - > JSON
 - > (X)HTML
- Support content negotiation
 - > Accept header

GET /foo

Accept: application/json

- > URI-based

GET /foo.json

REST in Five Steps

Use Stateless Communications

- Long lived identifiers
- Avoid sessions
- Everything required to process a request contained in the request

HATEOAS Principle

- Hypermedia As The Engine Of Application State.
 - > REST applications should provide at most one fixed URL: all application URLs, with related resources, should be “dynamically” discovered, and navigated, starting from that fixed URL, thanks to the use of hypermedia links throughout all resources.

Common Patterns: Container, Item

Server in Control of URI Path Space

- List container contents: GET /container
- Add item to container: POST /container
 - > With item in request
 - > URI of item returned in HTTP response header
 - e.g. Location: `http://host/container/item`
- Read item: GET /container/item
- Update item: PUT /container/item
 - > With updated item in request
- Remove item: DELETE /container/item

Common Patterns: Map, Key, Value

Client in Control of URI Path Space

- List key-value pairs: GET /map
- Put new value to map: PUT /map/{key}
 - > With entry in request
e.g. PUT /map/dir/contents.xml
- Read value: GET /map/{key}
- Update value: PUT /map/{key}
 - > With updated value in request
- Remove value: DELETE /map/{key}

Advantages of a RESTful Approach

The Sun Cloud API Experience

- Reduced client coding errors.
- Reduced invalid state transition calls.
- Fine grained evolution without (necessarily) breaking old clients.

All of these could be said to be consequences of the HATEAOAS principle behind RESTful web services.

http://blogs.sun.com/craigmcc/entry/why_hateoas.

Advantages of a RESTful Approach

- Addressability: URLs.
- A uniform, constrained interface: HTTP Methods.
- A stable protocol: HTTP.
- Interoperability.
- Ubiquity.
- Familiarity.

<http://bill.burkecentral.com/2008/06/16/resteasy-mom-an-exercise-in-jax-rs-restful-ws-design/>

Key Benefits

Server Side

- Horizontal scaling
- Straightforward failover
- Cacheable
- Reduced coupling
- Works well with existing Web infrastructure

Key Benefits

Client Side

- Bookmarkable
- Easy to experiment in browser
- Broad programming language support
- Choice of data formats

Drawbacks of REST

- If the system is a very large one, then designing based on REST could become a very complex task.
 - > No direct bridge to the OOP world
 - > no standard formal language to describe interaction, *a la* WSDL
- Few tools.
- Restrictions for GET length sometimes may be a problem.
- Implementing Security on a REST system is an issue.



Developing Web Services with Java™ Technology

Module 8

RESTful Web Services: JAX-RS

Objectives

On completion of this module, you should:

- Understand how the five principles of RESTful web services map to JAX-RS constructs
 - Understand how to implement REST web services using JAX-RS
 - Understand how to deploy REST web services using Jersey, an implementation of JAX-RS

JAX-RS in Five Steps

JAX-RS defines constructs that mirror each of the five steps in our introduction to REST:

- ① Give everything an ID
 - ② Link things together
 - ③ Use standard HTTP methods
 - ④ Support multiple representations
 - ⑤ Use stateless communications

JAX-RS and Jersey

JAX-RS is only a specification. To implement RESTful web services, one also needs an implementation of JAX-RS:

- Project Jersey is the reference implementation of JAX-RS.

JAX-RS in Five Steps

Give Everything an ID

- All entities and actions that clients can refer to map to methods in some service provider POJO class.
 - `@Path` annotations on a method specify the URI that is mapped to that specific method.
 - > `@Path` annotations on a method, if any, are relative to the `@Path` annotation for its class, if any.
 - > all `@Path` annotations are relative to the application's deployment context.
 - > `@Path` annotations can include embedded parameters, which are mapped to parameters to the corresponding method call.

JAX-RS IDs

Simple Path

```
1 @Path("/airports")
2 public class AirportRM {
3     // ...
4     @Path("/numAirports")
5     public String getNumAirports() {
6         List<String> codes = dao.getAllCodes( null );
7         return
8             (codes == null) ? "0" : "" + codes.size();
9     }
}
```

GET /airports/numAirports

JAX-RS IDs

Path with Embedded Parameters

```
1  @Path("/airports")
2  public class AirportRM {
3      // ...
4      @Path("/nameByCode/{code}")
5      public String getNameByCode(
6          @PathParam("code") String code ) {
7          Airport airport = null;
8          try { airport = dao.findByName( null, code ); }
9          catch( Exception ex ) {
10              }
11          return
12              (airport != null) ? airport.getName() : "(not_found)";
13      }
```

GET /airports/nameByCode/JFK

JAX-RS IDs

Path with Form Parameters

```
1 @Path("/airports")
2 public class AirportRM {
3     // ...
4     @Path("/add")
5     public String addAirport(
6         @FormParam("code") String code,
7         @FormParam("name") String name ) {
8         Airport newAirport = null;
9         try { newAirport = dao.add( null, code, name ); }
10        catch( Exception ex ) {}
11        return (newAirport != null) ? "ok" : "fail";
12    }
```

POST /airports/add?code=LGA&name=LaGuardia

JAX-RS in Five Steps

Link Things Together

- Application-level logic is expected to provide information to the client incrementally, linking entities to each other.
 - Support classes are provided to make this more convenient.
 - > `UriInfo` is an injectable type that provides the application with convenience functions to obtain information about the current application and request URI.
 - > `UriBuilder` is a utility class that can build complex URIs with embedded parameters.

JAX-RS in Five Steps

Use Standard HTTP Methods: GET

```
1  @Path("/airports")
2  public class AirportRM {
3      @GET
4      @Path("/nameByCode/{code}")
5      public String getNameByCode(
6          @PathParam("code") String code ) {
7          Airport airport = null;
8          try { airport = dao.findByName( null, code ); }
9          catch( Exception ex ) {
10          }
11          return
12              (airport != null) ? airport.getName() : "(not found)";
13      }
```

JAX-RS in Five Steps

Use Standard HTTP Methods: POST

```
1  @Path("/airports")
2  public class AirportRM {
3      @POST
4      @Path("/add")
5      public String addAirport(
6          @FormParam("code") String code,
7          @FormParam("name") String name ) {
8          Airport newAirport = null;
9          try { newAirport = dao.add( null, code, name ); }
10         catch( Exception ex ) {}
11         return (newAirport != null) ? "ok" : "fail";
12     }
```

JAX-RS in Five Steps

Support Multiple Representations

```
1 @Path("/airports")
2 public class AirportRM {
3     @GET
4     @Path("/byCode/{code}")
5     @Produces({"application/xml", "application/json"})
6     public Airport getByCode(
7         @PathParam("code") String code ) {
8         return dao.findByCode( null, code );
9     }
```

- JAXB types can be presented in the proper form transparently.

JAX-RS Parameters

- Parameter and return types for JAX-RS services are easy to manage, if they are JAXB types

```
1  @XmlRootElement
2  public class Airport
3      implements Serializable {
4          private String code;
5          private String name;
6          public Airport() {}
7          public Airport( String code, String name ) {
8              this.code = code; this.name = name;
9          }
```

- Else, the application can provide marshall and unmarshall functions of its own.

JAX-RS in Five Steps

Use Stateless Communications

- Default lifecycle creates an instance of the root resource service provider POJO for each request.
 - > Stateless logic would be the only way to guarantee proper behavior.
- Two other choices of lifecycle are available:
 - > `@Singleton` on JAX-RS root resources indicates a single instance should be reused for all requests.
 - > `@PerSession` on JAX-RS root resources indicates a new instance should be created and reused for each client *session*.

Indicating JAX-RS Root Resources

The JAX-RS runtime looks for classes that implement *Application*:

- If such a class is found, it is used to determine what root resources (and providers) are associated with the application it describes.
- If no such class is found, all root resources and providers packaged as part of the application are deployed together.

Indicating JAX-RS Root Resources

Explicit Declaration of Root Resources

```
1  public class MyApplication
2      extends Application {
3          public Set<Class<?>> getClasses() {
4              Set<Class<?>> s = new HashSet<Class<?>>();
5              s.add(AirportRM.class);
6              return s;
7          }
8      }
```

Indicating JAX-RS Root Resources

Runtime Retrieval of Root Resources

- PackagesResourceConfig searches for any JAX-RS resources in any packages named explicitly, or any packages “contained” within them:

```
1  public class MyApplication
2      extends PackagesResourceConfig {
3          public MyApplication() {
4              super("com.example;other.package");
5          }
6      }
```

Deploying JAX-RS Services

- JAX-RS supports deploying JAX-RS services:
 - > To a web container, through a generic JAX-RS servlet
 - > To the JAX-WS runtime
 - > To the Grizzly framework for scalable server applications
- Jersey also supports deploying within a simple Java VM.

Deploying Within a Java VM

```
1 public class AirportRM {  
2     // ...  
3     static public void  
4     main(String[] args) throws IOException {  
5         String contextUrl =  
6             "http://localhost:8080/jaxrs";  
7         if (args.length > 0)  
8             contextUrl = args[1];  
9         HttpServer server =  
10            HttpServerFactory.create( contextUrl );  
11            server.start();  
12    }  
13 }
```

- No service providers need be instantiated!



Developing Web Services with Java™ Technology

Module 9

JAX-RS Web Service Clients

Objectives

On completion of this module, you should be able to:

- Create JAX-RS clients using URL and HttpURLConnection.
 - Create JAX-RS clients using the Jersey Client API.

Communicating with Web Servers

- Java provides a simple mechanism for communicating with HTTP servers , via URL objects and their associated URLConnections.
 - Jersey provides a client API, for convenient access to JAX-RS web services.
 - Third-party libraries, such as Apache's HttpClient, provide finer-grained access to HTTP servers.

Simplest Java Client

```
1 public class SimplestClient {
2     static public void main( String[] args )
3         throws Exception {
4     String contextURL = "http://localhost:8080/jaxrs";
5     String resourcePath = "/airports";
6     String requestPath = "/numAirports";
7     String urlString =
8         contextURL + resourcePath + requestPath;
9     URL url = new URL( urlString );
10    InputStream result = (InputStream) url.getContent();
11    Scanner scanner = new Scanner( result );
12    System.out.println( "Result:_" + scanner.next() );
13 }
14 }
```

PathParam Java Client

```
1  public class PathParamClient {  
2      static public void main( String[] args )  
3          throws Exception {  
4              String contextURL = "http://localhost:8080/jaxrs";  
5              String resourcePath = "/airports";  
6              String requestPath = "/nameByCode/";  
7              String param = "LGA"; // need URL-encoding  
8              String urlString =  
9                  contextURL + resourcePath + requestPath + param;  
10             URL url = new URL( urlString );  
11             InputStream result = (InputStream) url.getContent();  
12             BufferedReader reader =  
13                 new BufferedReader(new InputStreamReader(result));  
14             System.out.println( "Result:_" + reader.readLine() );  
15         }  
16     }
```

FormParam Java Client

```
1  public class FormParamClient {  
2      static public void main( String[] args )  
3          throws Exception {  
4      String contextURL = "http://localhost:8080/jaxrs";  
5      String resourcePath = "/airports";  
6      String requestPath = "/add";  
7      String code = "LGA";           // need URL-encoding  
8      String name = "LaGuardia";   // need URL-encoding  
9      String urlString =  
10         contextURL + resourcePath + requestPath;  
11      URL url = new URL( urlString );  
12      HttpURLConnection connection =  
13          (HttpURLConnection) url.openConnection();
```

- **URLConnection provides more control ...**

FormParam Java Client

```
14 connection.setRequestMethod( "POST" );
15 connection.setAllowUserInteraction( true );
16 connection.setDoOutput( true );
17 connection.setDoInput( true );
18 connection.connect();
19 OutputStream os = connection.getOutputStream();
20 PrintWriter writer = new PrintWriter( os );
21 writer.print( "code=" + code + "&name=" + name );
22 writer.close();
23 InputStream result = connection.getInputStream();
24 BufferedReader reader =
25     new BufferedReader( new InputStreamReader(result) );
26 System.out.println( "Result:_" + reader.readLine() );
27 }
28 }
```

Drawbacks of the Simple Approach

- Requires explicit matching of URL rewrite rules:
 - > To avoid invalid URLs, parameters may need to be provided using URL-encoding.
- Requires some awareness of the structure of HTTP messages
- Requires low-level I/O programming.

The Jersey Client API

The Jersey Client API revolves around two entities:

- WebResource instances represent JAX-RS resources.
 - Communications between the client and the JAX-RS resource are encapsulated within these instances.
- Client defines a configuration point for the Jersey runtime. It also acts as a factory for WebResources.

Simplest Jersey Client

```
1 public class SimplestJerseyClient {  
2     static public void main( String[] args ) {  
3         String contextURL = "http://localhost:8080/jaxrs";  
4         String resourcePath = "/airports";  
5         String requestPath = "/numAirports";  
6         String urlString =  
7             contextURL + resourcePath + requestPath;  
8         Client client = Client.create();  
9         WebResource resource =  
10            client.resource( urlString );  
11         String result = resource.get( String.class );  
12         System.out.println( "Result:" + result );  
13     }  
14 }
```

Customizing Request Message

WebResource allows the application to customize:

- HTTP method used by request
 - > Including payload, when method allows it.
- Request Headers
- Query Parameters
- Request Cookies

QueryParam Jersey Client

```
1  public class QueryParamJerseyClient {  
2      static public void main( String[] args ) {  
3          String contextURL = "http://localhost:8080/jaxrs";  
4          String resourcePath = "/airports";  
5          String requestPath = "/codeByName";  
6          String name = "LaGuardia"; // No URL-Encoding!  
7          String urlString =  
8              contextURL + resourcePath + requestPath;  
9          Client client = Client.create();  
10         WebResource resource =  
11             client.resource( urlString );  
12         String result =  
13             resource.queryParam("name", name).get(String.class);  
14         System.out.println( "Result:" + result );  
15     }  
16 }
```

WebResource.get

- WebResource.get accepts a type (or a list of types) to use when creating the value to return to the caller. The types accepted as parameters include:
 - > Classes with a constructor that accepts a single String.
 - > JAXB classes.
- It is also possible to specify the representation that the client expects for the payload of the reply message.

Specifying Expected Return Type

```
1 public class JSONObjectJerseyClient {  
2     static public void main( String[] args ) {  
3         String urlString =  
4             "http://localhost:8080/jaxrs/airports/byCode/LGA";  
5         Client client = Client.create();  
6         WebResource resource =  
7             client.resource(urlString);  
8         Airport result =  
9             resource  
10            .accept( "application/json" )  
11            .get( Airport.class );  
12         System.out.println( "Result:" + result );  
13     }  
14 }
```

Submitting Form Data

```
1 public class FormSubmitJerseyClient {  
2     static public void main( String[] args ) {  
3         String url = "http://localhost:8080/jaxrs/airports/add";  
4         Client client = Client.create();  
5         WebResource resource = client.resource(url);  
6         MultivaluedMap<String, String> params =  
7             new MultivaluedMapImpl();  
8         params.add( "code", "JFK" );  
9         params.add( "name", "John_F._Kennedy_Airport" );  
10        String result =  
11            resource  
12                .type( "application/x-www-form-urlencoded" )  
13                .post( String.class, params );  
14        System.out.println( "Result:" + result );  
15    }  
16 }
```

Obtaining Reply Metadata

`ClientResource` represents the complete reply message received by the client. Its API allows the application to access:

- Status code
- Message payload
- Response Headers
- Response Cookies

Obtaining Reply Metadata

```
1  public class ClientResponseJerseyClient {  
2      static public void main( String[] args ) {  
3          String urlString =  
4              "http://localhost:8080/jaxrs/airports/byCode/LGA";  
5          Client client = Client.create();  
6          WebResource resource = client.resource(urlString);  
7          ClientResponse response =  
8              resource.accept( "application/json" )  
9                  .get( ClientResponse.class );  
10         System.out.println("Code:" +  
11                         response.getStatus());  
12         System.out.println("Result:" +  
13                         response.getEntity(Airport.class));  
14     }  
15 }
```

Jersey Client API Filters

- Filters allow clients to perform common operations on any JAX-RS interaction, independent of which particular interaction they each are.
- Similar to `ServletFilter` in the Servlet specification, or `Handler` in the JAX-WS specification.

Simple Client with Logging Filter

```
1  public class LoggingClient {  
2      static public void main( String[] args ) {  
3          String urlString =  
4              "http://localhost:8080/jaxrs/airports/byCode/LGA";  
5          Client client = Client.create();  
6          client.addFilter( new LoggingFilter() );  
7          WebResource resource = client.resource(urlString);  
8          Airport result =  
9              resource  
10             .accept( "application/json" )  
11             .get( Airport.class );  
12          System.out.println( "Result:" + result );  
13      }  
14  }
```



Developing Web Services with Java™ Technology

Module 10

JAX-RS and JavaEE

Objectives

On completion of this module, you should:

- Understand how to deploy POJO web services to a web container
 - Understand how to define a web service in terms of an Enterprise Java Bean
 - Understand how to deploy an EJB web service to an EJB container
 - Describe the benefits associated with implementing a web service as an EJB

A Simple POJO Web Service

```
1  @Path("/airports")
2  public class AirportRM {
3      // ...
4      @GET
5      @Path("/nameByCode/{code}")
6      public String getNameByCode(
7          @PathParam("code") String code ) {
8          Airport airport = null;
9          try { airport = dao.findByName( null, code ); }
10         catch( Exception ex ) {
11             }
12         return
13         (airport != null) ? airport.getName() : "(not_found)";
14     }
15     private AirportDAO dao = new AirportDAO();
16 }
```

Deploying a Web Service on JavaSE

```
1 public class AirportRM {  
2     // ...  
3     static public void  
4     main(String[] args) throws IOException {  
5         String contextUrl =  
6             "http://localhost:8080/jaxrs";  
7         if (args.length > 0)  
8             contextUrl = args[1];  
9         HttpServer server =  
10            HttpServerFactory.create( contextUrl );  
11            server.start();  
12    }  
13 }
```

Limitations of JavaSE Deployment

- HTTP Server built into JavaSE not designed for production use:
 - > Limited support for scalability
 - Thread pooling.
 - HTTP sessions in memory only.
 - > Limited infrastructure for security
 - Support for encryption HTTPS.
 - No builtin support for authentication and authorization.
 - Is there an alternative?

Deploying to a Web Container

- JAX-RS Web services can leverage a web container's infrastructure to process HTTP messages:
 - > The web container hands incoming HTTP requests to a servlet that feeds the JAX-RS runtime within the web container.
 - > As a convenience to the developer, JAX-RS in Glassfish will allow the developer to ignore the actual servlet responsible for initial processing of incoming HTTP requests.
 - But this is not currently supported.

Deploying to a Web Container

Simple Approach

- By default, Jersey will deploy all resources (and providers) found within the (web) application.
 - An instance of a class that extends `Application` can be created, to control more precisely which of the resources currently within the (web) application to deploy.
 - If necessary, a `servlet` element naming the `Application` subclass as the name of a servlet can be used. Once present, that name can be used for further configuration.

Deploying to a Web Container

Alternative Approach #1

```
1 <servlet>
2     <servlet-name>ServletAdaptor</servlet-name>
3     <servlet-class>
4         com.sun.jersey.spi.container.servlet.ServletContainer
5     </servlet-class>
6     <init-param>
7         <param-name>javax.ws.rs.Application</param-name>
8         <param-value>com.example.jaxrs.MyApp</param-value>
9     </init-param>
10    </servlet>
11    <servlet-mapping>
12        <servlet-name>ServletAdaptor</servlet-name>
13        <url-pattern>/resources/*</url-pattern>
14    </servlet-mapping>
```

Deploying to a Web Container

Alternative Approach #2

```
1 <servlet>
2   <servlet-name>Jersey Web Application</servlet-name>
3   <servlet-class>
4     com.sun.jersey.spi.container.servlet.ServletContainer
5   </servlet-class>
6   <init-param>
7     <param-name>
8       com.sun.jersey.config.property.packages
9     </param-name>
10    <param-value>com.example;other.package</param-value>
11  </init-param>
12 </servlet>
```

Additional Features

Logging

```
1 <servlet>
2   <servlet-name>Jersey Web Application</servlet-name>
3   <servlet-class>
4     com.sun.jersey.spi.container.servlet.ServletContainer
5   </servlet-class>
6   <init-param>
7     <param-name>
8       com.sun.jersey.spi.container.ContainerRequestFilters
9     </param-name>
10    <param-value>
11      com.sun.jersey.api.container.filter.LoggingFilter
12    </param-value>
13  </init-param>
```

- You could also specify

com.sun.jersey.spi.container.ContainerResponseFilters

Deploying to a Web Container

Packaging and Deployment

- Package into the WAR file the web service implementation class and all its supporting structure as if it were a servlet.
- Deploy WAR file to web container as usual

Deploying to a Web Container

- Advantages:
 - > Security infrastructure built in
 - > HTTP session management built in
 - > Support for scalability and availability built in (though some support is built into JAX-RS anyway)
- Disadvantages:
 - > Transaction management still up to the application
 - > More complex deployment
 - > Requires a web container

Incorporating Security Constraints

- Best practices for security address these security principles:
 - > Maintaining Corporate Security Policies
 - > Self-Preservation
 - > Defense in Depth
 - > Least Privilege
 - > Compartmentalization
 - > Proportionality
- Least Privilege requires authentication and authorization constraints on services.

Specifying Security Constraints

A Secured Web Service

```
1  @Path("/secureAirports")
2  public class SecureAirportRM {
3      @POST
4      @Path("/add")
5      @RolesAllowed("administrator")
6      public String addAirport(
7          @FormParam("code") String code,
8          @FormParam("name") String name ) {
9          Airport newAirport = null;
10         try {
11             newAirport = dao.add( null, code, name );
12         }
13         catch( Exception ex ) {}
14         return (newAirport != null) ? "ok" : "fail";
15     }
```

Configuring the Web Application

Unfortunately, web containers need a little help to get our SecureAirportManager to work as intended:

- The `web.xml` file has to specify:
 - > that there are protected URLs to be considered
 - > which authentication protocol to use
 - > which roles to expect
- The server-specific configuration has to specify:
 - > the mapping from roles expected to principals that can assume that role
 - > the set of principals known.

