

Developing Web Services Using Java Technology

Volume II • Student Guide

DWS-4050-EE6 Rev A

D65185GC11
Edition 1.1
September 2010
D69078

ORACLE®

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Disclaimer

This document contains proprietary information, is provided under a license agreement containing restrictions on use and disclosure, and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except as expressly permitted in your license agreement or allowed by law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

Sun Microsystems, Inc. Disclaimer

This training manual may include references to materials, offerings, or products that were previously offered by Sun Microsystems, Inc. Certain materials, offerings, services, or products may no longer be offered or provided. Oracle and its affiliates cannot be held responsible for any such references should they appear in the text provided.

Restricted Rights Notice

If this documentation is delivered to the U.S. Government or anyone using the documentation on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS

The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

Trademark Notice

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

This page intentionally left blank.

Oscar Hernando Nivia Aragon (onivia@hotmail.com) has a
non-transferable license to use this Student Guide.

This page intentionally left blank.

Oscar Hernando Nivia Aragon (onivia@hotmail.com) has a
non-transferable license to use this Student Guide.

Contents

Course Map	xxx
Topics Not Covered	xxxii
How Prepared Are You?	xxxiii
Introductions	xxxiv
How to Use Course Materials	xxxv
Conventions	xxxv
Course Projects	xxxvii
The Traveller Project	xxxvii
The Auction Project	xlii
1 Introduction to Web Services	1-1
What Is a Web Service?	1-2
Exploring the Need for Web Services	1-3
The Drive Towards Web Services	1-3
Challenges of IT Integration	1-4
Conceptual Model	1-5
Exposing Application Functions as a Web Service	1-7
Properties of a Web Service	1-8
Comparison With Other Remote Component Access Mechanisms	1-8
Benefits of Web Services	1-9

CONTENTS**CONTENTS**

Web Services in B2B and B2C Scenarios	1-10
Service-Oriented Architecture (SOA) and Web Services	1-10
Limitations of Web Services	1-11
Characteristics of a Web Service	1-12
Web Service Elements	1-12
Web Service Life Cycle	1-13
Major Web Services Models	1-16
Web Service Initiatives, Standards, and Specifications	1-18
Governing Bodies	1-18
Web Service Initiatives	1-19
Web Service Specifications and APIs	1-19
The SOAP Specification	1-19
The Web Services Description Language	1-20
Web Services Interoperability	1-22
An Interoperable Architecture	1-22
The WS-I Organization	1-24
WS-I Basic Profile Support	1-25
WS-I Basic Profile Web Services	1-26
Development Approaches	1-27
Code First Approach	1-28
Contract First Approach	1-28
Web Service Endpoints	1-29
JavaEE Web Service Support	1-29
2 Using JAX-WS	2-1
Additional Resources	2-2

CONTENTS**CONTENTS**

Overview of JAX-WS	2-3
Creating a Web Service Using JAX-WS	2-9
Creating a Web Service Using JAX-WS: Bottom-Up	2-9
Customizing the JAX-WS Web Service	2-17
Creating a Web Service Using JAX-WS: Top-Down	2-22
Writing a WSDL Description of a Service	2-24
Generating JAX-WS Artifacts	2-27
Schema Validation	2-34
Comparing Development Approaches	2-35
Strong Typing for Web Services	2-35
Benefits and Costs of Starting from a Java Development Approach	2-36
Benefits and Costs of Starting from WSDL Development Approach	2-37
Deploying POJO Web Service Providers	2-38
Debugging Web Service Interactions	2-39
3 SOAP and WSDL	3-1
Relevance	3-2
Additional Resources	3-3
SOAP	3-4
Basic Structure of a SOAP Message	3-4
The SOAP Specification	3-4
Messaging Over the Web	3-5
Nodes	3-6
SOAP Message Format	3-7
Transport Protocols for SOAP	3-11
WSDL	3-16