Configuring web.xml

Securing Web Service URLs

```
1   <security-constraint>
2       <display-name>SecureAirportRM</display-name>
3       <web-resource-collection>
4           <web-resource-name>
5               SecureAirportRM
6           </web-resource-name>
7           <url-pattern>/resources</url-pattern>
8           <http-method>GET</http-method>
9           <http-method>POST</http-method>
10      </web-resource-collection>
11      <auth-constraint>
12          <role-name>administrator</role-name>
13          <role-name>client</role-name>
14      </auth-constraint>
15  </security-constraint>
```

Configuring web.xml

Configuring Login

```
16 <login-config>
17     <auth-method>BASIC</auth-method>
18     <realm-name>file</realm-name>
19 </login-config>
20 <security-role>
21     <role-name>client</role-name>
22 </security-role>
23 <security-role>
24     <role-name>administrator</role-name>
```

Configuring sun-web.xml

```
1 <sun-web-app error-url="">
2   <context-root>/JavaEEJAXRS</context-root>
3   <security-role-mapping>
4     <role-name>client</role-name>
5     <principal-name>tracy</principal-name>
6   </security-role-mapping>
7   <security-role-mapping>
8     <role-name>administrator</role-name>
9     <principal-name>kelly</principal-name>
10  </security-role-mapping>
```

- We could have mapped a role-name to a *group*, instead of listing the individual principals here...

Retrieving Security Information

- Sometimes, the simple role-based access control strategy is not enough.
- Applications can add further authorization constraints *programmatically*:
 - it may be necessary to obtain the specific identity of the caller, and not just the role (or group) they belong to:
 - JAX-RS uses *injection*

Retrieving Security Information

A Servlet

```
1 @WebServlet(name="SecureServlet",
2             urlPatterns={"/SecureServlet"})
3 public class SecureServlet extends HttpServlet {
4     protected void
5         processRequest(HttpServletRequest request,
6                         HttpServletResponse response)
7     throws ServletException, IOException {
8         response.setContentType("text/html;charset=UTF-8");
9         PrintWriter out = response.getWriter();
10        String user = request.getRemoteUser();
11        Principal principal = request.getUserPrincipal();
```

Retrieving Security Information

- The API that allows a Servlet to retrieve security information is built into one of the parameters to its service() call
 - > they are always available
 - The only arguments provided to a web service method are those required by its application-level service definition. How to match the API given to a Servlet?
 - > Context objects provide useful services to components
 - > How can a context object be provided to a web service?

Retrieving Security Information

Dependency Injection

```
1 @Path("/secureAirports")
2 public class SecureAirportRM {
3     // ...
4     private AirportDAO dao = new AirportDAO();
5     @Context SecurityContext secContext;
6     @Context ServletContext webContext;
7 }
```

- The annotation `@Context` indicates to the JAX-RS runtime that it must inject these values.

Retrieving Security Information

Logging Callers

```
1  @Path("/add")
2  @RolesAllowed("administrator")
3  public String addAirport(
4      @FormParam("code") String code,
5      @FormParam("name") String name ) {
6      Airport newAirport = null;
7      try {
8          newAirport = dao.add( null, code, name );
9          webContext.log("add:" + 
10                  secContext.getUserPrincipal());
11      }
12      catch( Exception ex ) {}
13      return (newAirport != null) ? "ok" : "fail";
14 }
```

Accessing the Web Infrastructure

- The following JAX-WS entities can be injected using `@Context`

SecurityContext HttpHeaders

Request UriInfo

- The following web infrastructure entities can be injected using `@Context`

ServletConfig ServletContext

HttpServletRequest HttpServletResponse

Simple Jersey Client

```
1 public class SimplestJerseyClient {  
2     static public void main( String[] args ) {  
3         String contextURL = "http://localhost:8080/jaxrs";  
4         String resourcePath = "/airports";  
5         String requestPath = "/numAirports";  
6         String urlString =  
7             contextURL + resourcePath + requestPath;  
8         Client client = Client.create();  
9         WebResource resource =  
10            client.resource( urlString );  
11         String result = resource.get( String.class );  
12         System.out.println( "Result:" + result );  
13     }  
14 }
```

Authenticating Jersey Client

```
1 public class AuthenticatingJerseyClient {  
2     static public void main( String[] args ) {  
3         String contextURL = "http://localhost:8080/jaxrs";  
4         String resourcePath = "/airports";  
5         String requestPath = "/numAirports";  
6         String urlString =  
7             contextURL + resourcePath + requestPath;  
8         Client client = Client.create();  
9         ClientFilter authFilter =  
10            new HTTPBasicAuthFilter("login", "password");  
11         client.addFilter(authFilter);  
12         WebResource resource =  
13             client.resource( urlString );  
14         String result = resource.get( String.class );
```

A POJO Web Service Using a DAO

```
1  @Path("/airports")
2  public class AirportRM {
3      // ...
4      @GET
5      @Path("/byCode/{code}")
6      @Produces({"application/xml", "application/json"})
7      public Airport getByCode(
8          @PathParam("code") String code) {
9          return dao.findByCode( null, code );
10     }
11     private AirportDAO dao = new AirportDAO();
12 }
```

- Does it need to manage persistence?

A More Complex POJO Web Service

```
1  @Path("/airports")
2  public class AirportRM {
3      @DELETE
4      @Path("/removeByCode/{code}")
5      public void removeByCode(
6          @PathParam("code") String code ) {
7          EntityManager em =
8              GenericDAO.getEMF().createEntityManager();
9          EntityTransaction tx = em.getTransaction();
10         tx.begin();
11         dao.remove(em, dao.findByName(em, code));
12         tx.commit();
13         em.close();
14     }
```

- This had to be one transaction...

Limitations of a POJO Web Service

- A POJO Web Service endpoint is “just a POJO” – and so shares some limitations of all POJOs:
 - > explicit transaction management
 - > explicit persistence context management
- By default, POJO web services are stateless, and a new instance is allocated for each call:
 - > this is an advantage from the point of view of concurrency
 - > this is a disadvantage from the point of view of scalability

JAX-RS provides mechanisms to manage this...

Enterprise Java Beans

Goals of the Enterprise Java Beans specification include:

- The Enterprise JavaBeans architecture will support the development, deployment, and use of web services.
- The Enterprise JavaBeans architecture will make it easy to write applications: application developers will not have to understand low-level transaction and state management details, multi-threading, connection pooling, or other complex low-level APIs.

Advantages of EJBs

EJBs are “just” *smart* POJOs:

- Approaches to improve scalability can be declarative:
 - > Service instance lifecycle choice: shared stateless pool, shared singleton, per-client.
 - > Concurrency management built-in: stateless and stateful (per-client) instances guarantee safe concurrent execution, while singletons support declarative concurrency control.
- Transaction management can be declarative
 - > Persistence contexts can be managed transparently (one context per transaction), or programmatically.

Creating Web Services from EJBs

```
1  @Path("/airportsSLSB")
2  @Stateless
3  public class AirportRM {
4      // ...
5      @GET
6      @Path("/numAirports")
7      public String getNumAirports() {
8          List<String> codes = dao.getAllCodes();
9          return
10             (codes == null) ? "0" : "" + codes.size();
11     }
12     @EJB private AirportDAO dao;
13 }
```

Injecting EJBs

```
1  @Path("/airportsSLSB")
2  @Stateless
3  public class AirportRM {
4      @GET
5      @Path("/numAirports")
6      public String getNumAirports() {
7          List<String> codes = dao.getAllCodes();
8          return
9              (codes == null) ? "0" : "" + codes.size();
10     }
11     @EJB private AirportDAO dao;
12 }
```

- AirportRM obtains its DAO via *injection*
 - > whether/how that AirportDAO is shared is transparent!

A More Complex EJB Web Service

```
1  @GET
2  @Path("/numAirports")
3  public String getNumAirports() {
4      List<String> codes = dao.getAllCodes();
5      return
6          (codes == null) ? "0" : "" + codes.size();
7  }
8  @DELETE
9  @Path("/removeByCode/{code}")
10 public void removeByCode(
11     @PathParam("code") String code ) {
12     dao.remove(dao.findByCode(code));
13 }
14 @EJB private AirportDAO dao;
```

Types of EJBs

There are three types of EJBs that can also be web service endpoints:

- Stateless
- Singleton
- Stateful – but not interoperable

There is fourth type of EJB – Message-Driven Beans.

Stateless Session Beans

Stateless session beans are pools of shared objects:

- any one instance can only be used by one client at a time
- instances return to the pool after each use

Pros and Con: Stateless EJBs

- Advantages:
 - > Improved scalability through shared pool of instances
 - > Transparent transaction management
- Disadvantages:
 - > Increased network overhead when engaging in conversations

Singleton EJBs

Singleton beans are single instances per JVM:

- all clients share the same instance
- a concurrency control policy must be specified

Singleton Web Service EJBs

```
1  @Path("/airportsSingletonEJB")
2  @Singleton
3  public class SingletonAirportRM {
4      // ...
5      @GET
6      @Path("/numAirports")
7      public String getNumAirports() {
8          List<String> codes = dao.getAllCodes();
9          return
10             (codes == null) ? "0" : "" + codes.size();
11     }
12     @EJB private AirportDAO dao;
13 }
```

- Note that the annotation is javax.ejb.Singleton

Concurrency Management

Two strategies are available:

- Container-managed concurrency
 - > Container will use read/write locks to control concurrent access to the singleton
 - > Developer describes read/write behavior of each method declaratively
- Bean-managed concurrency
 - > Developer uses Java mechanisms (such as **synchronized** blocks) to control concurrent access to the singleton.

Singleton Web Service EJBs

```
1  @Path("/airportsSingletonEJB")
2  @Singleton
3  @Lock(LockType.READ)
4  public class SingletonAirportRM {
5      @POST
6      @Path("/add")
7      @Lock(LockType.WRITE)
8      public String addAirport(
9          @FormParam("code") String code,
10         @FormParam("name") String name) {
11     Airport newAirport = null;
12     try { newAirport = dao.add(code, name); }
13     catch( Exception ex ) {}
14     return (newAirport != null) ? "ok" : "fail";
15 }
```

Pros and Cons: Singletons

- Advantages:
 - > Convenient caching of shared state
 - > Transparent transaction management
- Disadvantages:
 - > Increased network overhead when engaging in conversations
 - > Possible bottleneck due to concurrency control policy



Developing Web Services with Java™ Technology

Module 11

More Complex Services Using JAX-RS and Jersey

Objectives

On completion of this module, you should:

- Understand how to produce and consume custom types.
- Define JAX-RS web services that provide results by linking to other resources.
- Understand how to manage exceptions.
- Define JAX-RS web services in terms of resources and sub-resources.
- Understand the different scopes defined by JAX-RS for web services endpoints

Parameters and Return Types

- Simple types can be accepted as parameters – but only `String` is an acceptable simple return type.
- Complex types that are JAXB-compliant are marshalled and unmarshalled into XML using JAXB.
 - JAX-RS will also marshall and unmarshall JAXB-compliant objects into JSON, building on the JAXB infrastructure.
- Other types can be marshalled and unmarshalled.

Simple Types as Parameters

Valid parameter types in general include:

- ① Primitive types
 - ② Types that have a constructor that accepts a single String argument
 - ③ Types that have a static method named valueOf() with a single String argument
 - ④ List<T>, Set<T>, or SortedSet<T>, where T satisfies 2 or 3 above.

Return Types

Valid return types in general include:

- **byte[]** and **String**
 - Application-supplied JAXB classes and
`javax.xml.bind.JAXBElement`
 - `MultivaluedMap<String, String>`
 - Response **and** **GenericEntity**
 - Allows the application to control the HTTP return code and payload of the return message.

Returning a Supported Complex Type

```
1  @Path("/airports")
2  public class AirportRM {
3      // ...
4      @GET
5      @Path("/byCode/{code}")
6      @Produces({"application/xml", "application/json"})
7      public Airport getByCode(
8          @PathParam("code") String code ) {
9          return dao.findByCode( null, code );
10     }
11     private AirportDAO dao = new AirportDAO();
12 }
```

Returning a Supported Complex Type

JAXB Application Type

```
1 @Entity
2 @XmlRootElement
3 public class Airport
4     implements Serializable {
5     private String code;
6     private String name;
7     public Airport() {}
8     public Airport( String code, String name ) {
9         this.code = code; this.name = name;
10    }
```

Controlling Responses

- Response **instances** grant finer control of responses:
 - > **HTTP Response type:** ok(), noContent(), created(), notModified()
 - > **Response Payload:** entity()
- Response **implements Builder pattern**
 - > build() must be invoked to **create** Response
- UriInfo and UriBuilder **can help build URIs.**

Providing Location of New Item

```
1  @Path("/betterAirports")
2  public class BetterAirportRM {
3      @POST
4      @Path("/add")
5      public Response addAirport(
6          @FormParam("code") String code,
7          @FormParam("name") String name ) {
8          Airport newAirport = dao.add(null, code, name);
9          UriBuilder ub = uriInfo.getBaseUriBuilder();
10         URI uri =
11             ub.path(AirportRM.class, "getByCode").build(code);
12         return Response.created(uri).entity(newAirport).build();
13     }
14     private AirportDAO dao = new AirportDAO();
15     @Context UriInfo uriInfo;
```

Another Complex Return Type

```
1 @Path("/betterAirports")
2 public class BetterAirportRM {
3     @Path("/listComplete")
4     @GET
5     @Produces({"application/xml"})
6     public List<Airport>
7     listComplete() {
8         List<Airport> result = dao.list(null);
9         return result;
10    }
11    private AirportDAO dao = new AirportDAO();
```

- But this is not the REST way ...

Complex Values – the REST Way

- Getting too much information

```
<airports>
  <airport>
    <id>1</id><version>1</version>
    <code>MCO</code><name>Orlando</name>
  </airport>
</airports>
```

- The REST way

```
<airports>
  <airport ref="/airports/byCode/MCO">MCO</airport>
</airports>
```

Using JAXB – the REST Way

```
1 package com.example.jaxrs.resources.helpers;
2 @XmlRootElement(name="airport")
3 public class Airport {
4     @XmlValue public String code;
5     @XmlAttribute public String ref;
6     @XmlTransient public com.example.traveller.model.Airport airport;
7     public Airport() {}
8     public
9     Airport(com.example.traveller.model.Airport a, UriBuilder ub) {
10        code = a.getCode();
11        ref = buildRef(a,ub);
12        airport = a;
13    }
14    private String
15    buildRef(com.example.traveller.model.Airport a, UriBuilder ub) {
16        URI result =
17            ub.path(BetterAirportRM.class,"getByCode").build(a.getCode());
18        return result.toString();
19    }
20 }
```

Another Complex Return Type

The REST Way

```
1  @Path("/betterAirports")
2  public class BetterAirportRM {
3      @Path("/list")
4      @GET
5      @Produces({"application/xml","application/json"})
6      public List<com.example.jaxrs.resources.helpers.Airport>
7      listWithLinks() {
8          UriBuilder ub = uriInfo.getBaseUriBuilder();
9          List<Airport> candidates = dao.list(null);
10         List<com.example.jaxrs.resources.helpers.Airport> result =
11             new ArrayList<com.example.jaxrs.resources.helpers.Airport>();
12         for (Airport a : candidates) {
13             com.example.jaxrs.resources.helpers.Airport proxy =
14                 new com.example.jaxrs.resources.helpers.Airport(a, ub);
15             result.add(proxy);
16         }
17         return result;
18     }
```

Default Response

- It is possible to introduce a method that is the default response
 - > when the incoming URL matches the path to the resource class, only

```
1 @Path("/betterAirports")
2 public class BetterAirportRM {
3     @GET
4     @Produces({"application/xml","application/json"})
5     public List<com.example.jaxrs.resources.helpers.Airport>
6     getDefault() {
7         return listWithLinks();
8     }
```

Entity Providers

- In general, the mappings between Java objects and their external representations are the responsibility of *Entity Providers*.
- An Entity Provider is a class that:
 - implements the `MessageBodyReader` or `MessageBodyWriter` interface
 - is annotated with `@Provider`
- Providers are singletons, and so must be thread-safe.

A Complex Type Not Supported

```
1  @Path("/planner")
2  public class PlannerRM {
3      @Path("routesSummary") @GET
4      @Produces({"application/xml", "application/json"})
5      public Map<String, String>
6      getRoutesFromSummary(@QueryParam("start") String code) {
7          Airport start = airportDAO.findByCode(null, code);
8          Map<Flight, Airport> candidates =
9              flightDAO.findRoutesFrom(null, start);
10         Map<String, String> result = new HashMap<String, String>();
11         for(Flight f : candidates.keySet()) {
12             Airport dest = candidates.get(f);
13             result.put(f.getNumber(), dest.getCode());
14         }
15         return result;
16     }
```

MessageBodyWriter

- A MessageBodyWriter has to implement:
 - getSize
Called to determine the size of the response. A non-negative value is used for Content-Length.
 - isWriteable
When registered, a MessageBodyWriter indices a “willingness” to handle particular type/mediaType combinations. This function is called to check whether this writer can actually handle a particular request.
 - writeTo
Called to actually convert the value to be returned to its appropriate representation by way of this writer.
- MessageBodyReaders need equivalent functions.

A MessageBodyWriter for Maps

```
1 @Provider
2 @Produces(MediaType.APPLICATION_XML)
3 public class MapMessageBodyWriter
4 implements MessageBodyWriter<Map<String, String>>{
5     // ...
6     @Override
7     public long getSize(Map<String, String> m, Class<?> type,
8             Type genericType, Annotation[] annotations,
9             MediaType mediaType) {
10        return -1;
11    }
```

A MessageBodyWriter for Maps

```
1  @Provider
2  @Produces(MediaType.APPLICATION_XML)
3  public class MapMessageBodyWriter
4  implements MessageBodyWriter<Map<String, String>>{
5      public boolean isWriteable(Class<?> type, Type generic,
6          Annotation[] annotations, MediaType mediaType) {
7      MediaType targetMedia = MediaType.APPLICATION_XML_TYPE;
8      return
9          (generic.equals(targetType()) &&
10             mediaType.isCompatible(targetMedia));
11     }
12     private Type targetType() {
13         Type[] types = getClass().getGenericInterfaces();
14         ParameterizedType myIf = (ParameterizedType) types[0];
15         return myIf.getActualTypeArguments()[0];
16     }
```

A MessageBodyWriter for Maps

```
17  public void writeTo(Map<String, String> m,
18      Class<?> type, Type genericType,
19      Annotation[] annotations, MediaType mediaType,
20      MultivaluedMap<String, Object> httpHeaders,
21      OutputStream entityStream)
22  throws IOException, WebApplicationException {
23      StringBuffer sb = new StringBuffer("<map>");
24      for (Map.Entry<String, String> entry : m.entrySet()) {
25          String key = entry.getKey(), val = entry.getValue();
26          sb.append("<entry>");
27          sb.append("<key>").append(key).append("</key>");
28          sb.append("<value>").append(val).append("</value>");
29          sb.append("</entry>");
30      }
31      sb.append("</map>");
32      entityStream.write(sb.toString().getBytes());
33 }
```

Throwing Exceptions

JAX-RS resources may throw exceptions:

- Instances of `WebApplicationException` may be mapped directly to a Response
- If *exception mappers* are available, the most appropriate one must be used to generate a Response
- Unchecked exceptions will be thrown to the container
- Checked exceptions that cannot be thrown will be wrapped in a container-specific exception, then thrown to the container.

A Resource that Throws Exceptions

```
1 @Path("/betterAirports")
2 public class BetterAirportRM {
3     @POST
4     @Path("/add")
5     public Response addAirport(
6         @FormParam("code") String code,
7         @FormParam("name") String name ) {
8         Airport newAirport = dao.add(null, code, name);
9         UriBuilder ub = uriInfo.getBaseUriBuilder();
10        URI uri =
11            ub.path(AirportRM.class, "getByCode").build(code);
12        return Response.created(uri).entity(newAirport).build();
13    }
```

- DAO methods may throw PersistenceException

An ExceptionMapper

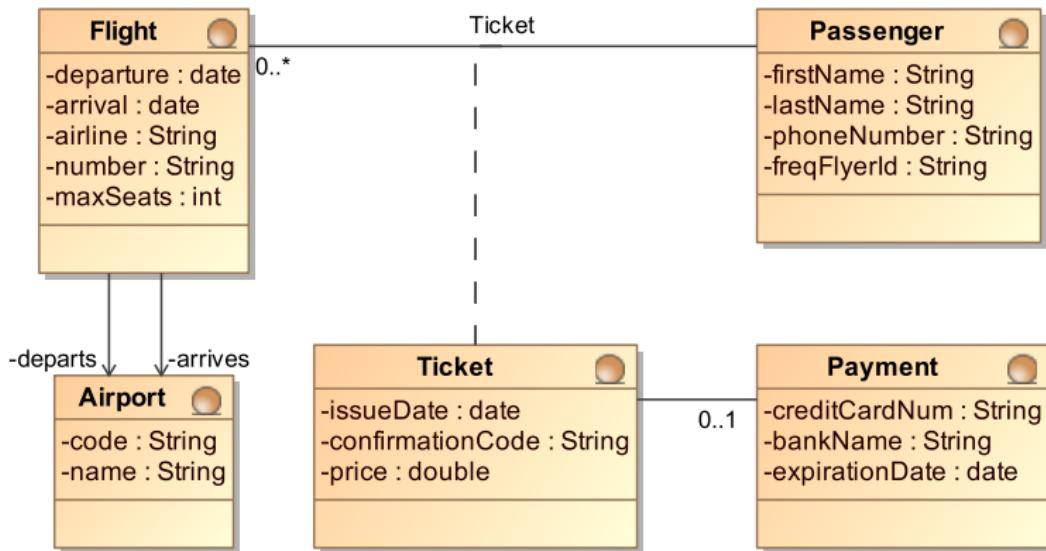
```
1 @Provider
2 public class DAOExceptionMapper
3     implements ExceptionMapper<PersistenceException> {
4     public Response toResponse(PersistenceException ex) {
5         if (ex instanceof EntityExistsException) {
6             return Response.notAcceptable(null)
7                 .header("DAO-Message", ex)
8                 .build();
9         } else {
10             return
11                 Response.serverError().header("DAO-Message", ex)
12                 .build();
13         }
14     }
15 }
```

Using Resources and Sub-resources

- We had seen how simple JAX-RS web services are implemented through a single resource class, whose methods are mapped to URI requests, invoked via HTTP.
- In more complex applications, a single resource class would not be enough. JAX-RS handles this by introducing the notion of a *sub-resource*: a resource that is not directly accessible from a client, but rather must be accessed relative to some other resource.