Primary Elements Contained in a WSDL File	3-17
Variations of WSDL	3-23
Evolution of WSDL	3-39
4 JAX-WS and JavaEE	4-1
Deploying a Web Service to a Web Container	4-2
Authentication and Authorization	4-5
Creating a Web Service From an EJB	4-17
Limitations of POJO Web Services	4-17
Enterprise Java Beans	4-19
Transaction Management	4-24
Scalability	4-25
5 Implementing More Complex Services Using JAX-WS	5-1
Additional Resources	5-2
Complex Arguments and Return Values	5-3
Exception Handling	5-7
Mapping WSDL-to-Java Exception Classes	5-7
The JAX-WS API Exception Classes	5-9
Using Predefined Exception Classes in Web Services	5-11
Using Custom-Defined Exception Classes in Web Services	5-12
6 JAX-WS Web Service Clients	6-1
Web Service Clients	6-2
Asynchronous Interactions	6-8
7 Introduction to RESTful Web Services	7-1
What are RESTful Web Services?	7-2

Advantages and Disadvantages	7-7
8 RESTful Web Services: JAX-RS	8-1
Additional Resources	8-2
JAX-RS	8-3
Mapping REST Principles to JAX-RS Constructs	8-5
Deploying a JAX-RS Web Service Provider	8-12
9 JAX-RS Web Service Clients	9-1
Writing JAX-RS Clients Using HttpURLConnection	9-2
Writing JAX-RS Clients Using the Jersey Client API	9-6
10 JAX-RS and JavaEE	10-1
Deploying a Web Service to a Web Container	10-2
Creating a Web Service From an EJB	10-16
Transaction Management	10-19
Scalability	10-20
11 Implementing More Complex Services Using JAX-RS and Jersey	11-1
Additional Resources	11-2
Parameters and Return Values	11-3
Linking To Other Resources	11-8
Custom Marshalling and Unmarshalling	11-12
Exception Handling in JAX-RS	11-17
Using Resources and Sub-resources	11-19
Resource Scopes	11-24
12 Trade-Offs Between JAX-WS and JAX-RS Web Services	12-1

CONTENTS**CONTENTS**

Additional Resources	12-2
Comparing SOAP and REST	12-3
Impedance Mismatch	12-4
JAX-WS Web Services	12-6
JAX-RS Web Services	12-9
SOAP Compared to REST	12-11
13 Web Services Design Patterns	13-1
Additional Resources	13-2
Design Patterns in the Context of Web Services	13-3
Web Services Design Patterns	13-5
“PAOS” Interactions	13-5
Asynchronous Interaction	13-8
JMS Bridge	13-24
Web Services Deployment Patterns	13-28
HTTP Load Balancing	13-28
Container Cluster	13-32
14 Best Practices and Design Patterns for JAX-WS	14-1
Web Services Design Patterns	14-2
Web Service Cache	14-2
Web Service Broker	14-5
Web Service Logger	14-8
Handling Exceptions in Web Services	14-12
Handling Exceptions in Web Service Client	14-13
Exception Management in Web Services	14-15
15 Best Practices and Design Patterns for JAX-RS	15-1

CONTENTS**CONTENTS**

A XML Schema	A-1
Additional Resources	A-2
Creating a Basic Schema	A-3
Structuring XML Schemas	A-10
Using Data Types in an XML Schema	A-12
Using Advances Features in Schemas	A-17
B JAXB: the Java XML Binding API	B-1
Schema Validation	B-5
C JAXP and SAAJ	C-1
Additional Resources	C-2
JAXP	C-3
SAX	C-3
DOM	C-6
SAAJ	C-10
D JAX-WS Handlers	D-1
Incorporating Request Metadata	D-2
Using JAX-WS Handlers	D-4
E Code Listings	E-1

Oscar Hernando Nivia Aragon (onivia@hotmail.com) has a
non-transferable license to use this Student Guide.

List of Figures

1	Course Map	xxx
2	Course Map – JAXWS	xxxi
3	Use Cases for Traveller Project	xxxviii
4	Domain Types for Traveller Project	xxxviii
5	Application Services for the Traveller Project	xl
6	DAO Classes for the Traveller Project	xli
7	Use Cases for Auction System	xlii
8	Domain Types for Auction System	xlv
1.1	A Web Service	1-3
1.2	Conceptual Model	1-6
1.3	Exposing Business Logic as Web Services	1-7
1.4	Web Service Elements	1-12
1.5	Web Service Life Cycle	1-14
1.6	Interacting with a Web Service via Registry	1-15
1.7	Interacting with a Web Service – Direct	1-16
1.8	An Interoperable Architecture	1-23
1.9	Development Approaches	1-27
1.10	JavaEE 6 APIs	1-30
2.1	JAXWS Artifacts	2-5

LIST OF FIGURES***LIST OF FIGURES***

2.2 Starting From a Java Class	2-10
2.3 Sample SOAP Request for AirportManager	2-15
2.4 Sample SOAP Response from AirportManager	2-15
2.5 Raw SOAP/HTTP Request	2-16
2.6 Raw SOAP/HTTP Response	2-17
2.7 Target Namespace for Application Elements	2-21
2.8 Starting From a WSDL Description	2-23
2.9 Structure of a WSDL file	2-23
2.10 Definition of a Service Using WSDL	2-25
3.1 A Simple Web Service Interaction	3-5
3.2 Extensible Web Service Interactions	3-7
3.3 Extensible Message Representation	3-8
3.4 Sample SOAP Message	3-9
3.5 Sample SOAP Response	3-9
3.6 SOAP Message embedded in HTTP	3-13
3.7 SOAP Request Embedded in HTTP	3-14
3.8 SOAP Response Embedded in HTTP	3-15
3.9 Structure of a WSDL File	3-18
3.10 WSDL Elements as UML	3-19
3.11 WSDL Interaction Scenarios	3-20
3.12 WSDL binding Element	3-21
3.13 Sample SOAP Message	3-23
3.14 Sample SOAP Message, RPC Style	3-26
3.15 Sample RPC/literal SOAP Message with Complex Value	3-28
3.16 Sample Message for Document-style Service	3-33

*LIST OF FIGURES**LIST OF FIGURES*

3.17 Sample Message Using Document/literal “wrapped”	3-38
3.18 Representation of concepts defined by WSDL 1.1 and WSDL 2.0 documents	3-40
4.1 Structure of WAR file with a JAX-WS-based web service fig:Structure of WAR file with a JAX-WS-based web service	4-4
4.2 EJB Container Services	4-20
5.1 JAX-WS Exception Class Hierarchy	5-10
6.1 Developing Web Service Clients	6-3
6.2 A Prototypical Operation	6-9
6.3 An Asynchronous Request	6-11
6.4 A Prototypical Operation – Revisited	6-12
6.5 Async Requests – Another Approach	6-13
7.1 Standard HTTP Methods	7-4
10.1 Simple Approach to Deploying a JAX-RS Web Service	10-4
11.1 Returning Complex Values via XML	11-9
11.2 Returning Complex Values the REST Way	11-9
11.3 Traveller Project Domain Recap	11-19
11.4 Valid Queries	11-23
12.1 Dave Podnar’s 5 Stages of Dealing with Web Services	12-3
12.2 Impedance Mismatch	12-4
12.3 JAXB Mapping XML to Java	12-5
12.4 JAX-WS – WSDL to Java	12-6
12.5 JAX-WS – SOAP to Java	12-7
12.6 JAX-RS Mapping Message to Java	12-9