Traveller Project

Domain (Recap)



Simple Resource Classes

- Resource classes like `AirportRM` are *self-sufficient*:
 - > They can answer any relevant queries directly.
 - > All methods have both an HTTP action and a `@Path`
 - A method with an HTTP action but no `@Path` matches an empty path.
 - > Methods in such classes are known as *sub-resource methods*.
- Not every resource class can be implemented so.

A Simple Resource Class

```
1  @Path("/airports")
2  public class AirportRM {
3      @GET
4      @Path("/numAirports")
5      public String getNumAirports() {
6          List<String> codes = dao.getAllCodes( null );
7          return
8              (codes == null) ? "0" : "" + codes.size();
9      }
10     @GET
11     @Path("/byCode/{code}")
12     @Produces({"application/xml","application/json"})
13     public Airport getByCode(
14         @PathParam("code") String code ) {
15         return dao.findByCode( null, code );
16     }
17     private AirportDAO dao = new AirportDAO();
18 }
```

Resource Classes and Sub-Resources

- JAX-RS does not require that every resource class be self-sufficient:
 - Classes with a `@Path` annotation can be referenced directly by clients.
 - Classes without such an annotation can only be referenced through another controlling resource.
 - These classes are known as *sub-resources*
- To delegate to a sub-resource, a resource class defines a method that
 - Returns an instance of the sub-resource class, and
 - Does **not** have an HTTP method annotation.

A More Complex Resource Class ...

```
1  @Path("/flights")
2  public class FlightRM {
3      @GET @Path("/byNumber/{number}")
4      @Produces({"application/xml", "application/json"})
5      public Flight getByNumber(
6          @PathParam("number") String number) {
7          return dao.findByNumber(number);
8      }
9      private FlightDAO dao = new FlightDAO();
10 }
```

- This is not very RESTful
 - > Could we ask for the departure airport for a flight?

... and A Sub-Resource

```
1 @Path("/flights")
2 public class FlightRM {
3     @Path("/byNumber/{number}/departs")
4     public AirportResource getDepartsByNumber(
5         @PathParam("number") String number) {
6         Flight flight = dao.findByNumber(number);
7         return new AirportResource(flight.getDeparts());
8     }
```

- Methods without an HTTP action allow a resource class to delegate to a dependent resource
 - > Such methods are known as *sub-resource locators*
 - > FlightRM would rather not know how to present Airports

The Sub-Resource Class

```
1  public class AirportResource {  
2      AirportResource(Airport airport) {  
3          this.airport = airport;  
4      }  
5      @GET @Produces({"application/xml","application/json"})  
6      public com.example.jaxrs.resources.helpers.Airport getDefault() {  
7          UriBuilder ub = uriInfo.getBaseUriBuilder();  
8          return  
9              new com.example.jaxrs.resources.helpers.Airport(airport,ub);  
10     }  
11     @GET @Path("/code") public String getCode() {  
12         return airport.getCode();  
13     }  
14     @GET @Path("name") public String getName() {  
15         return airport.getName();  
16     }  
17     private Airport airport;  
18     @Context UriInfo uriInfo;  
19 }
```

Valid Queries

- /flights/byNumber/AA123

```
<flight>
  <number>AA123</number>
  <departs>
    <airport><code>MCO</code><name>Orlando</name></airport>
  </departs>
  ...
</flight>
```

- /flights/byNumber/AA123/departs

```
<airport><code>MCO</code><name>Orlando</name></airport>
```

- /flights/byNumber/AA123/departs/code

MCO

Resource Scopes

- Interactions in JAX-RS are *stateless* by default:
 - > Resource instances can be discarded after each interaction safely.
 - > A single instance could be reused, if:
 - No state is kept in the instance between invocations.
 - Instance is thread safe.
- Sometimes, *stateful* interactions are more convenient.
- JAX-RS provides several *resource scopes*, to accomodate these choices.

Request Scope

- Default scope for resource instances is *request* scope:
 - No annotation is needed.
 - New instances of resource classes are allocated for each request.
 - dependency injection is performed once request processing starts, but before resource instance is invoked.
 - Instances are discarded after each request is completed.
- But remember that providers are always singletons!

A Request-Scoped Resource

```
1  @Path("/secureAirports")
2  public class SecureAirportRM {
3      @POST
4      @Path("/add")
5      public String addAirport(
6          @FormParam("code") String code,
7          @FormParam("name") String name ) {
8          Airport newAirport = null;
9          try {
10              newAirport = dao.add( null, code, name );
11              webContext.log("add:_" +
12                           secContext.getUserPrincipal());
13          }
14          catch( Exception ex ) {}
15          return (newAirport != null) ? "ok" : "fail";
16      }
17      private AirportDAO dao = new AirportDAO();
18      @Context SecurityContext secContext;
19      @Context ServletContext webContext;
```

The Singleton Scope

Resource instances can have *singleton* scope:

- Application can return a list of singletons.
- Resource class can be annotated with Singleton
- Characteristics:
 - > Single instances are allocated, at the time the application is initialized.
 - > Singletons must be thread-safe.
 - > Field-level dependency injection is performed once.
 - No request-related @Context entity may be injected as a field of a Singleton resource.

How do singletons obtain request context?

Injecting Parameters

- JAX-RS can perform dependency injection at two points in the lifecycle of resources:
 - At object initialization.
 - Annotated fields of the resource class can be injected – but field values cannot be request-specific, since there is no request, yet.
 - Every time a resource method is invoked.
 - Annotated parameters to the method can be injected – and parameter values include request-specific entities, since the method invocation corresponds to a client request.

Singletons and Request Context

```
1  @Singleton
2  @Path("/singletonSecureAirports")
3  public class SingletonSecureAirportRM {
4      @POST
5      @Path("/add")
6      public String addAirport(
7          @FormParam("code") String code,
8          @FormParam("name") String name,
9          @Context SecurityContext secContext) {
10         Airport newAirport = null;
11         try {
12             newAirport = dao.add( null, code, name );
13             webContext.log("add:"+secContext.getUserPrincipal());
14         }
15         catch( Exception ex ) {}
16         return (newAirport != null) ? "ok" : "fail";
17     }
18     private AirportDAO dao = new AirportDAO();
19     @Context ServletContext webContext;
```

Injectable Values

Scope	Type
Request	HttpServletRequest, HttpHeaders, HttpServletResponse
Global	SecurityContext, UriInfo, Request, Providers, ServletConfig, ServletContext



Developing Web Services with Java™ Technology

Module 12

Trade-Offs Between JAX-WS and JAX-RS Web Services

Objectives

On completion of this module, you should be able to:

- Understand the trade-offs involved in the choice to implement a web service using either JAX-WS or JAX-RS technology.

Dealing with Web Services

Dave Podnar's 5 Stages of Dealing with Web Services:

Denial It's Simple Object Access Protocol, right?

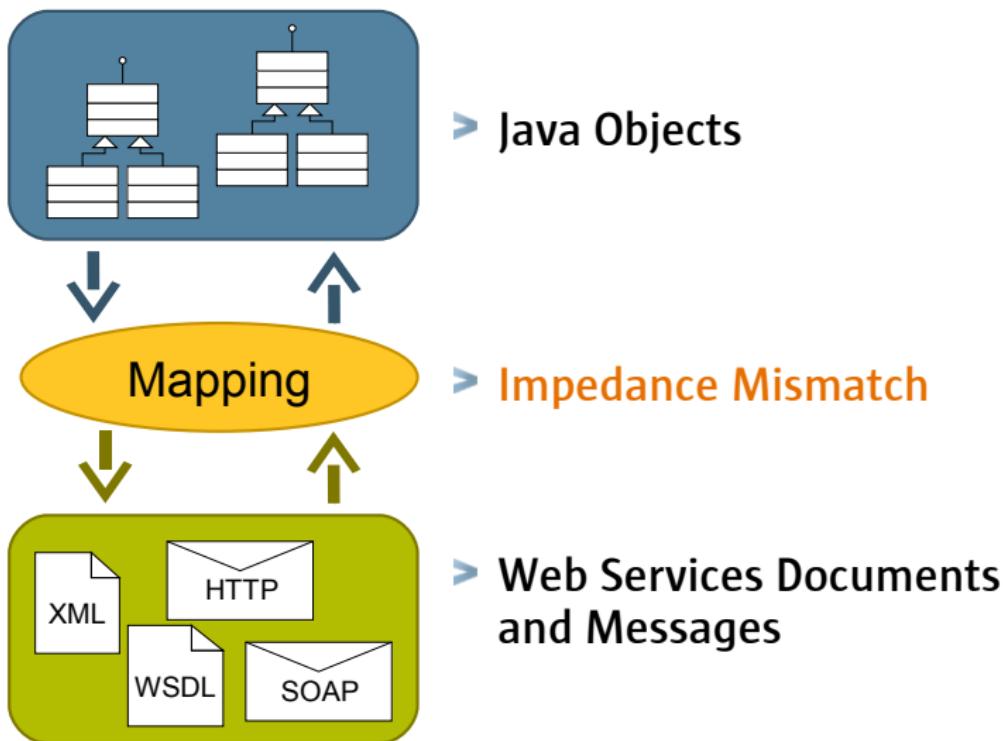
Over Involvement OK, I will read the SOAP, WSDL, WS-I BP, JAX-WS, SAAJ, JAXB, ... specs. Next, I will check the Wiki and finally follow an example showing service and client sides.

Anger I cannot believe they made it **so difficult!**

Guilt Everyone is using Web Services – it must be me, I must be missing something.

Acceptance It is what it is, Web Services are not simple or easy.

Impedance Mismatch



JAXB

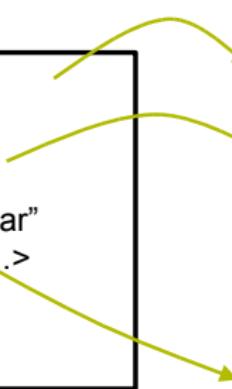
XML Schema

```
<complexType  
name="Foo">  
  <sequence>  
    <element name="bar"  
      type="FooBar" ...>  
    ...  
  </sequence>  
</complexType>
```

Java Code

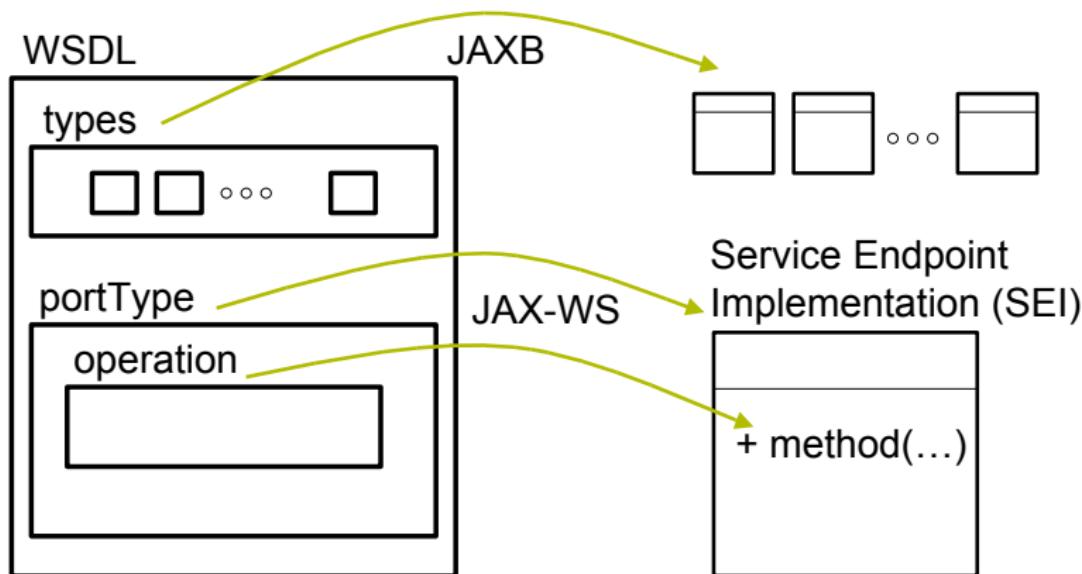
```
Foo  
public FooBar getBar() { ... }  
  
public setBar(FooBar fb) { ... }
```

```
FooBar
```



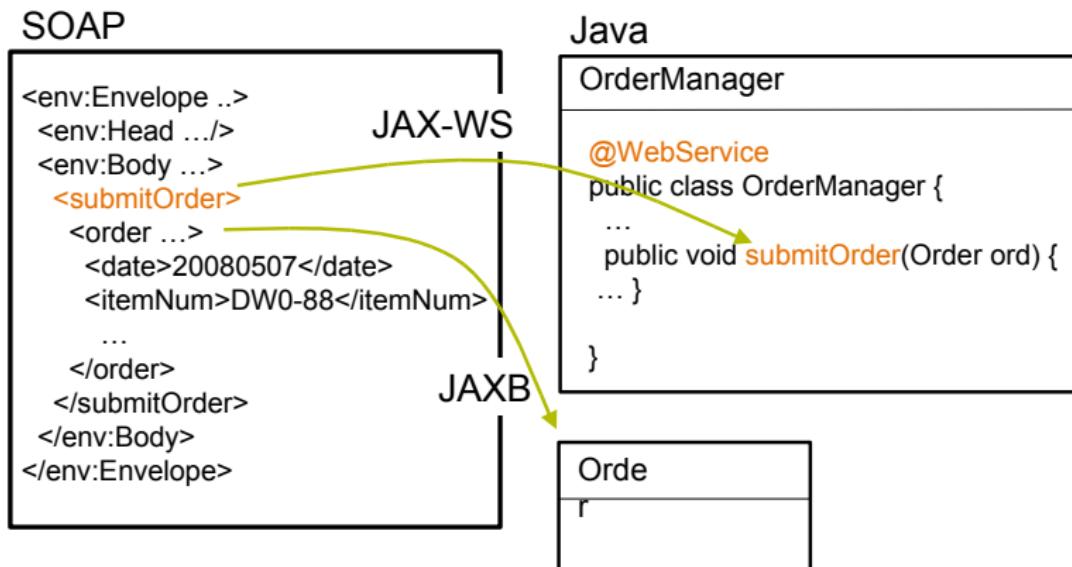
JAX-WS

WSDL to Java

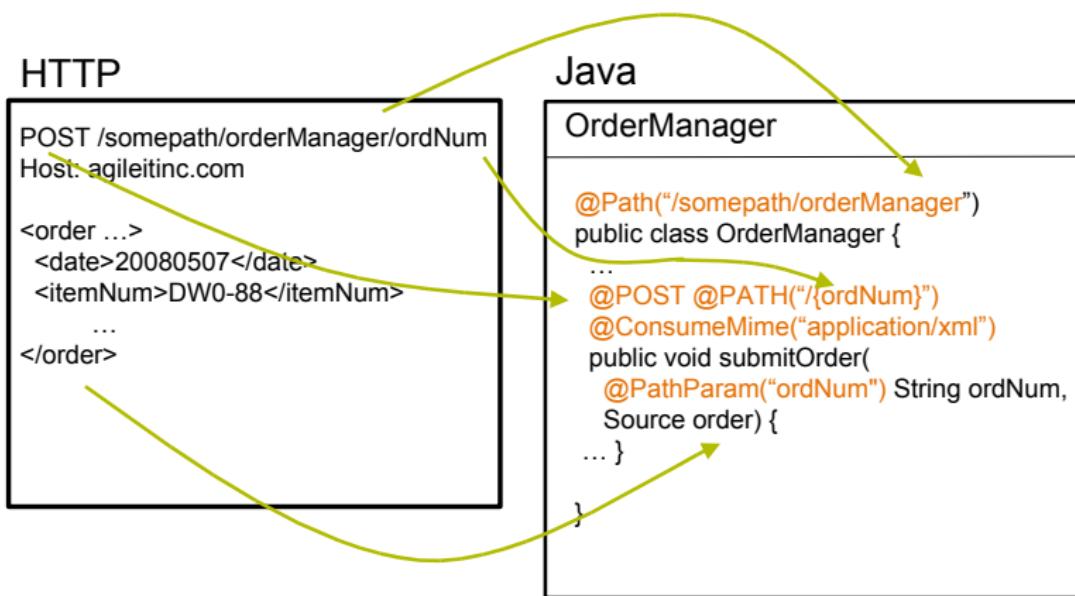


JAX-WS

SOAP to Java



JAX-RS



Approaches to Development



Code First



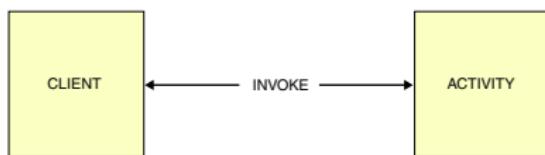
Contract First



Meet in the Middle

Activities vs. Resources

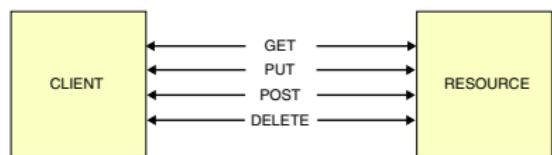
Soap



Soap Services Model

Activities

REST



REST Services Model

Data

Bank Transactions

Lyrics database

Enterprise vs. Web 2.0

Soap

- Main focus:
 - > SOA
 - > EAI
- Supports different protocol bindings
- WS-* stack supports enterprise features

REST

- Main focus:
 - > Web 2.0
- Simple usage, ad-hoc
- Light-weight
- “Per API” tool support

Generality vs. Simplicity

Soap

- Idea:
 - > Have a spec for every possible use case
 - > Cover everything (and introduce extensibility to cover everything else)
- Web Services as the silver bullet of distributed computing

REST

- Idea:
 - > Keep the “standards” as simple and high-level as possible
 - > Let engineers figure out the details on demand
- Web services as simple Web integration technology



**Developing Web Services with Java™
Technology**

Module 13

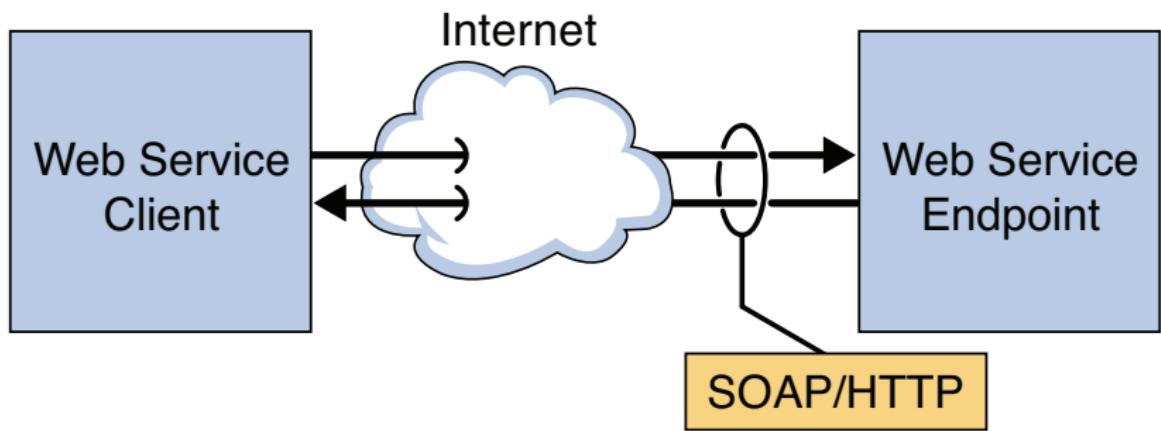
Web Services Design Patterns

Objectives

On completion of this module, you should be able to:

- Describe web services-based design patterns
- Describe web services-based deployment patterns

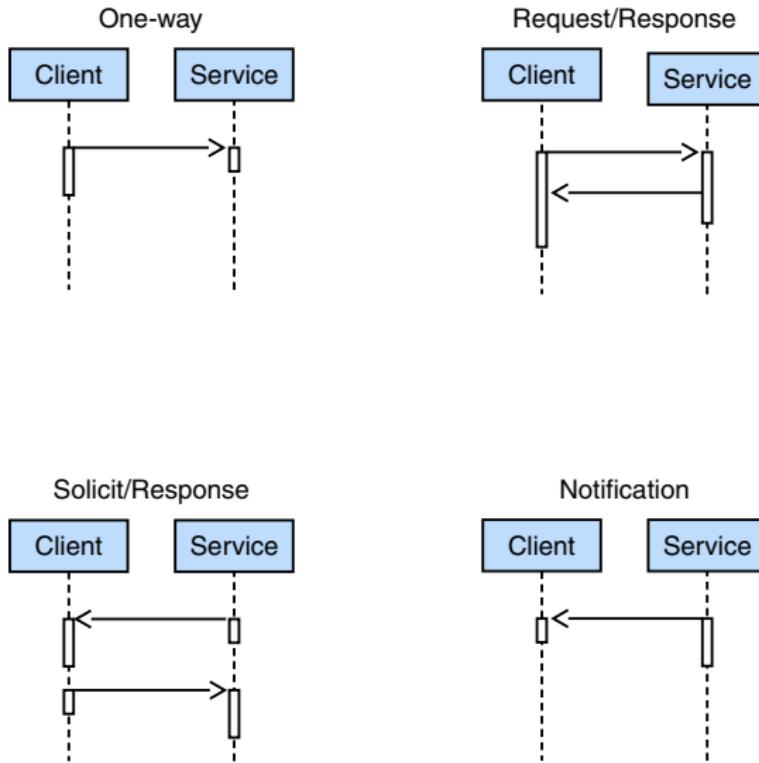
Design Patterns and Web Services



Web Services Design Patterns

- “PAOS” Interactions
 - Asynchronous Interaction
 - JMS Bridge

Web Services Interactions



“PAOS” Interactions

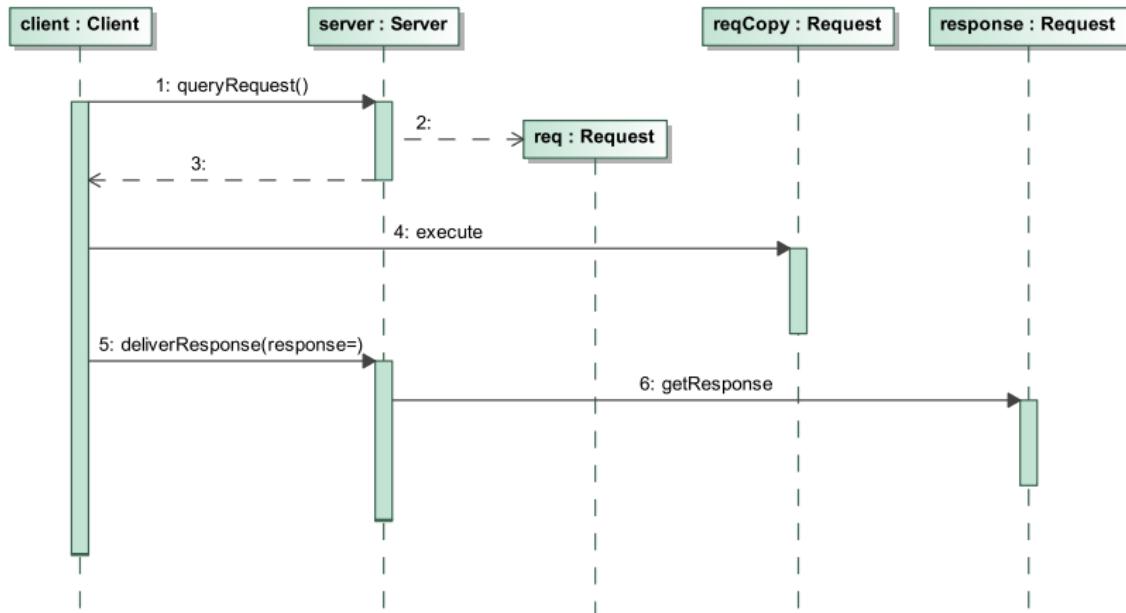
Problem Web services interactions may need to support server-initiated interaction with the client.