LIST OF FIGURES***LIST OF FIGURES***

12.7 Approaches to Web Services Development	12-12
12.8 JAX-WS Activities	12-12
12.9 JAX-RS Resources	12-13
13.1 Simple Web Services Interaction	13-4
13.2 Possible modes of interaction in WSDL 1.1	13-6
13.3 “PAOS” Interactions	13-7
13.4 A Simple Strategy	13-10
13.5 Server Push	13-12
13.6 Server Push Class Diagram	13-13
13.7 Server Push Sequence Diagram	13-14
13.8 Server Push Class Diagram - Second Approach	13-15
13.9 Server Push Sequence Diagram - Second Approach	13-16
13.10 Complex Strategies: JAX-WS Provider API	13-19
13.11 JMS Bridge Scenario	13-25
13.12 JMS Bridge Solution	13-26
13.13 Application Partitioning	13-30
13.14 Load Balancing	13-31
14.1 WS Handlers	14-3
14.2 Opportunities for Caching	14-4
14.3 Web Service Broker	14-6
14.4 Web Service Broker Interaction	14-7
14.5 Opportunities for Logging	14-9
14.6 Sample Service Using Handlers	14-10
14.7 Handler Configuration	14-10
A.1 Creating a Basic Schema	A-4

*LIST OF FIGURES**LIST OF FIGURES*

A.2 Linking to a Schema	A-5
A.3 Determining the Number of Items	A-6
A.4 Adding Subelements	A-6
A.5 Using the Choice Element	A-7
A.6 Declaring an Attribute	A-8
A.7 Deriving a New Element	A-9
A.8 Structuring XML Schemas	A-10
A.9 Defining XML Schema Types	A-11
A.10 Applying Types to a Schema	A-11
A.11 Defining Simple XML Schema Types	A-13
A.12 Restricting an Integer	A-13
A.13 Using Patterns	A-14
A.14 Using Enumerations	A-15
A.15 Using Dates	A-15
A.16 Restricting Strings	A-16
A.17 Defining Mixed Content	A-17
A.18 Defining Empty Elements	A-18
A.19 Using Annotations	A-19
A.20 Defining Namespaces	A-20
A.21 Assigning a Prefix	A-21
A.22 Including Other Schema Documents	A-21
B.1 SAX Parsers	B-2
B.2 DOM Parsers	B-2
B.3 JAXB Application	B-3
B.4 JAXB Architecture	B-3

LIST OF FIGURES***LIST OF FIGURES***

B.5 JAXB Architecture	B-4
C.1 SAX Overview	C-3
C.2 Using SAX	C-5
C.3 Using DOM Parsers	C-7
C.4 Getting Root Elements	C-8
C.5 DOM Elements	C-9
C.6 DOM Element Hierarchy	C-9
C.7 SAAJ Representation	C-10
C.8 A SOAP Message in SAAJ	C-11
C.9 SAAJ API	C-13
C.10 SAAJ and DOM	C-14
C.11 Attachments in SAAJ	C-14
C.12 SOAP Faults in SAAJ	C-16
D.1 JAX-WS Runtime Architecture	D-4
D.2 JAX-WS Handler Types	D-5
D.3 Executing JAX-WS Handlers	D-5

List of Tables

1.1	Web Services vs Other Protocols	1-9
1.2	Basic Web Service Elements	1-13
1.3	Web Service Specifications and APIs	1-20
2.1	WSDL-to-Java Technology Component Translations	2-6
3.1	SOAP Namespaces	3-9
4.1	Authorization Annotations	4-6
4.2	Properties Available via MessageContext	4-14
4.3	Properties Available via MessageContext	4-15
7.1	Standard Semantics for HTTP Methods	7-4
11.1	Standard Entity Types in JAX-RS	11-12
11.2	Injectable Values	11-27
15.1	HTTP Action Semantics – Examples	15-2
15.2	HTTP Status Codes – 2xx	15-4
15.3	HTTP Status Codes – 3xx	15-5
15.4	HTTP Status Codes – 4xx	15-6

LIST OF TABLES

LIST OF TABLES

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Oscar Hernando Nivia Aragon (onivia@hotmail.com) has a
non-transferable license to use this Student Guide.