Forces The implementation of this service interaction needs to be portable.

Solution A server-initiated interaction can be captured by “encapsulating” it within a larger conversation between client and server – which is always client-initiated.

“PAOS” Interactions

Embedding a “PAOS” Interaction



“PAOS” Interactions

- Advantages:
 - > Server-initiated interactions can be captured within any client-initiated framework.
 - Disadvantages:
 - > Since the client cannot predict when the server might want to initiate an interaction, the client is forced to query the server frequently enough to reduce any delay in processing the server's request.
 - This could lead to significant network overhead.

Asynchronous Interaction

Problem Client requests are too expensive to process, which leads to unacceptable delays for the client while processing takes place.

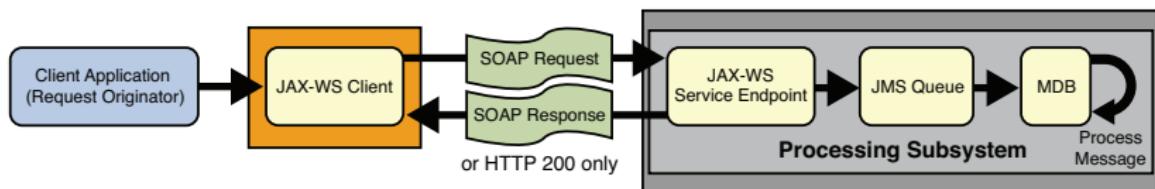
Forces Keep client and server interaction simple.

Solution Clients need not wait for the request to be processed.

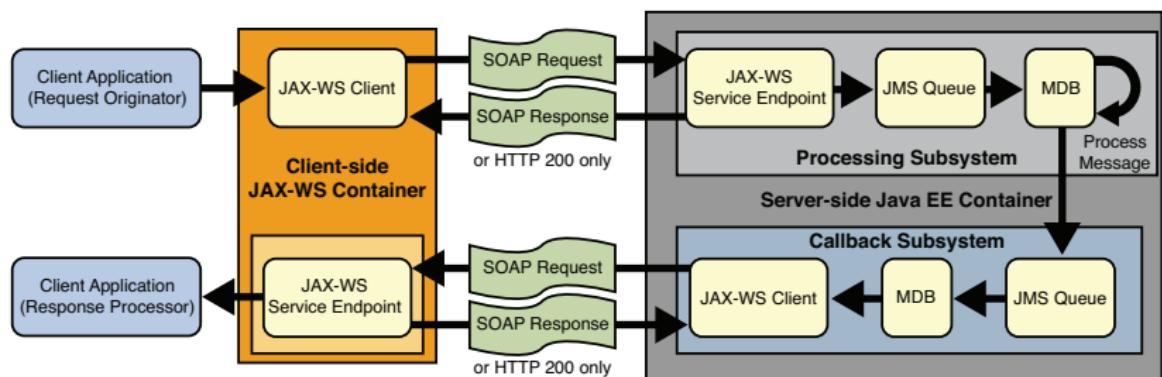
Alternative Scenarios

- Client needs nothing from processing request:
 - > Web service contemplates input with no output
 - Can require WS-ReliableTransport
 - Client needs information that results from processing request:
 - > Must decouple request from response
 - > Must deliver responses to clients
 - > Must match requests to responses

A Simple Strategy

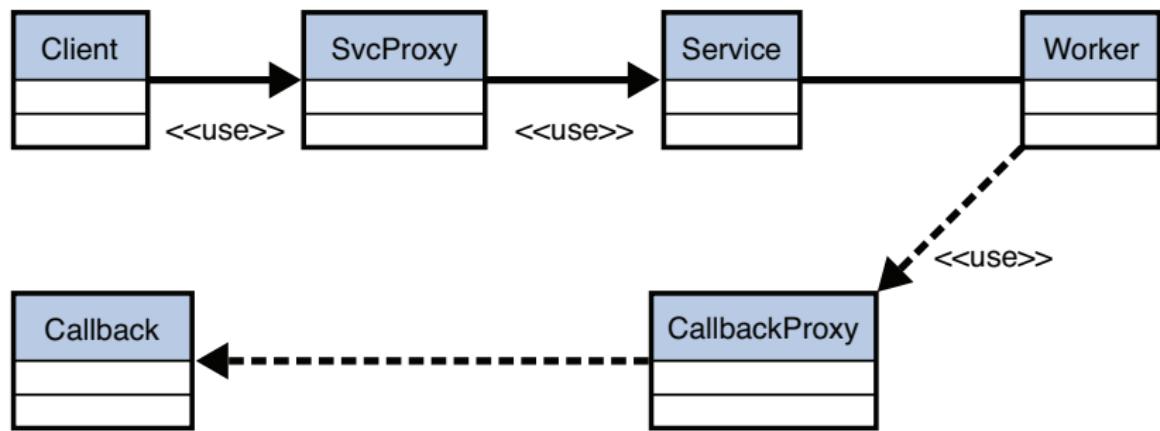


Complex Strategy: Server Push



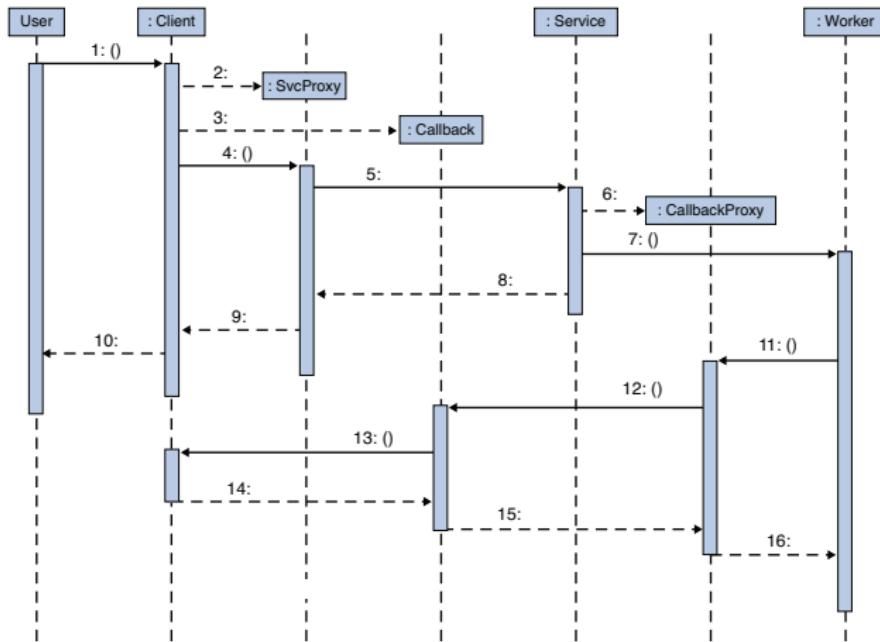
Server Push

Class Diagram



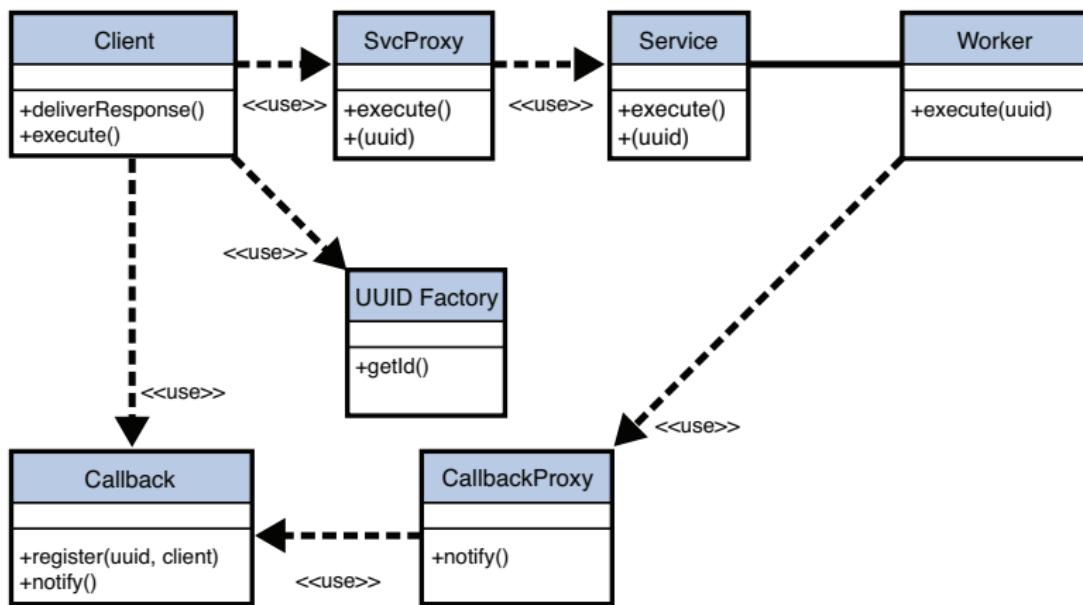
Server Push

Sequence Diagram



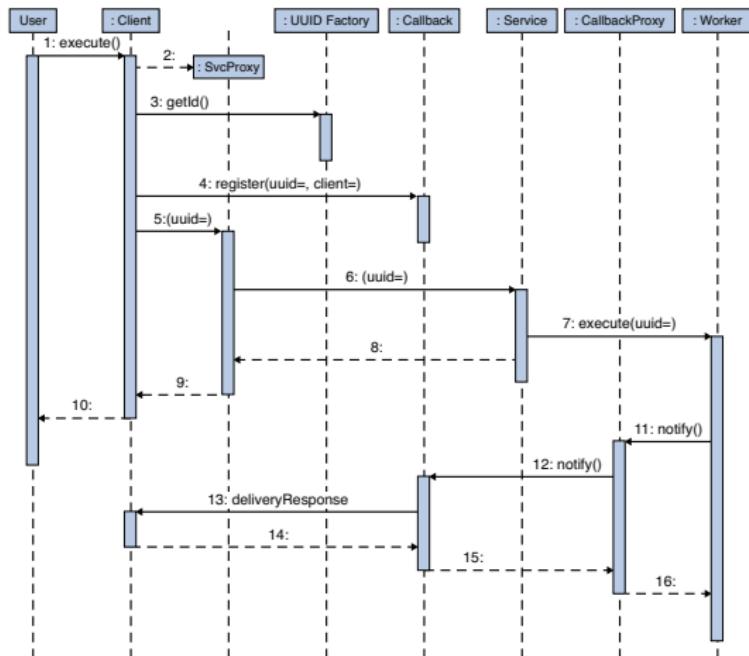
Server Push – Another Approach

Class Diagram

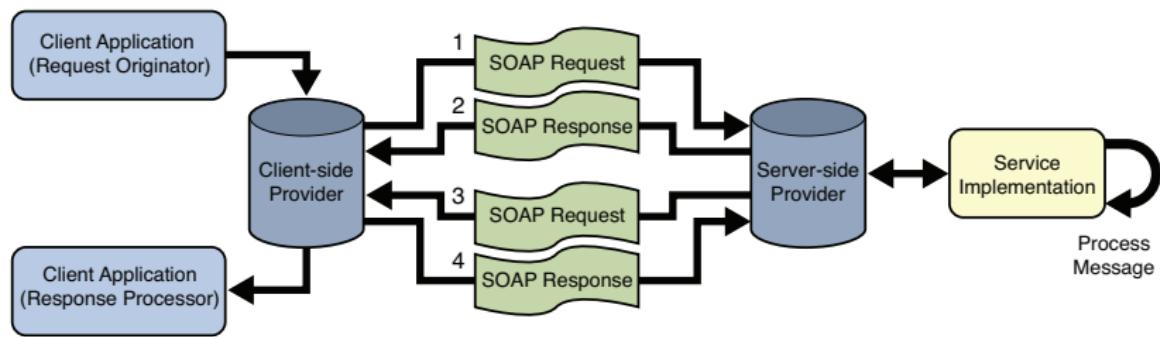


Server Push – Another Approach

Sequence Diagram



Complex Strategies: JAX-WS Provider API



Dispatch<T> and Provider<T>

```
1 public interface Dispatch<T> {  
2     //synchronous request-response  
3     T invoke(T msg);  
4     //async request-response  
5     Response<T> invokeAsync(T msg);  
6     Future<?> invokeAsync(T msg, AsyncHandler<T> h);  
7     // one-way  
8     void invokeOneWay(T msg);  
9 }
```

```
1 public interface Provider<T> {  
2     T invoke(T msg, Map<String, Object> context);  
3 }
```

Dispatch Client

Setup

```
1  public class MessagingAPIClient {  
2      public static void main(String[] args) throws Exception {  
3          QName serviceName =  
4              new QName("urn:examples", "Examples");  
5          QName portName = new QName("urn:examples", "Hello");  
6          Service service = Service.create(serviceName);  
7          String url = "http://127.0.0.1:8081/MessagingAPI";  
8          service.addPort(portName, SOAPBinding.SOAP11HTTP_BINDING,  
9                             url);  
10         Class msgClass = MessagingAPIMessage.class;  
11         JAXBContext jaxbCtx =  
12             JAXBContext.newInstance(msgClass);  
13         Dispatch<Object> port =  
14             service.createDispatch(portName, jaxbCtx, Mode.PAYLOAD);
```

Dispatch Client

Asynchronous Call

```
1  MessagingAPIMessage request =
2      new MessagingAPIMessage("sayHello", "Tracy");
3  AsyncHandler<Object> responseHandler =
4      new AsyncHandler<Object>() {
5          public void handleResponse(Response<Object> resp) {
6              try {
7                  MessagingAPIMessage result =
8                      (MessagingAPIMessage) resp.get();
9                  System.out.println(result.getResult());
10             }
11             catch( Exception e )
12                 {}
13         }
14     };
15 port.invokeAsync( request, responseHandler );
```

Provider Server

Setup

```
1 @ServiceMode (Mode.PAYLOAD)
2 @WebServiceProvider (portName="Hello", serviceName="Examples",
3                      targetNamespace="urn:examples")
4 public class MessagingAPIServer implements Provider<Source> {
5     MessagingAPIServer() throws JAXBException {
6         Class msgClass = MessagingAPIMessage.class;
7         jaxbContext = JAXBContext.newInstance( msgClass );
8     }
9     // ...
10    private JAXBContext jaxbContext;
11    public static void main(String[] args) throws Exception {
12        String url = "http://127.0.0.1:8081/MessagingAPI";
13        MessagingAPIServer server = new MessagingAPIServer();
14        Endpoint endpoint = Endpoint.publish( url, server );
15    }
16 }
```

Provider Server

Processing

```
1  public Source invoke(Source payload) {  
2      try {  
3          Unmarshaller u = jaxbContext.createUnmarshaller();  
4          MessagingAPIMessage message =  
5              (MessagingAPIMessage) u.unmarshal( payload );  
6          message.setResult("Hello, " + message.getArgument());  
7          return new JAXBSource( jaxbContext, message );  
8      }  
9      catch( Exception ex )  
10     { throw new WebServiceException( ex ); }  
11 }
```

Asynchronous Interaction

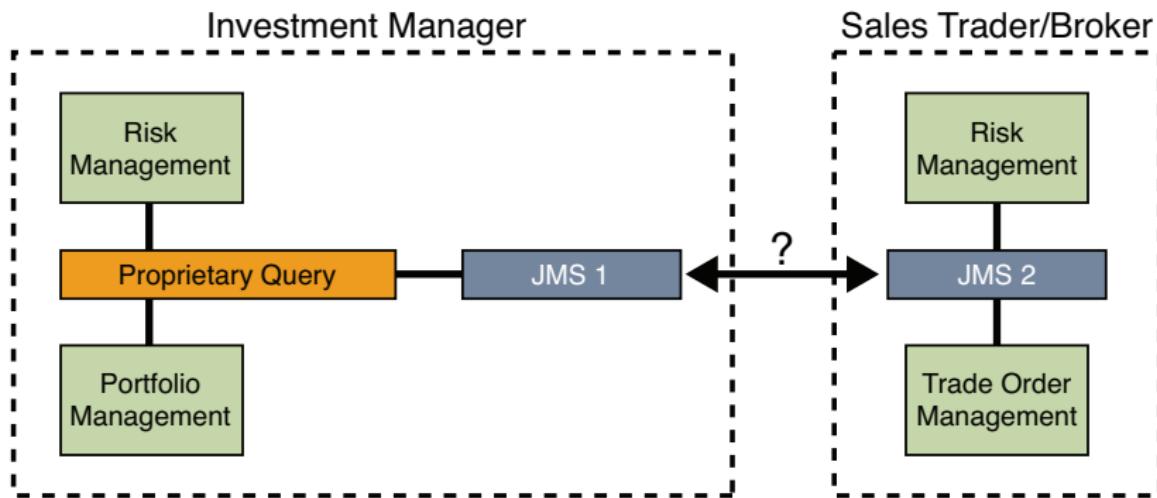
- Advantages:
 - > Decoupling request and response allows clients to continue processing while the server performs expensive computations.
 - > JAX-WS provides support for this pattern explicitly, minimizing programming effort.
- Disadvantages:
 - > Some implementations result in more complex application logic.

JMS Bridge

- Problem** Distributed enterprise applications often span multiple implementations of the Java EE platform
- Forces** Support multiple JMS implementations transparently
- Solution** Use web services as an implementation-neutral bridge between different JMS implementations

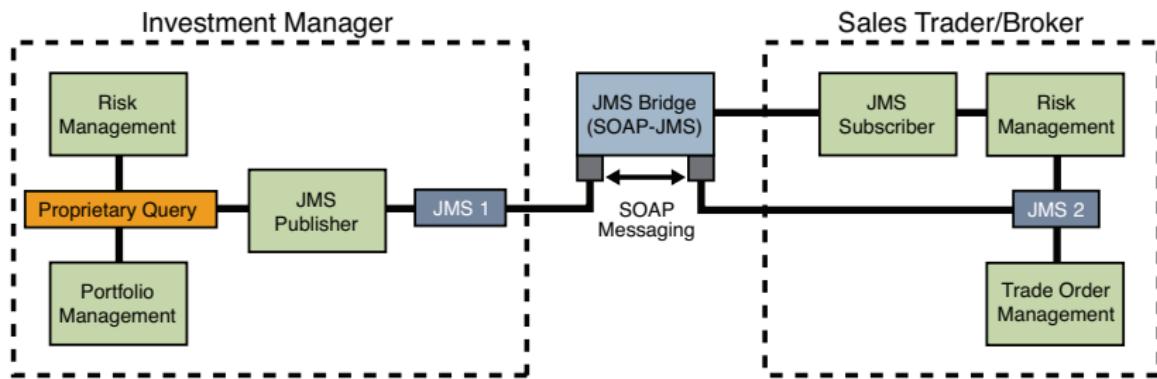
JMS Bridge

Sample Scenario



JMS Bridge

Sample Solution



JMS Bridge

- Advantages:
 - > JMS bridge is vendor/JMS-independent, so it can guarantee messaging between any two JMS implementations.
- Disadvantages:
 - > JMS bridge incurs overhead corresponding to the XML encoding, transmission, and XML decoding associated with the relaying of the JMS message

Web Services Deployment Patterns

- HTTP Load Balancer
- Container Cluster

HTTP Load Balancer

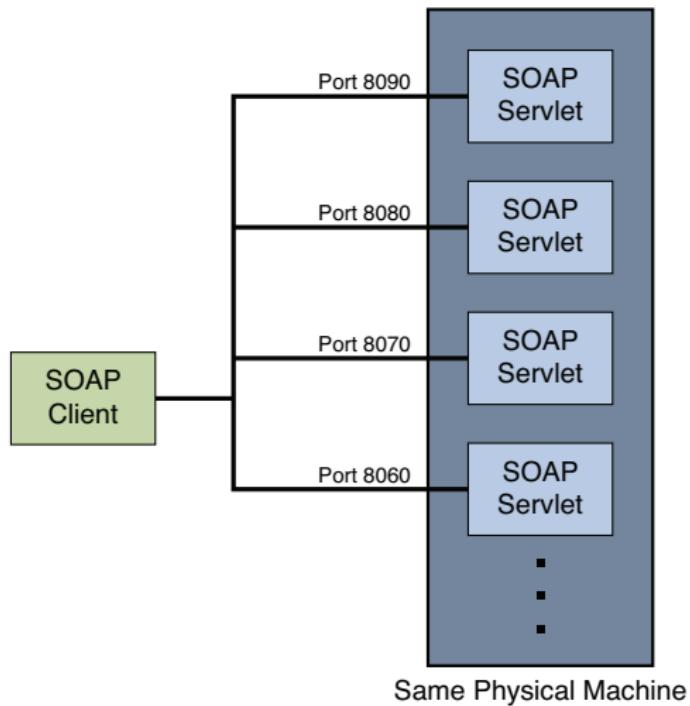
Problem Not enough resources to handle client load

Forces Scale up resources transparently

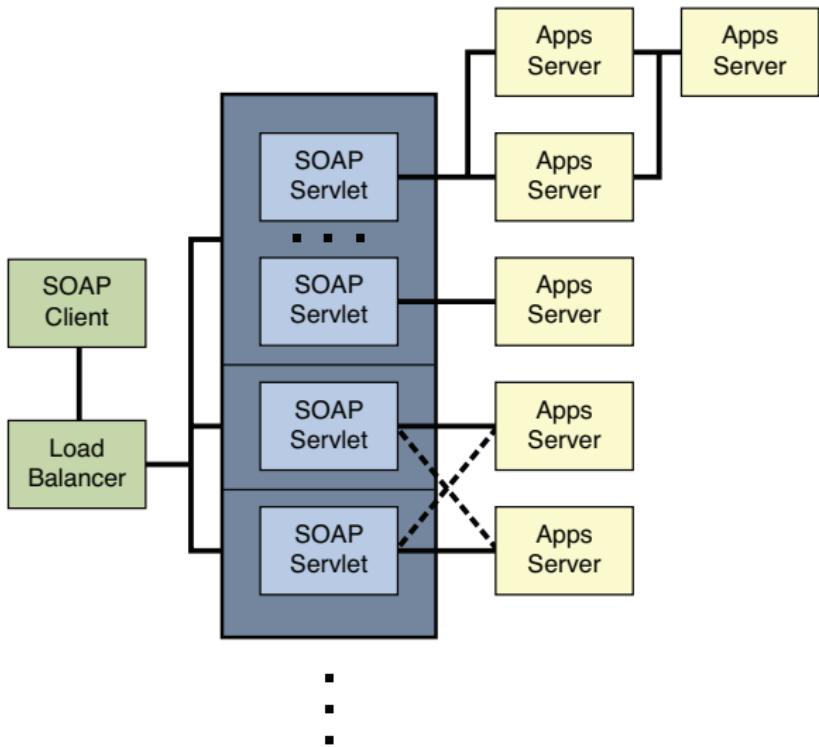
Solution

- Application partitioning
- Load balancing

Application Partitioning



Load Balancing



HTTP Load Balancer

- Advantages:
 - > Increases scalability by increasing resource availability
- Disadvantages:
 - > Increased manageability
 - > Poor choice of partitioning strategy would limit scalability benefit by mis-allocating resources

Container Cluster

Problem Ensure availability of application even during system failures

Forces Improve availability transparently

Solution Failover strategies in a clustered deployment

Container Cluster

- Advantages:
 - > Improved scalability via failover strategies within cluster
- Disadvantages:
 - > Increased manageability costs for cluster
 - > Potential increased overhead due to session synchronization over cluster



Developing Web Services with Java™
Technology

Module 14

Best Practices and Design Patterns for JAX-WS

Objectives

On completion of this module, you should be able to:

- Describe JAX-WS-specific design patterns
 - Recognize and apply best practices associated with implementing web services using JAX-WS

Web Services Design Patterns

- Web Service Cache
- Web Service Broker
- Web Service Logger

Web Service Cache

Problem Each client request results in overhead:

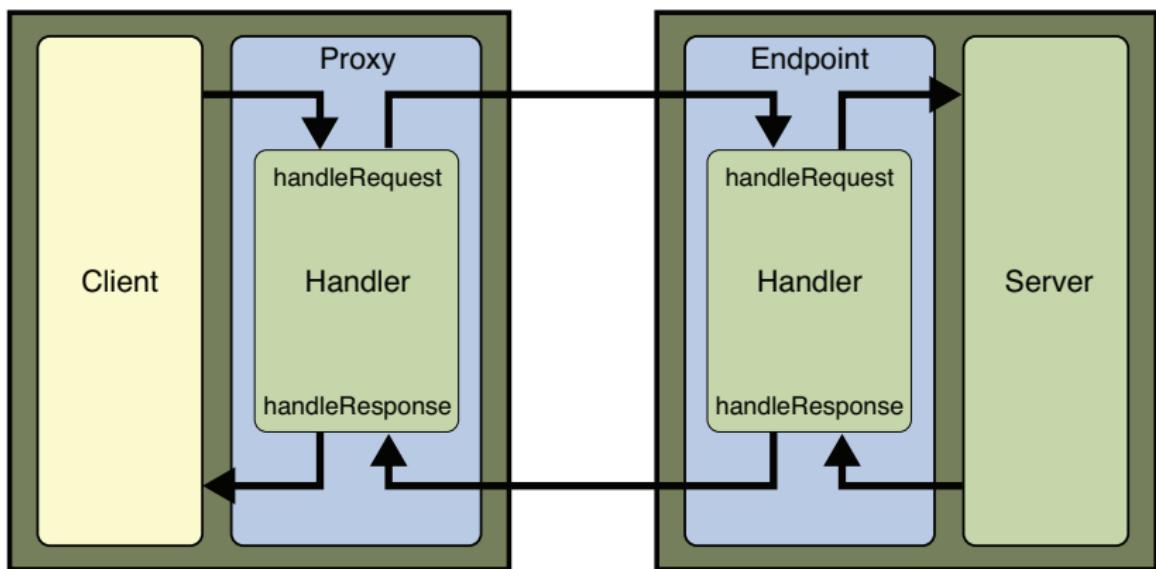
- Communication client <-> service
- Computation

Forces Minimize overhead associated with each call

Solution Cache answers previously computed

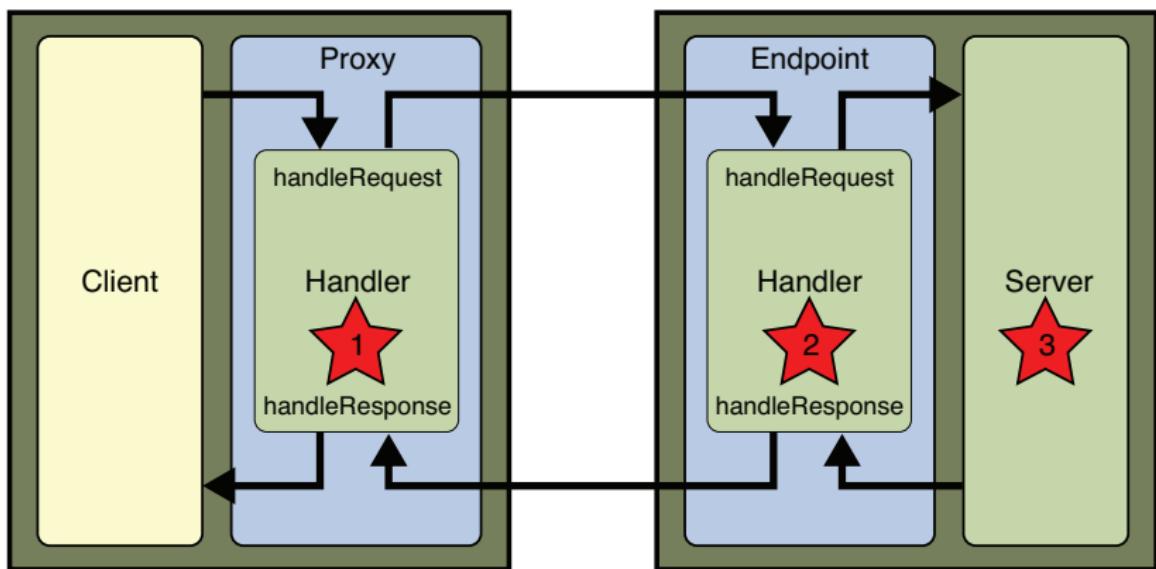
Caching Answers in Web Services

- Typical structure of a web service call:



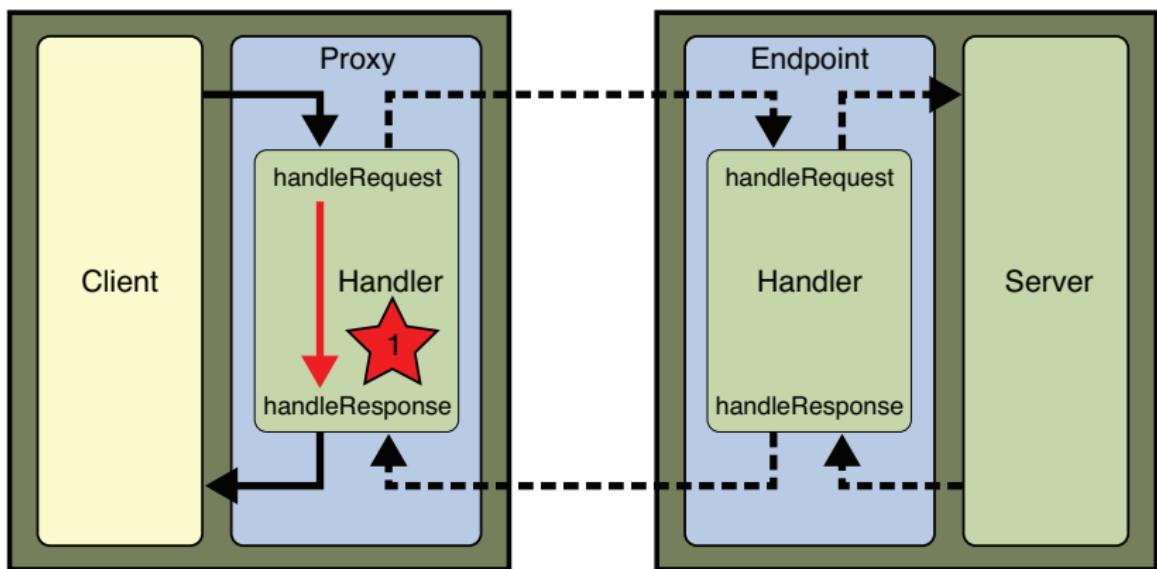
Caching Answers in Web Services

- Typical structure of a web service call:



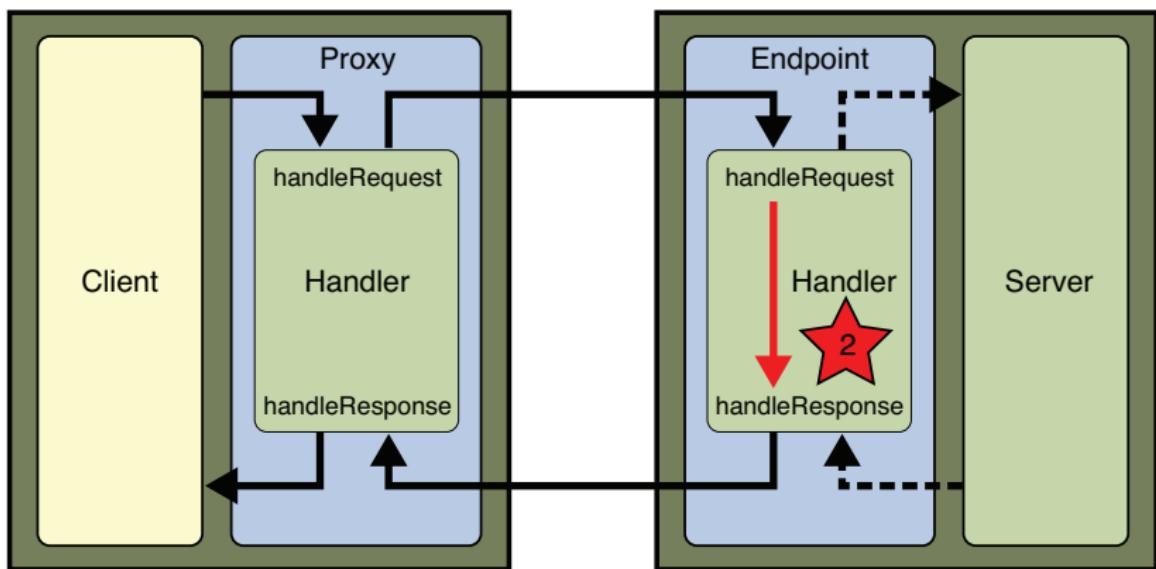
Caching Answers in Web Services

- Application-independent, client-side:



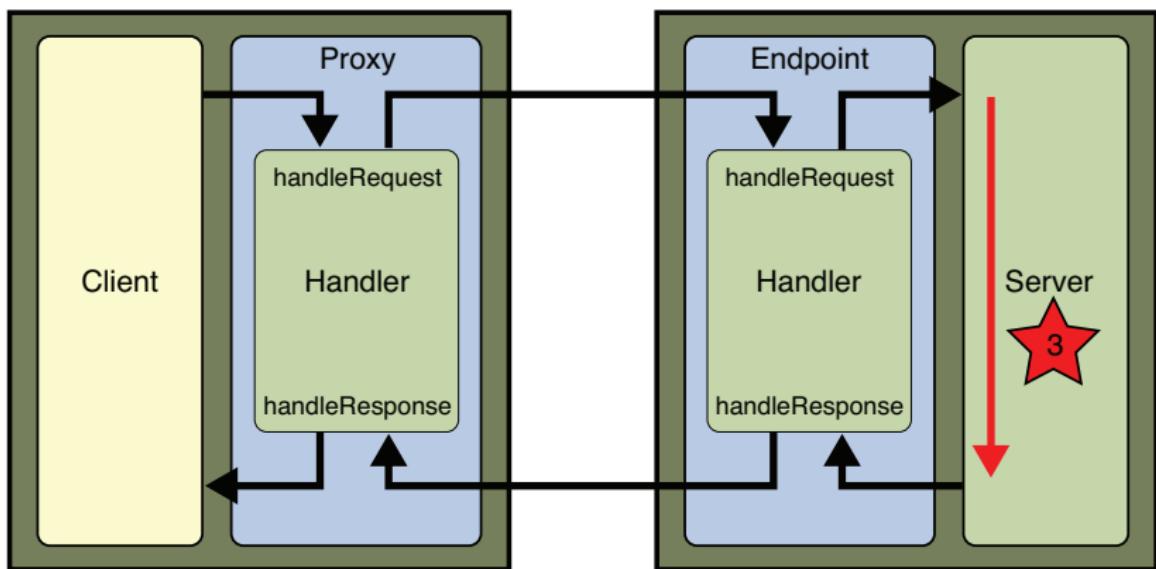
Caching Answers in Web Services

- Application-independent, server-side:



Caching Answers in Web Services

- Application-specific, server-side:



Web Services Cache

- Advantages:
 - > Minimizes overhead associated with operations:
 - > Network overhead
 - > Processing overhead
- Disadvantages:
 - > Realizing when to invalidate/refresh the cache could be complex
 - > Realizing when to cache could be complex
 - > Increases memory footprint to hold cache

Web Service Broker

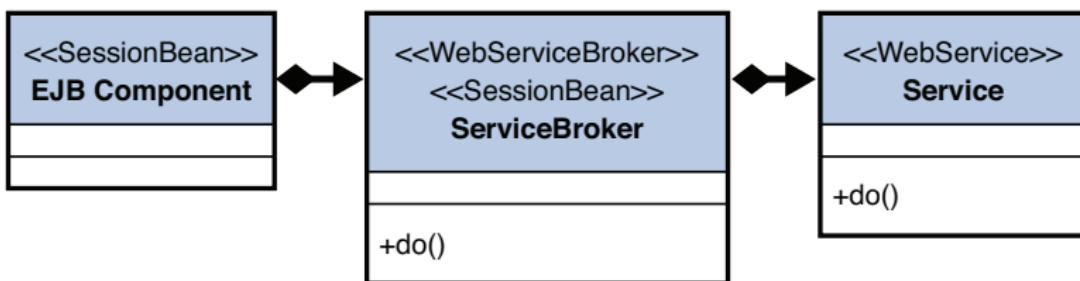
Problem Aggregate finer-grained (web) services

- Forces**
- Address integration “bumps”
 - Transaction semantics compared to WS
 - loose coupling

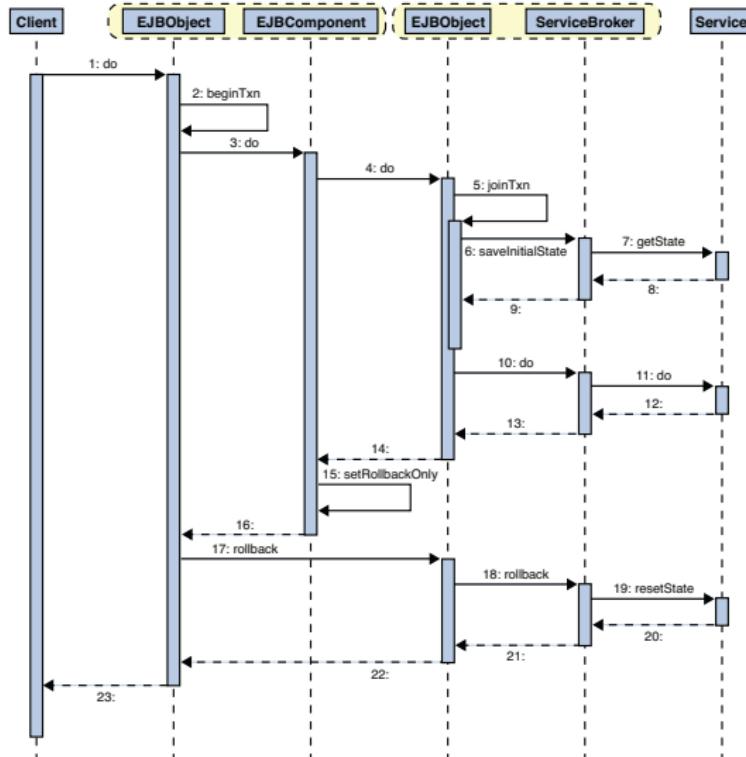
- Solution**
- Use Façade pattern to aggregate services
 - Implement application-level *compensating transactions*

Web Service Broker

Classes



Web Service Broker



Web Service Broker

- Advantages:
 - > Simplifies client design
- Disadvantages:
 - > Compensating transactions can be complex to implement

Web Service Logger

Problem Log operations as they are performed

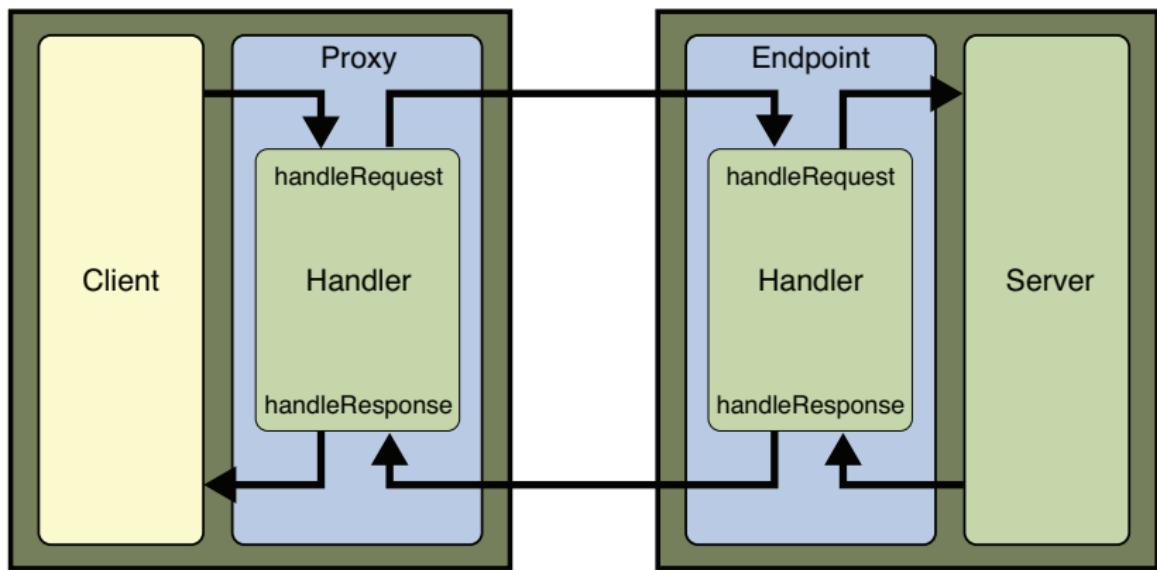
Forces Improve maintainability by making logging transparent to application logic

Solution

- Use Decorator pattern
- Use Chain of Responsibility (Pipeline) pattern

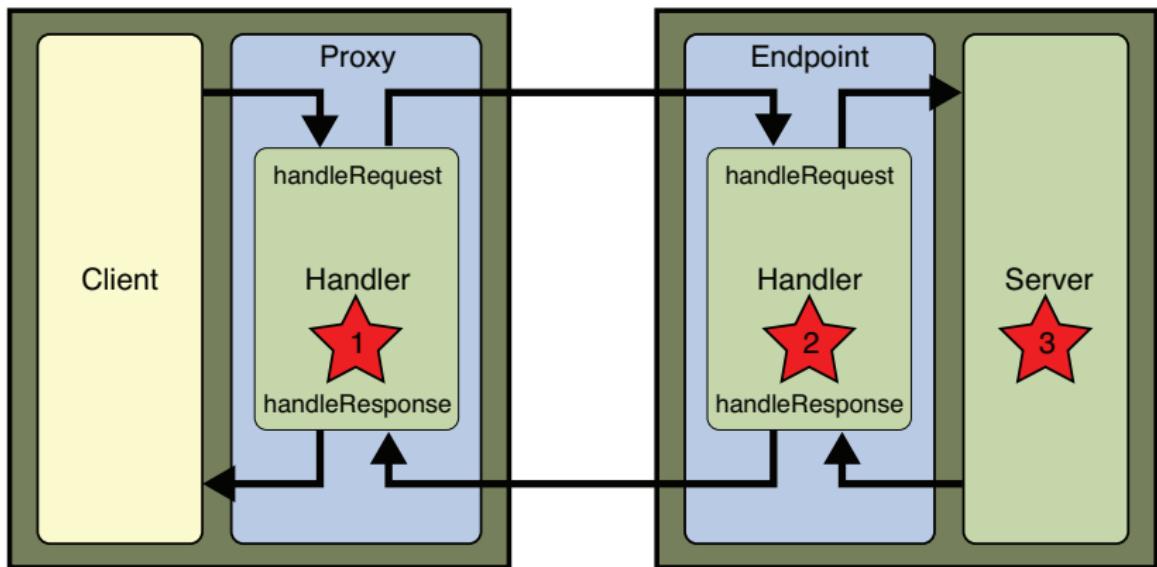
Logging Web Services Calls

- Typical structure of a web service call:



Logging Web Services Calls

- Opportunities for logging:



Sample Service

- A web service that uses handlers

```
1 @WebService  
2 @HandlerChain(file="serviceChain.xml")  
3 public class Service{  
4     ...  
5 }
```

Sample Service Configuration

```
1 <!-- define logging handler config -->
2 <handler name="logger"
3     type="examples.Handlers.LogHandler">
4     <parameter name="filename" value="log.txt"/>
5 </handler>
6 <!-- define the service -->
7 <service name="GreeterService" provider="java:RPC">
8     <requestFlow>
9         <handler type="logger"/>
10    </requestFlow>
11    <parameter name="className"
12        value="examples.Handlers.GreeterService"/>
13    <parameter name="allowedMethods" value="*"/>
14 </service>
```

Web Service Logger

- Advantages:
 - > Decoupling logging from other application-level concerns leads to more maintainable designs
 - > Strategy can be used to introduce other "cross-cutting" concerns
- Disadvantages:
 - > Additional configuration not evident in code is needed to incorporate logger into application



Developing Web Services with Java™
Technology

Module 15

Best Practices and Design Patterns for JAX-RS

Objectives

On completion of this module, you should be able to:

- recognize and apply best practices associated with implementing web services using JAX-RS.

RESTful Best Practices

- Obey standard HTTP action semantics.
- Leverage standard HTTP status codes.
- URI Best Practices:
 - > URI opacity.
 - > Query string extensibility.
- Support multiple representations.

HTTP Action Semantics

GET readonly and idempotent. Never changes the state of the resource

PUT an idempotent insert or update of a resource. Idempotent because it is repeatable without side effects.

DELETE resource removal and idempotent.

POST non-idempotent, “anything goes” operation

HTTP Action Semantics

Examples

URI	Action	Intent
/airports	GET	index
/airports	POST	create
/airports/{code}	GET	show
/airports/{code}	PUT	update
/airports/{code}	DELETE	destroy

<http://www.slideshare.net/calamitas/restful-best-practices>

Creating New Entities

Using POST

- Creating a new entity could be implemented by submitting a POST request:

POST /airports

with the new airport in the body of the request.

- What's wrong with this approach?

Creating New Entities

POST or PUT

- Dangers when using POST:
 - > If there is a server-side failure during processing, the client may not realize that the entity was not created.
 - > If the client submits the request twice, we could improperly create two entities.
 - > The problem is that POST is *not* idempotent
- What happens if we just use PUT instead of POST ?
 - > After all, PUT *is* idempotent...

Creating New Entities

Using PUT

- When using PUT, the application must be able to distinguish between:
 - > Requests that are meant to create *distinct* entities.
 - > Redundant requests to create the *same* entity.
- Clients can assign an id to each request submitted:

```
PUT /airports/clientGeneratedId
```

where *clientGeneratedId* can be used to disambiguate requests.

HTTP Status Codes

- 1xx – Informational
 - > Request received, continuing process.
- 2xx – Success
 - > Action successfully received, understood, and accepted.
- 3xx – Redirection
 - > Client must take additional action to complete the request.
- 4xx – Client Error
 - > Request contains bad syntax or cannot be fulfilled.
- 5xx – Server Error
 - > Server failed to fulfil an apparently valid request.

HTTP Status Codes

2xx Status Codes

Code	Meaning	Description
200	OK	Successful HTTP request.
201	Created	New resource created.
202	Accepted	Accepted for processing, but not yet completed.
204	No Content	Successfully processed, but no return content.
206	Partial Content	Only part of the resource returned.

http://en.wikipedia.org/wiki/List_of_HTTP_status_codes

HTTP Status Codes

3xx Status Codes

Code	Meaning	Description
300	Multiple Choices	Multiple options for the resource that the client may follow.
301	Moved Permanently	This and all future requests should be directed to the given URI.
303	See Other	Response can be found at URI using GET method.
304	Not Modified	Resource has not been modified since last requested.

HTTP Status Codes

4xx Status Codes

Code	Meaning	Description
400	Bad Request	Request contains bad syntax or cannot be fulfilled.
401	Unauthorized	Request refused, when authentication is possible but has failed or not yet been provided.
403	Forbidden	Request was legal, but the server refuses to respond to it.
404	Not Found	Resource could not be found but may be available again in the future.

HTTP Status Codes

4xx Status Codes

Code	Meaning	Description
405	Method Not Allowed	Request made using method not supported by that resource.
406	Not Acceptable	Resource can only generate content not acceptable given Accept headers sent in.
409	Conflict	Request could not be processed due to conflict in the request.

HTTP Status Codes

4xx Status Codes

Code	Meaning	Description
410	Gone	Resource no longer available and will not be available again.
412	Precondition Failed	Server does not meet precondition put on the request.
415	Unsupported Media Type	Request did not specify any media types the resource supports.
417	Expectation Failed	Server cannot meet requirement of Expect header field.
418	I'm a teapot	Response entity "MAY be short and stout".

URI Best Practices

- Use nouns for resources, rather than verbs.
 - > Resources are “things”, not “actions”.
- Minimize the use of query strings.
- Slashes – / – in URIs denote parent/child or whole/part relationships.
- Aim for “progressive disclosure”.

<http://www.xfront.com/sld059.htm>

URI Best Practices

- URI opacity
 - Users should not derive metadata from the URI path itself.
 - > The query string and fragment can have special meaning.
- Query string extensibility
 - A service provider should ignore any query parameters it does not understand during processing; when consuming other services, it should pass all ignored parameters along.
 - > This practice allows new functionality to be added without breaking existing services.

Representations

- Resources should support multiple representations
- Choice of representation through negotiation
 - > Server-driven negotiation.
 - service provider determines representation
 - can use the information provided in HTTP headers.
- Client-driven negotiation.
 - ① client initiates request
 - ② server returns a list of available representations
 - ③ client selects representation and sends a second request to the server.
- URI-specified representation

Request Validation

- Validate client requests:
 - > Perform data validation on resources received during PUT and POST.
 - > Use HTTP error codes to define and transfer exception information.
 - > Handle those exceptions on the client side.

<http://blog.feedly.com/2009/05/06/best-practices-for-building-json-rest-web-services/>



Developing Web Services with Java™ Technology

Module 16

References

References



References

The JavaOne 2002 logo is located at the top right of the slide. It consists of the word "JavaOne" in a large, bold, sans-serif font, with "2002" in smaller letters below it. A small "SUN" logo is positioned to the left of the main text.

A screenshot of a presentation slide from JavaOne. The title 'JavaOne' is displayed prominently in the top right corner. Below it, the slide content reads 'GlassFish Project Web Services Stack "Metro" : Easy to Use, Robust, and High-Performance'. At the bottom left, the names Jitendra Kotamraju, Marek Potoclar, and Sun Microsystems are listed, along with the number 154434. The bottom right corner features the Sun Microsystems logo.

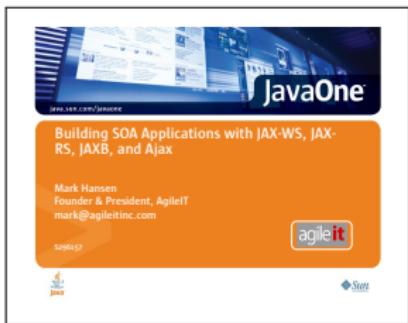
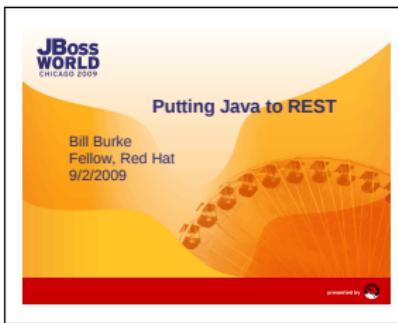
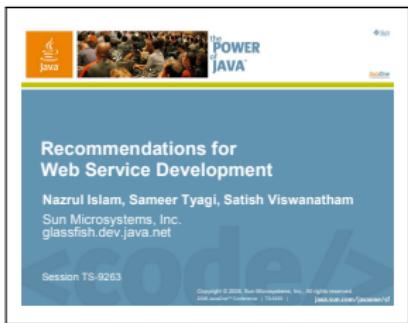
A slide from a JavaOne 2007 presentation. The top half features a Java logo (a coffee cup) and a photo of a crowded conference room. The bottom half contains the title and speaker information.

A presentation slide from JavaOne 2006. The title is "The Java API for XML Web Services (JAX-WS) 2.0". The authors are Roberto Chinnici and Rajiv Mordani. It features a background image of a JavaOne conference audience. The Java logo is in the top left corner, and the Sun Microsystems logo is in the top right. The bottom right corner has the JavaOne 2006 logo.

References



References





Developing Web Services with Java™ Technology

Appendix A

XML Schema

Objectives

On completion of this module, you should be able to:

- Create a basic XML Schema
- Structure XML Schemas
- Use data types in an XML Schema
- Use more advanced features in XML Schemas
 - > Use mixed content
 - > Use empty elements
 - > Use annotations
 - > Define namespaces
 - > Include other XML schema documents

Building a Schema

- Creating a basic schema
- Linking to a schema
- Determining the number of occurrences of an element
- Adding subelements
- Using the choice element
- Declaring an attribute
- Deriving a new element

Creating a Basic Schema

```
①<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>
 ②<xs:element name="people">
    ③<xs:complexType>
        ④<xs:sequence>
            <xs:element name="person" type="xs:string" />
        ⑤</xs:sequence> ⑥
    </xs:complexType>
</xs:element>
</xs:schema>
```

Schema Syntax

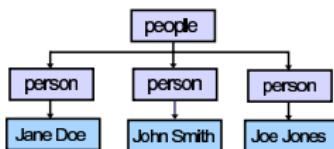
- ① Schema element and namespace.
 - ② The element tag declares the people element.
 - ③ The complexType tag defines the people element.
 - ④ The sequence tag defines the order of elements it contains.
 - ⑤ The element tag declares the person element.
 - ⑥ The type attribute declares this element as a string type.

Linking to a Schema

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- Pet club people file -->
❶<people xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
❷  xsi:noNamespaceSchemaLocation="file:07_04.xsd">
    <person>Jane Doe</person>
</people>
```

- ① Declaring the xsi schema instance namespace.
 - ② This attribute indicates where to find the schema file.

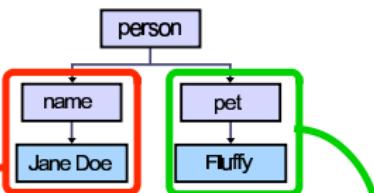
Determining the Number of Items



Schema File

```
<xs:schema xmlns:xs="http://w3.org/2001/XMLSchema">
  <xs:element name="people">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="person" type="xs:string" minOccurs="1" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

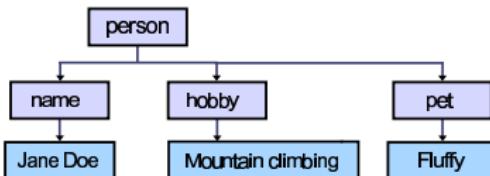
Adding Subelements



Schema File

```
<xs:sequence>
  <xs:element name="person" type="xs:string" minOccurs="1" maxOccurs="unbounded" />
    <s:complexType>
      <s:sequence>
        <xs:element name="name" type="xs:string" />
        <xs:element name="pet" type="xs:string" />
      </s:sequence>
    </s:complexType>
  </xs:element>
</xs:sequence>
```

Using the Choice Element



Schema File

```
<xs:complexType>
  <xs:sequence>
    <xs:element name="person" type="xs:string" minOccurs="1" maxOccurs="unbounded" />
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string" />

        <xs:choice minOccurs="1" maxOccurs="unbounded">
          <xs:element name="pet" type="xs:string" />
          <xs:element name="hobby" type="xs:string" />
        </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:sequence>
```

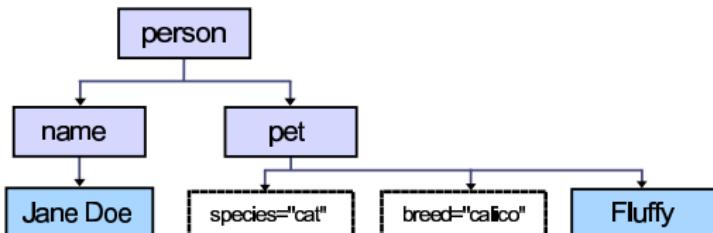
Declaring an Attribute

```
<xs:attribute name="name_here" type="data_type" use="attribute_type"  
    default="default_value" fixed="fixed_value"/>
```

Attribute Syntax

- ➊ Attribute name
- ➋ Attribute data type
- ➌ Attribute is required, optional, or prohibited
- ➍ Default value
- ➎ Fixed value

Deriving a New Element

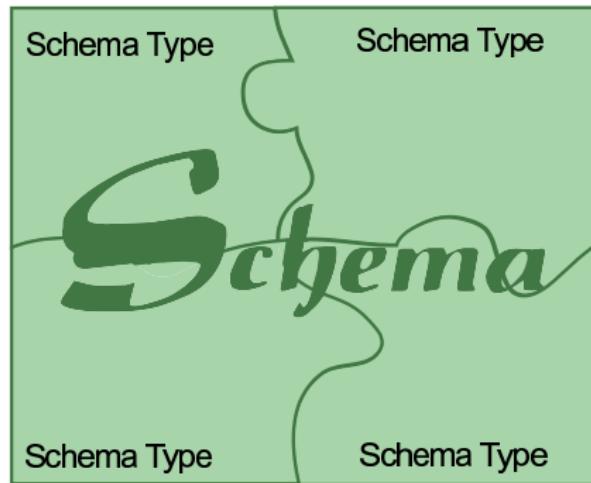


Schema File

```
<xs:element name="pet">
<xs:complexType>
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name="species" type="xs:string" />
      <xs:attribute name="breed" type="xs:string" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
</xs:element>
```

Structuring XML Schemas

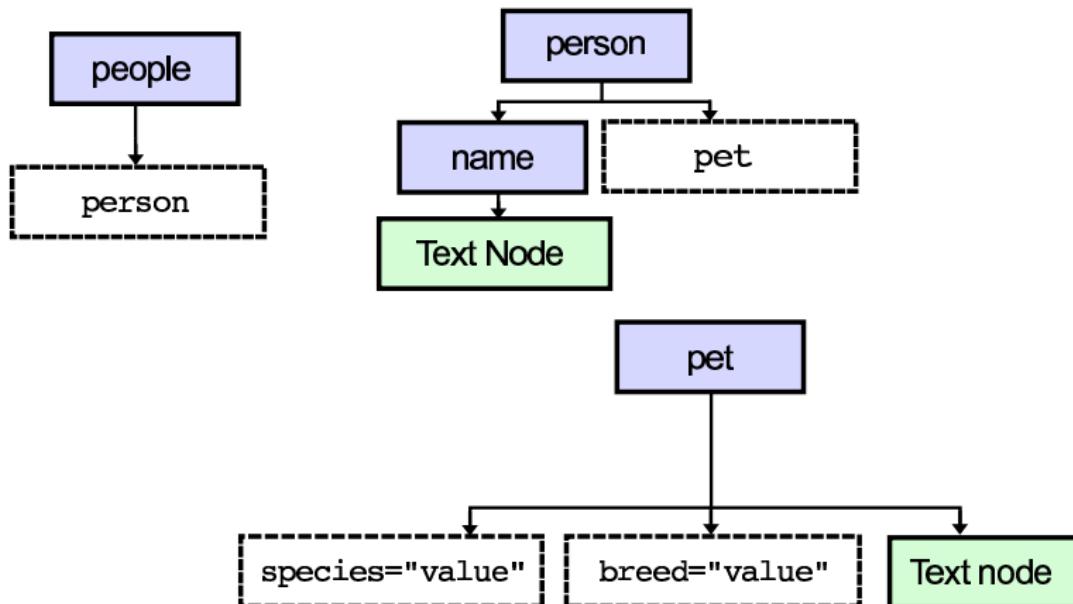
- Defining schema types
 - Applying types to a schema



Defining XML Schema Types

```
<xs:element name="pet">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="species" type="xs:string" />
        <xs:attribute name="breed" type="xs:string" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

Applying Types to a Schema



Using data types in an XML Schema

- Common schema data types
- Defining simple schema types
 - > restricting an integer
 - > using patterns
 - > using enumerations
 - > using dates
 - > restricting strings

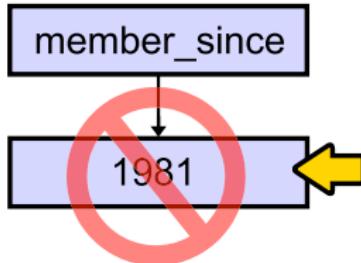
Defining Simple XML Schema Types

```
①<xs:simpleType name="member_sinceType">
②<xs:restriction base="xs:int">
    ③<xs:minInclusive value="1990" />
    <xs:maxInclusive value="2100" />
</xs:restriction>
</xs:simpleType>
```

- ① Define the name of this simpleType.
- ② The restriction element sets the simpleType that will be constrained.
- ③ The constraints to be applied to the simpleType.

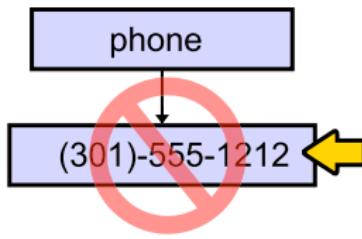
Restricting an Integer

```
<xs:simpleType name="member_sinceType">
  <xs:restriction base="xs:int">
    <xs:minInclusive value="1990" />
    <xs:maxInclusive value="2100" />
  </xs:restriction>
</xs:simpleType>
```



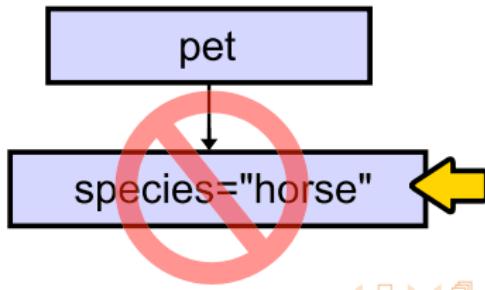
Using Patterns

```
<xs:simpleType name="phoneType">
  <xs:restriction base="xs:string">
    <xs:pattern value="\d{3}-\d{3}-\d{4}" />
  </xs:restriction>
</xs:simpleType>
```



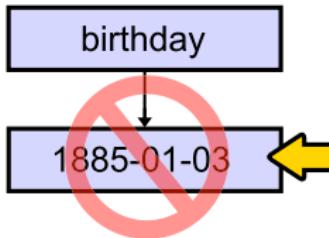
Using Enumerations

```
<xs:simpleType name="speciesType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="cat" />
    <xs:enumeration value="dog" />
    <xs:enumeration value="fish" />
  </xs:restriction>
</xs:simpleType>
```



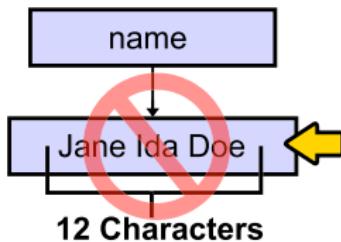
Using Dates

```
<xs:simpleType name="birthdayType">
  <xs:restriction base="xs:date">
    <xs:minInclusive value="1900-01-01" />
    <xs:maxInclusive value="2100-12-31" />
  </xs:restriction>
</xs:simpleType>
```



Restricting Strings

```
<xs:simpleType name="nameType">
  <xs:restriction base="xs:string">
    <xs:maxLength value="10" />
  </xs:restriction>
</xs:simpleType>
```



Using Advances Features in Schemas

- Use mixed content
- Use empty elements
- Use annotations
- Define namespaces
- Include other XML schema documents

Defining Mixed Content

```
<xs:complexType name="mixednameType" ① mixed="true">  
  <xs:sequence>  
    <xs:element name="middle" type="nameType" ② minOccurs="0" />  
  </xs:sequence> ③  
</xs:complexType>
```

Mixed Content Syntax

- ① Add the `mixed` attribute to the type definition and set the value to `true`.
- ② Define an element to be included inside the element.
- ③ Set the `minOccurs` to zero if you want the element to be optional.

Defining Empty Elements

```
<xs:complexType name="petType">
    <xs:attribute name="name" type="nameType" />
    <xs:attribute name="species" type="speciesType" />
    <xs:attribute name="breed" type="xs:string" />
</xs:complexType>
```

```
<pet name="Fluffy" species="cat" breed="calico" />
```

Using Annotations

```
①<xs:annotation>
  ②<xs:documentation xml:lang="en">
    ③Your desired comments here
      </xs:documentation>
    </xs:annotation>
```

- ① The annotation element.
- ② The documentation element declares the language type.
- ③ Annotation comment.

Defining Namespaces

```
<!-- schema document -->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" ①
    targetNamespace="http://denverpetclub.org/list/" ②
    xmlns="http://denverpetclub.org/list/" ③
    elementFormDefault="qualified" ④
    attributeFormDefault="unqualified"> ⑤

<!-- Instance document --> ⑥
<people xmlns:pc="http://denverpetclub.org/list/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://denverpetclub.org/list/ 07_21.xsd"> ⑦
```

Namespace Syntax

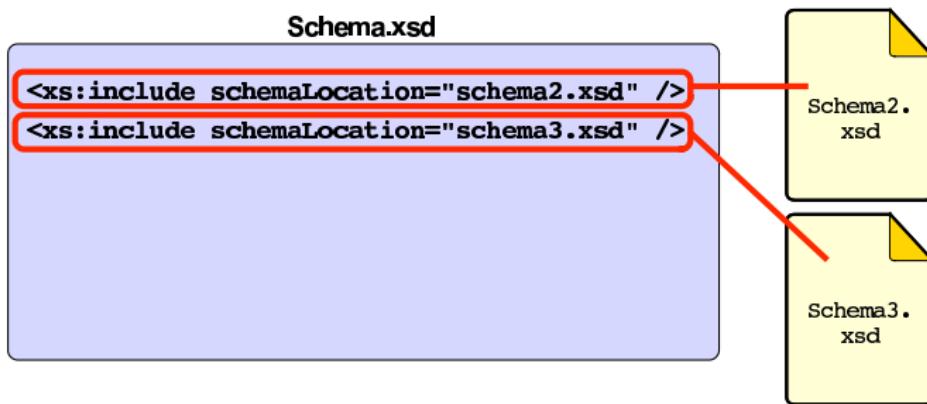
- ① The XML schema namespace.
- ② The target namespace for this schema.
- ③ The default namespace.
- ④ When you use qualified all elements must be preceded by their namespace prefix. When you use unqualified, they do not.
- ⑤ When you use qualified all attributes must be preceded by their namespace prefix. When you use unqualified they do not.
- ⑥ The default namespace.
- ⑦ Link to the target namespace and the schema file.

Assigning a Prefix

```
<!-- schema document -->
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
             targetNamespace="http://denverpetclub.org/list/"
             xmlns="http://denverpetclub.org/list/"
             elementFormDefault="qualified"
             attributeFormDefault="unqualified">

    <!-- Instance document -->
    <people xmlns:pc="http://denverpetclub.org/list/"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://denverpetclub.org/list/ :07_21.xsd:>
```

Including Other Schema Documents





Developing Web Services with Java™ Technology

Appendix B

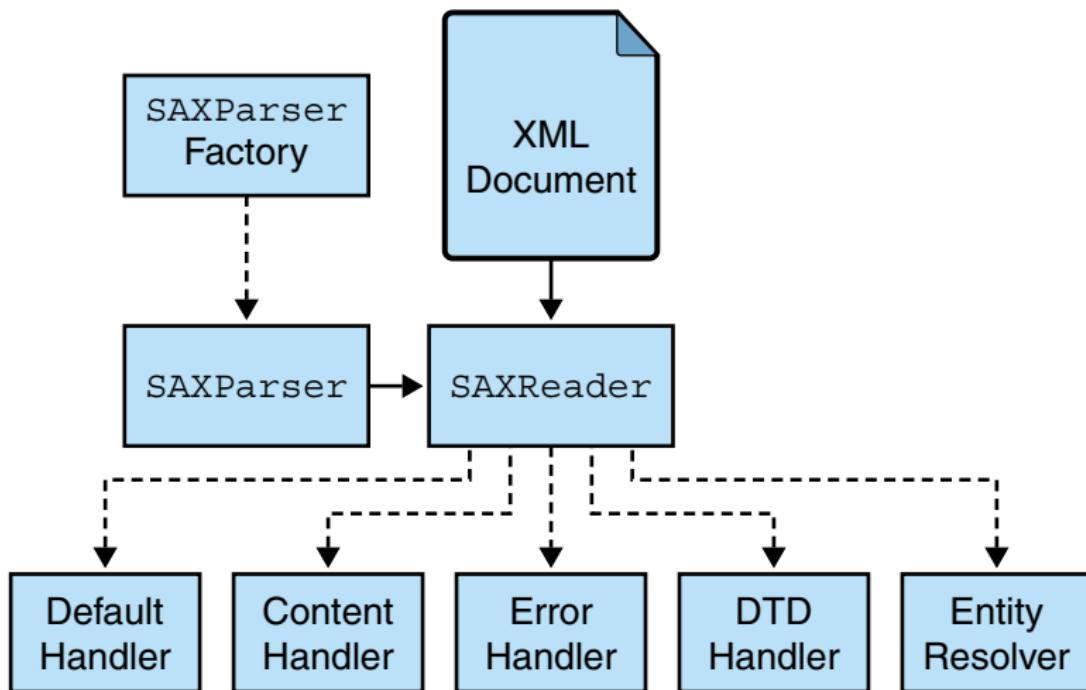
JAXB: the Java XML Binding API

Objectives

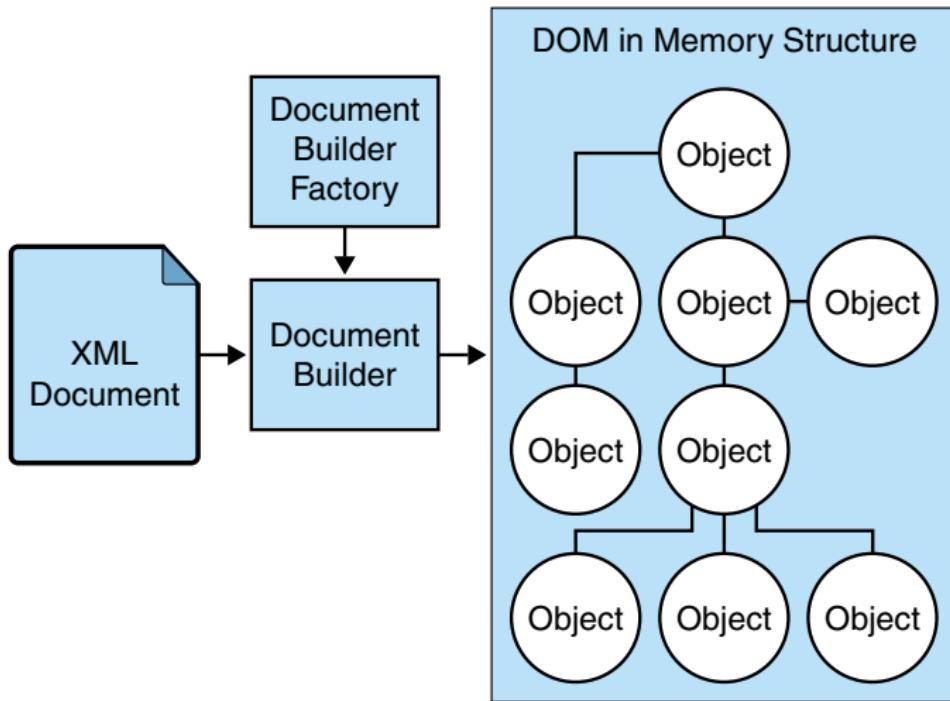
On completion of this module, you should be able to:

- understand how to use the JAX API to serialize (or “marshall”) Java objects into XML form
- understand how to use the JAX API to deserialize (or “unmarshall”) XML content into Java objects.

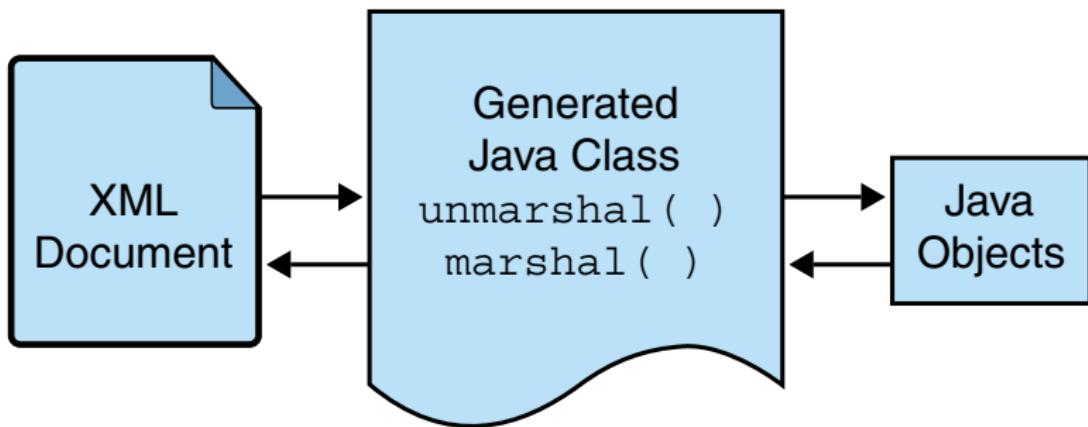
SAX Parsers



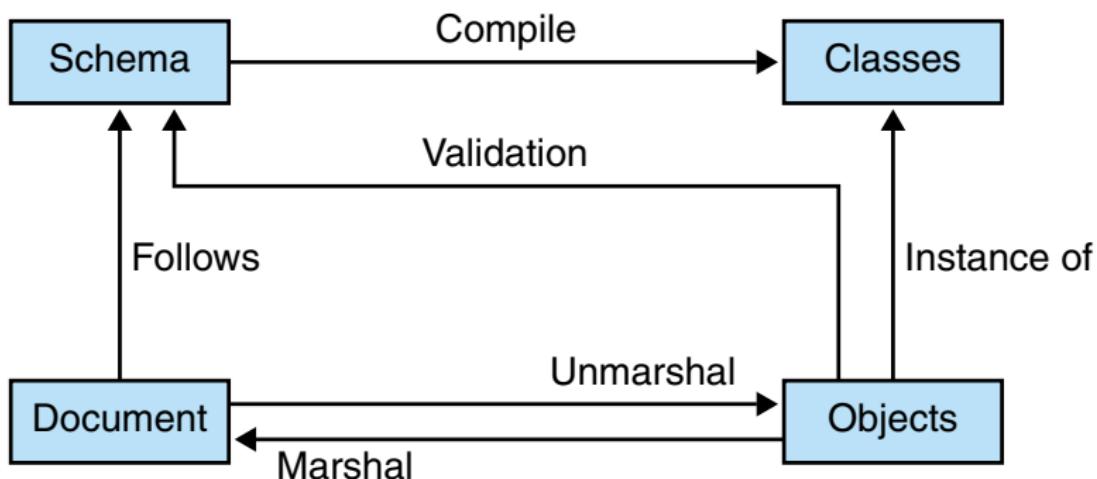
DOM Parsers



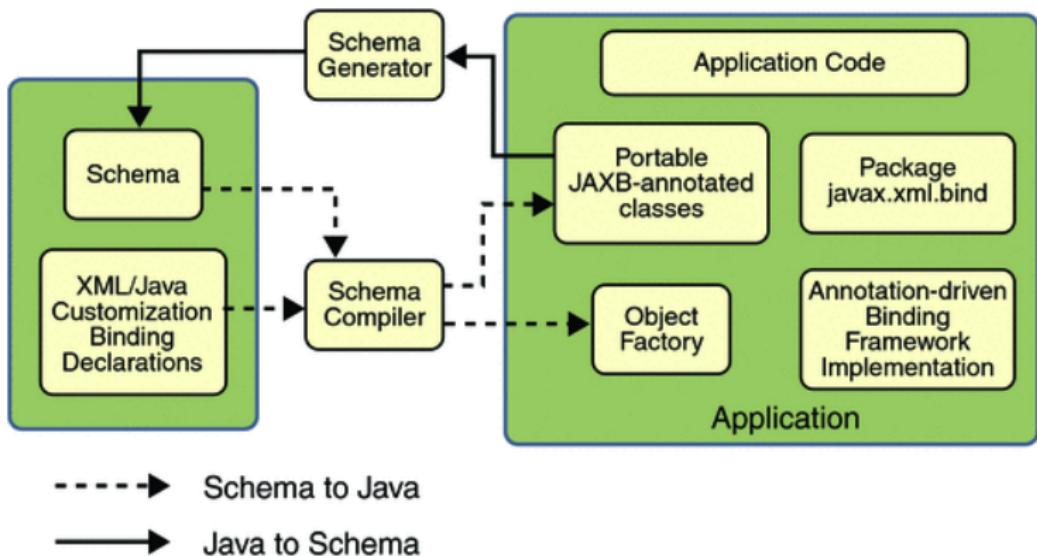
JAXB Application



JAXB Architecture



JAXB Architecture





Developing Web Services with Java™ Technology

Appendix C

JAXP and SAAJ

Objectives

On completion of this module, you should be able to:

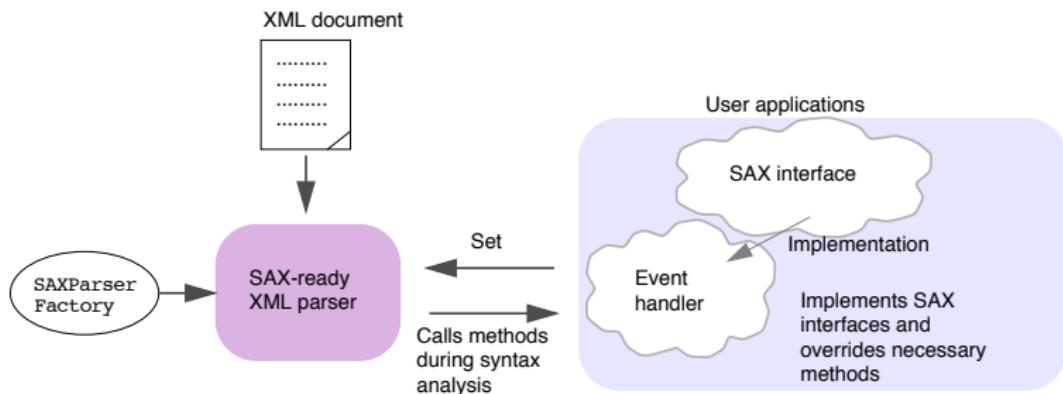
- Understand two different technologies available within Java to parse XML structures: SAX and DOM parsing.
 - Manipulate SOAP structures directly using the SAAJ API
 - > Extract information from a SOAP envelope
 - > Construct a SOAP envelope

Parsing XML in Java

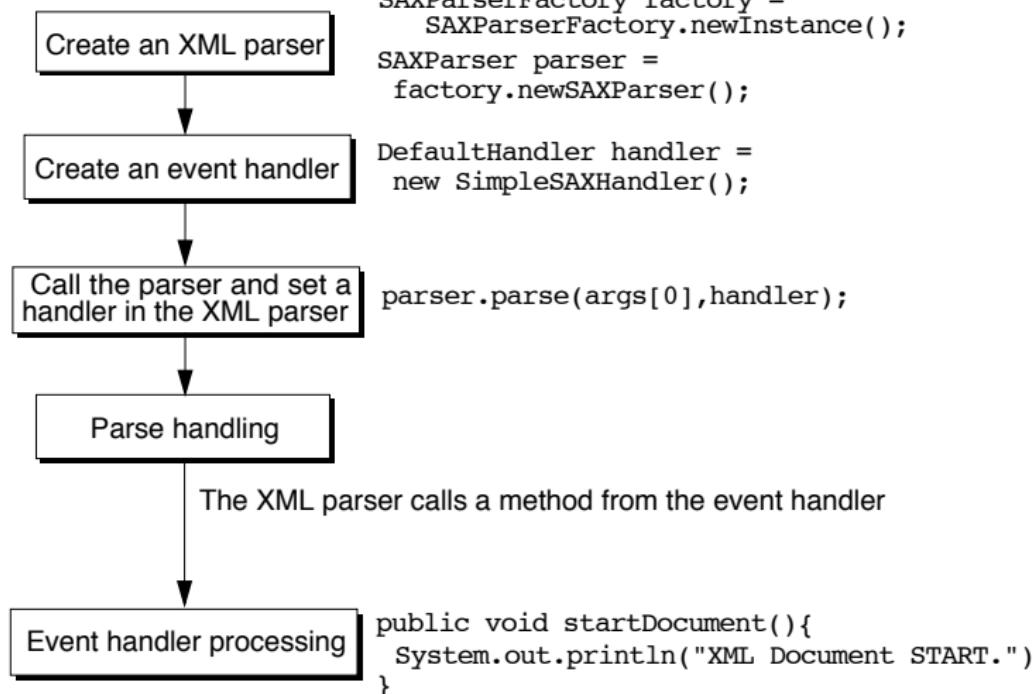
There are three different technologies to parse XML built into Java:

- SAX – the Simple API for XML Parsing
 - DOM – parsing XML to create Document Object Models
 - StAX – the Streaming API for XML Parsing

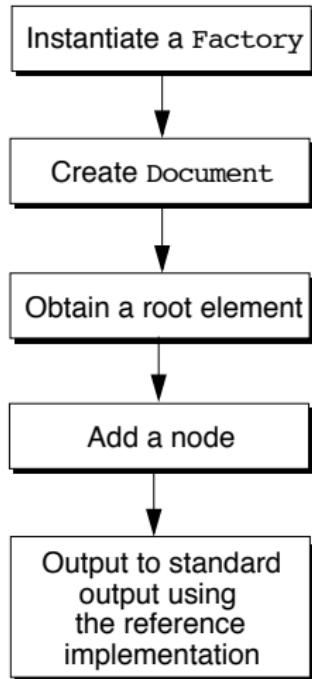
SAX Overview



Using SAX



Using DOM Parsers



```
DocumentBuilderFactory factory =  
    DocumentBuilderFactory.newInstance();  
DocumentBuilder builder =  
    factory.newDocumentBuilder();
```

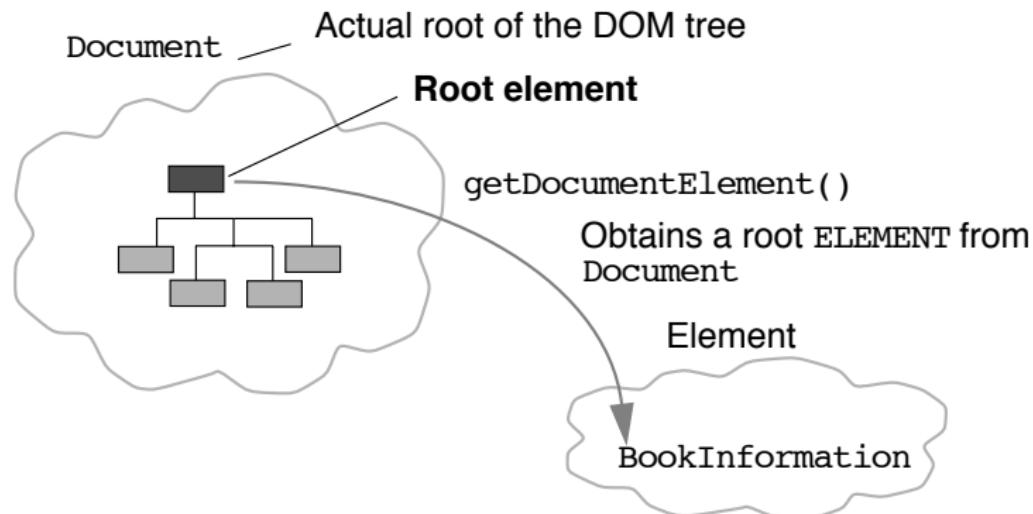
```
Document doc =  
    builder.parse( new File(args[0]) );
```

```
Element root =  
    doc.getDocumentElement();
```

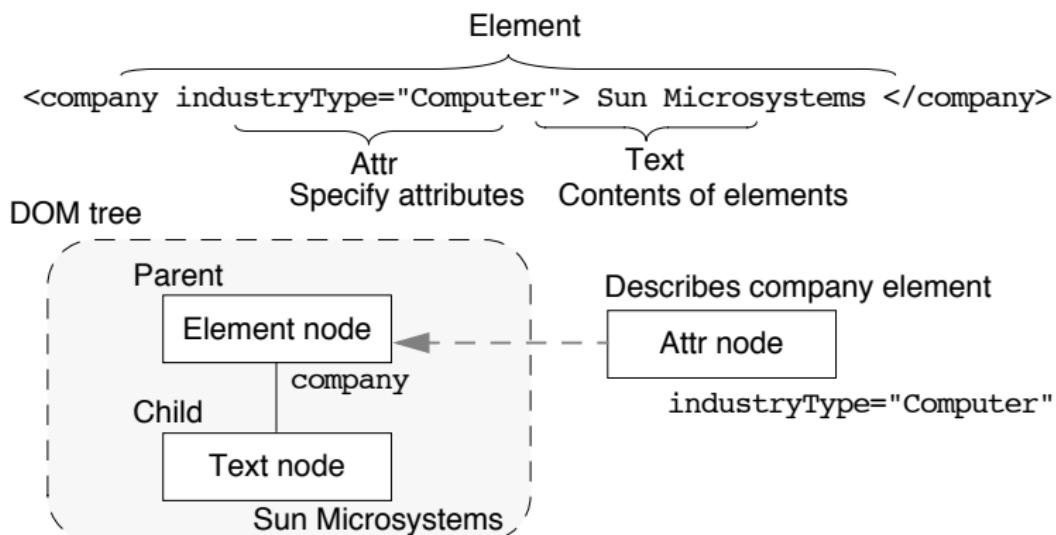
```
Comment comment =  
    doc.createComment("Training text");  
root.appendChild(comment);
```

```
XmlDocument xdoc = (XmlDocument) doc;  
xdoc.write(new OutputStreamWriter(  
    System.out));
```

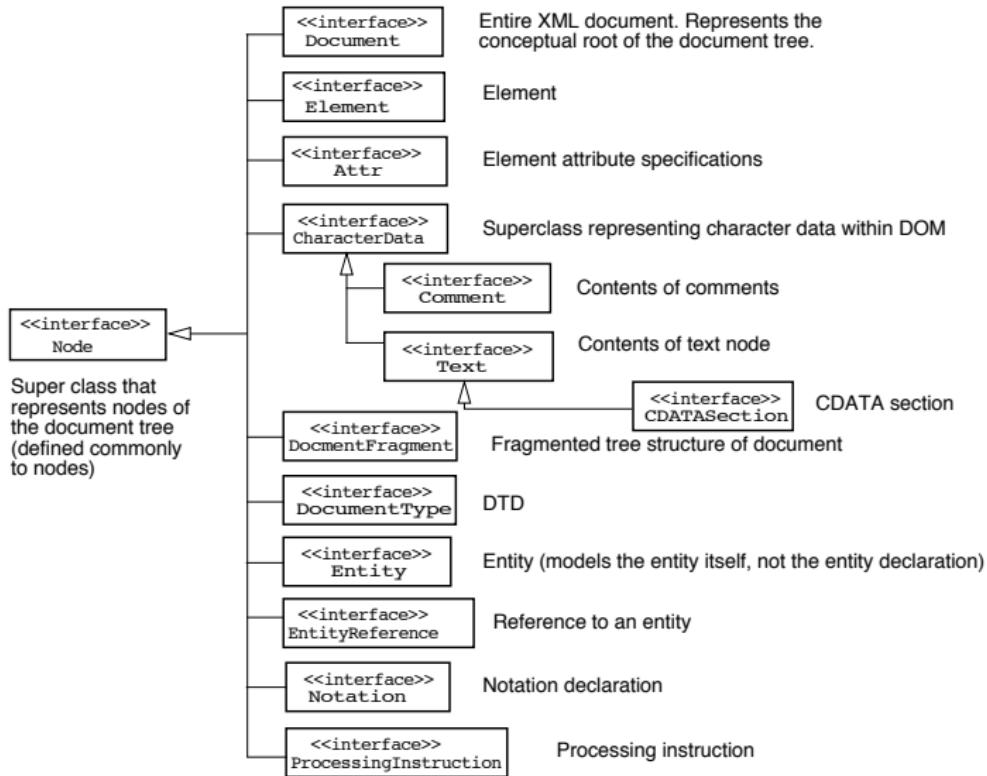
Getting Root Elements



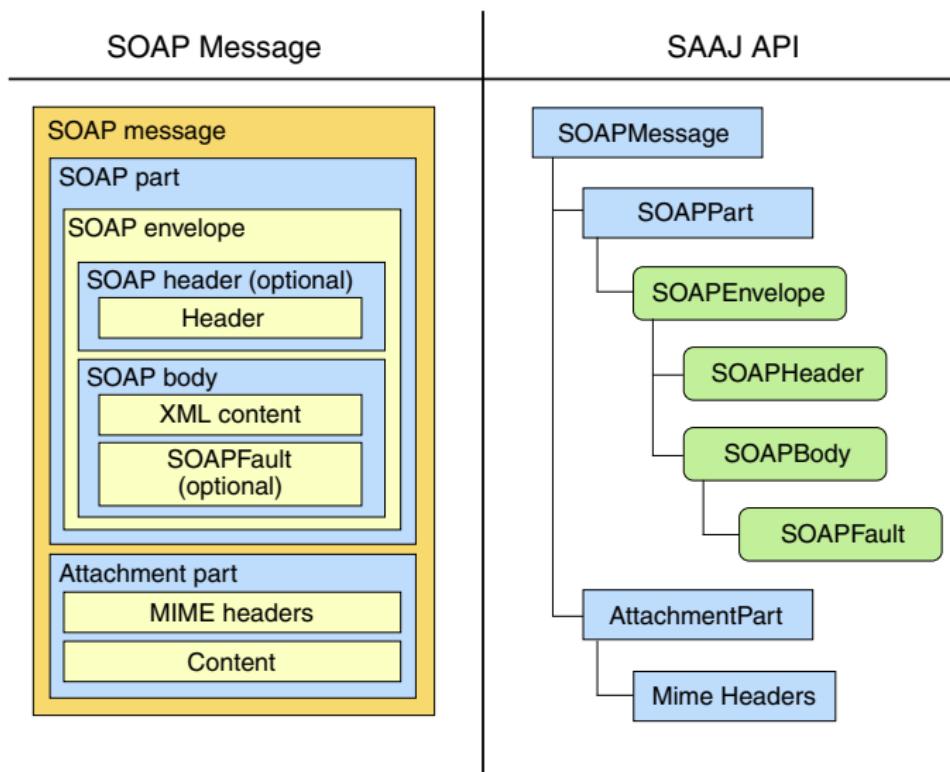
DOM Elements



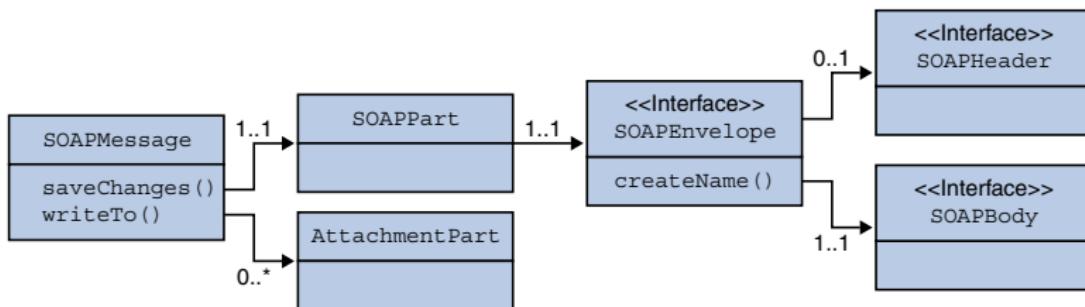
DOM Element Hierarchy



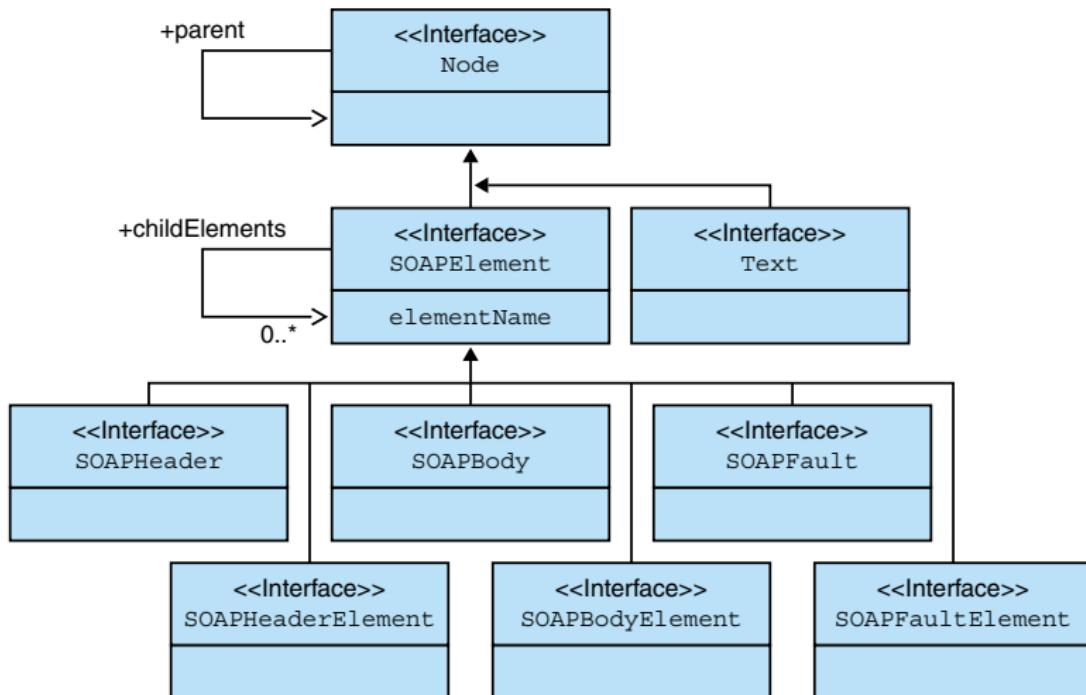
SAAJ Representation



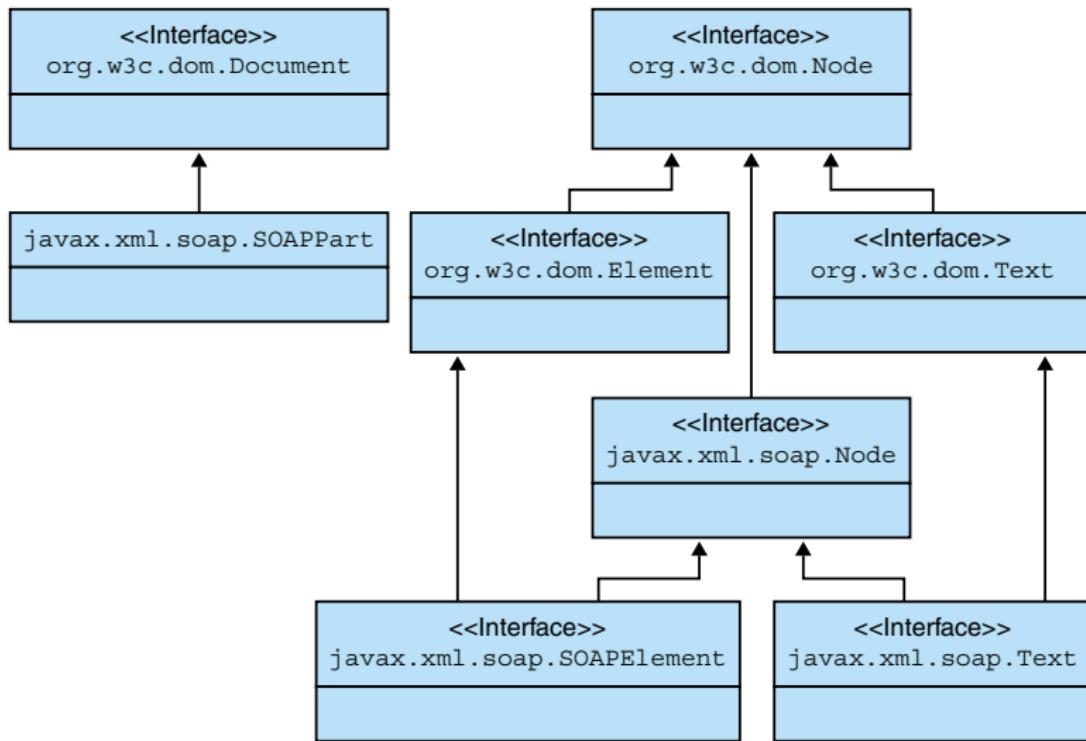
A SOAP Message in SAAJ



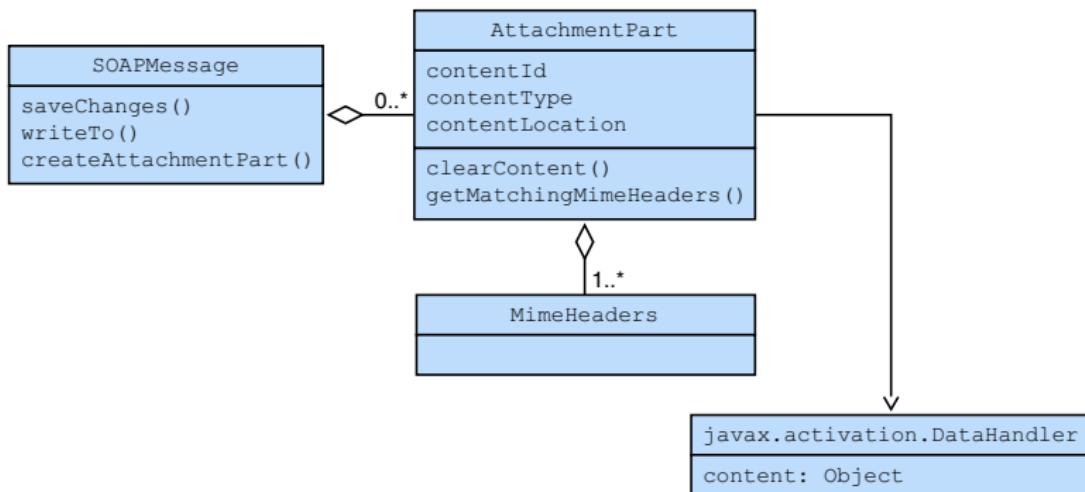
SAAJ API



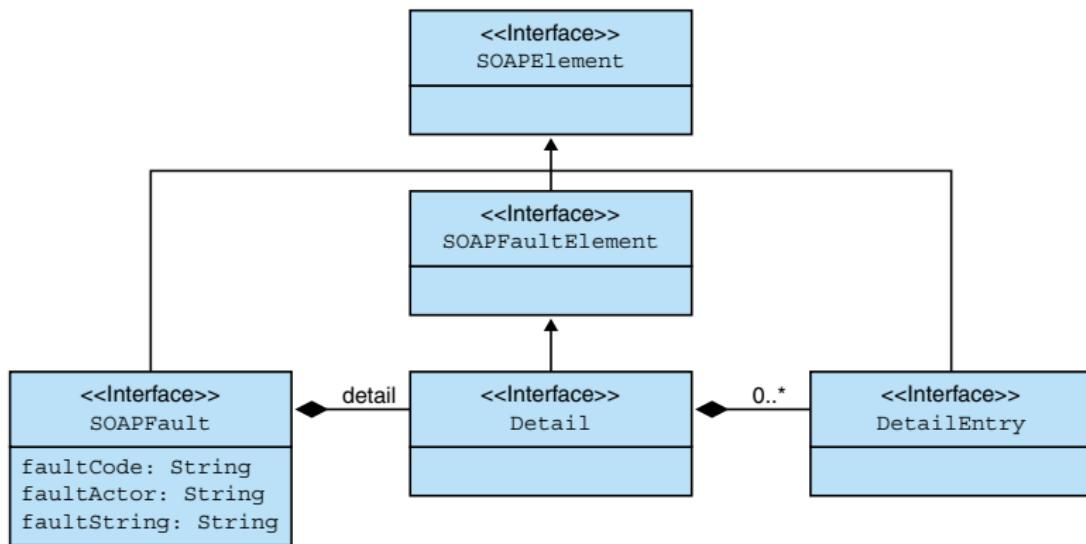
SAAJ and DOM



Attachments in SAAJ



SOAP Faults in SAAJ





Developing Web Services with Java™ Technology

Appendix D

JAX-WS Handlers

Objectives

On completion of this module, you should be able to:

- Incorporate request metadata into web service interactions
 - > Use WebServiceContext and BindingProvider to incorporate metadata
- Use JAX-WS Handlers to pre-process and post-process web service requests
 - > Define JAX-WS Handlers
 - > Incorporate handlers into processing flow through HandlerChain annotations

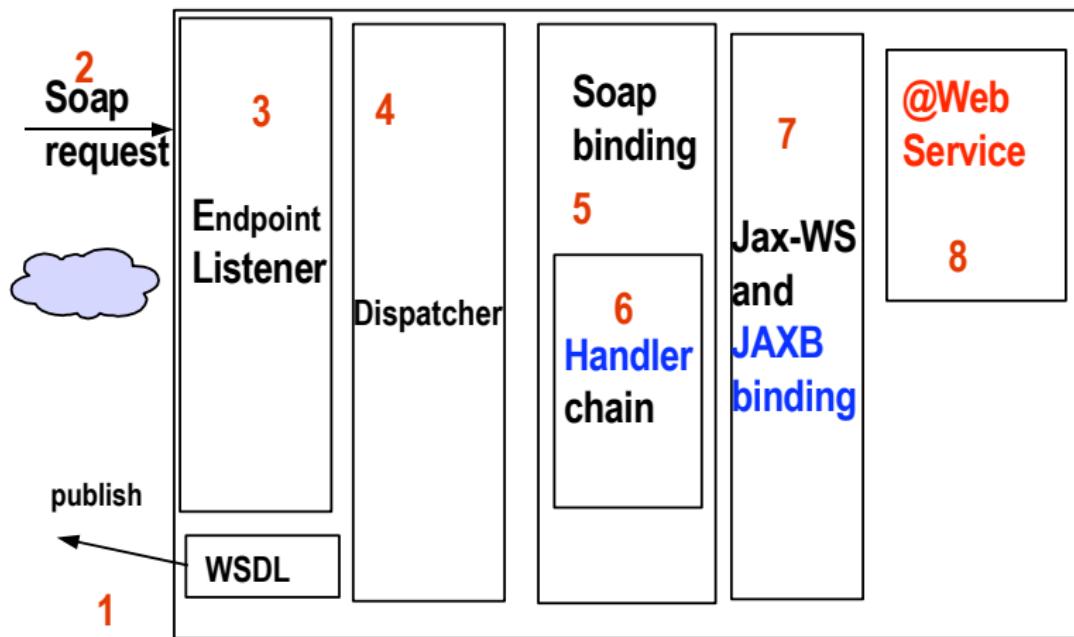
Simple POJO WS Client

```
1 public class SimpleClient {  
2     public static void main(String[] args) {  
3         AirportManagerService service =  
4             new AirportManagerService();  
5         AirportManager port =  
6             service.getAirportManagerPort();  
7         java.lang.String code = "LGA";  
8         java.lang.String name = "New_York_LaGuardia";  
9         long result = port.addAirport(code, name);  
10        System.out.println("Result = "+result);  
11    }  
12 }
```

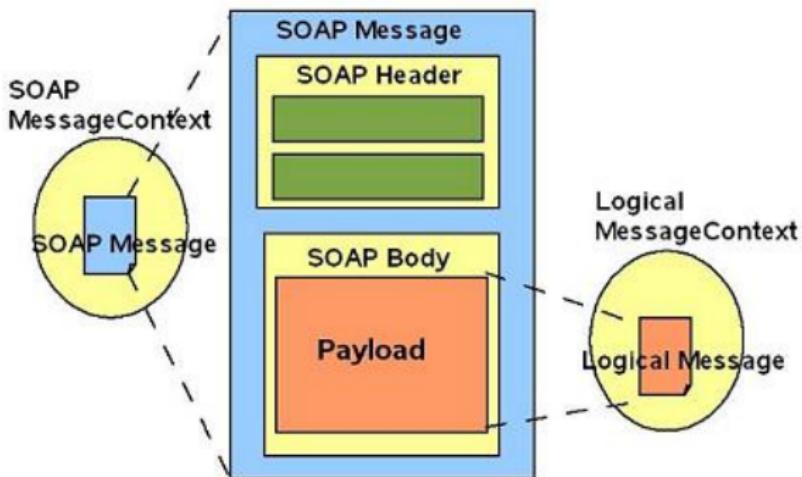
Authenticating POJO WS Client

```
1 public class AuthSimpleClient {  
2     public static void main(String[] args) {  
3         AirportManagerService service =  
4             new AirportManagerService();  
5         AirportManager port = service.getAirportManagerPort();  
6         Map<String, Object> reqCtx =  
7             ((BindingProvider) port).getRequestContext();  
8         reqCtx.put(BindingProvider.USERNAME_PROPERTY,  
9                     "tracy");  
10        reqCtx.put(BindingProvider.PASSWORD_PROPERTY,  
11                     "password");  
12        java.lang.String code = "LGA";  
13        java.lang.String name = "New_York_LaGuardia";  
14        long result = port.addAirport(code, name);  
15        System.out.println("Result_=_" + result);
```

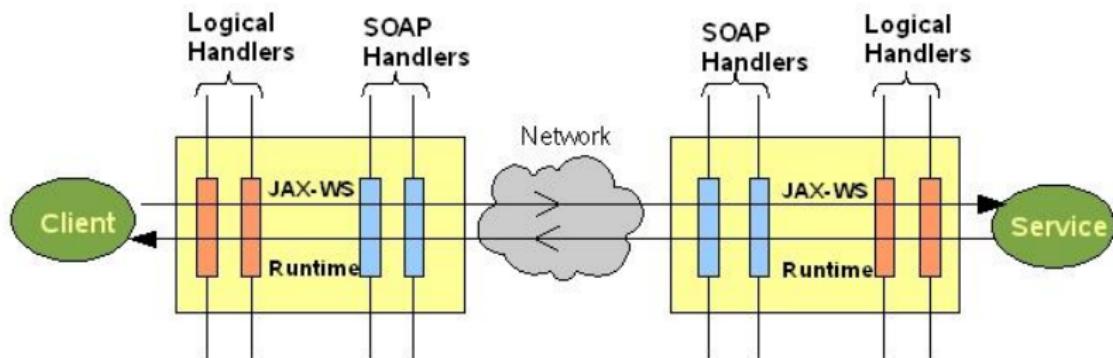
JAX-WS Runtime Architecture



JAX-WS Handler Types



Executing JAX-WS Handlers



Writing a JAX-WS Handler

A basic handler should implement the following three methods in Handler:

```
1 public interface
2 Handler<C extends MessageContext> {
3     boolean handleMessage(C ctx);
4     boolean handleFault(C ctx);
5     void close(MessageContext ctx);
6 }
```

Types of JAX-WS Handler

```
1 public interface  
2 SOAPHandler<C extends SOAPMessageContext>  
3 extends javax.xml.ws.handler.Handler<C> {  
4     public abstract java.util.Set getHeaders();  
5 }
```

```
1 public interface  
2 LogicalHandler<C extends LogicalMessageContext>  
3 extends javax.xml.ws.handler.Handler<C> {  
4 }
```

An Authentication Handler

```
1 public class AuthenticationHandler
2     implements SOAPHandler<SOAPMessageContext> {
3     public boolean
4         handleMessage(SOAPMessageContext smc) {
5             Boolean outboundProperty =
6                 (Boolean) smc.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
7             SOAPMessage msg = smc.getMessage();
8             if (outboundProperty) {
9                 try { embedAuthenticationData(msg); }
10                catch(Exception ex)
11                {}
12            } else {
13                try { validateAuthenticationData(msg); }
14                catch(Exception ex )
15                {}
16            }
17            return true;
18        }
19    }
```

An Authentication Handler

Embedding Authentication Data

```
19 private void  
20 embedAuthenticationData(SOAPMessage msg)  
21     throws SOAPException {  
22     SOAPEnvelope envelope = msg.getSOAPPart().getEnvelope();  
23     Name authName = envelope.createName("auth");  
24     Name userName = envelope.createName("user");  
25     Name passwordName = envelope.createName("password");  
26     SOAPHeader header = msg.getSOAPHeader();  
27     SOAPHeaderElement newHeader =  
28         header.addHeaderElement(authName);  
29     newHeader.addAttribute(userName, "tracy");  
30     newHeader.addAttribute(passwordName, "password");  
31 }
```

An Authentication Handler

Validating Authentication Data

```
32 private void
33 validateAuthenticationData(SOAPMessage msg)
34     throws SOAPException {
35     SOAPEnvelope envelope = msg.getSOAPPart().getEnvelope();
36     Name authName = envelope.createName("auth");
37     Name userName = envelope.createName("user");
38     Name passwordName = envelope.createName("password");
39     SOAPHeader header = msg.getSOAPHeader();
40     for ( Iterator authNodes = header.getChildElements(authName);
41           authNodes.hasNext(); ) {
42         SOAPHeaderElement authHeader = (SOAPHeaderElement) authNodes.next();
43         String user = authHeader.getAttributeValue(userName);
44         String password = authHeader.getAttributeValue(passwordName);
45         validate(user, password);
46     }
47 }
```