List of Code Examples

2.1	A Simple Service: AirportManager	2-11
2.2	A Simplest Web Service	2-11
2.3	Sample ant task for apt	2-13
2.4	Generated WSDL for AirportManager Class	2-13
2.5	XML Schema Generated for AirportManager	2-14
2.6	Custom WSDL Description: Class NamedAirportManager	2-18
2.7	Customized Generated WSDL	2-18
2.8	Customized Generated XML Schema	2-19
2.9	Capturing Overloaded Methods	2-20
2.10	Generated WSDL – Overloaded Operations	2-20
2.11	Custom WSDL Description: Custom Namespace	2-21
2.12	PassengerManager portType	2-26
2.13	XML Schema Type for addPassenger	2-26
2.14	WSDL Bindings for PassengerManager	2-27
2.15	Generated Java SEI	2-29
2.16	Service Implementation Class	2-30
2.17	Embedded Customization: Package	2-31
2.18	Embedded Customization: Class	2-32
2.19	Embedded Customization: Method	2-32
2.20	Class Generated From Customized WSDL	2-33

2.21 How to Enable Schema Validation	2-34
2.22 Simple Standalone Server	2-38
2.23 Finer Control over Standalone Server	2-39
3.1 WSDL Definition, RPC-Style	3-25
3.2 WSDL Message Definition, RPC-Style	3-25
3.3 WSDL Bindings Definition, RPC-Style	3-26
3.4 WSDL Description of Message with Complex Type	3-27
3.5 WSDL Schema Type for Complex Type	3-28
3.6 Specifying RPC/literal in Java	3-29
3.7 WSDL Description of Document-style Service	3-30
3.8 Message Description in Document-style Service	3-31
3.9 XML Schema Type for Document-style Service	3-32
3.10 WSDL Bindings for Document-style Service	3-32
3.11 Java-based Specification of Document/literal Style	3-34
3.12 WSDL Message Description Using Document/literal “wrapped”	3-36
3.13 XML Schema for Web Service Using Document/literal “wrapped”	3-37
3.14 WSDL Bindings Using Document/literal “wrapped”	3-37
3.15 Java-to-WSDL Using Document/literal “wrapped”	3-39
4.1 Deploying a JAX-WS Service to a Web Container	4-4
4.2 A Secured Web Service	4-7
4.3 Securing Resource URLs	4-8
4.4 Configuring Login	4-9
4.5 Fragment of Sample sun-web.xml File	4-10
4.6 Programmatic Authentication and Authorization	4-11

4.7	Dependency Injection and WebServiceContext	4-12
4.8	Logging Callers in JAXWS	4-13
4.9	Authenticating POJO WS Client	4-16
4.10	A POJO Data Access Object	4-17
4.11	A Simple Operation in a DAO	4-18
4.12	A More Complex POJO Web Service	4-19
4.13	A Data Access Object – Revisited	4-23
4.14	An EJB Data Access Object – AirportDAO	4-23
4.15	Creating Web Services from EJBs	4-24
4.16	A More Complex EJB Web Service	4-25
4.17	Singleton Web Service EJBs	4-27
4.18	Singleton Web Service EJB with Concurrency Policy	4-28
5.1	Service Class with Complex Return Type	5-4
5.2	Class with JAXB Annotations	5-4
5.3	Mapping Elements and Attributes	5-5
5.4	XML Representation	5-5
5.5	Enums in XML	5-6
5.6	FlightManager XML Schema	5-6
5.7	WSDL Code Fragment – fault Message	5-8
5.8	XML Schema Type Fragment – fault Message	5-8
5.9	Fault Type Generated by JAX-WS	5-9
5.10	Exception Type Generated by JAX-WS for Fault	5-9
5.11	Client-Side SOAPFaultException Exception Handling	5-11
5.12	Server-side SOAPFaultException Exception Handling	5-12
6.1	Fragment of client-side ant build script.	6-4
6.2	Simple POJO WS Client	6-5

6.3	Restricting the Schema	6-6
6.4	Validating POJO WS Client	6-6
6.5	JavaEE Web Service Clients	6-7
6.6	One-Way Operations	6-10
6.7	Requesting Asynchronous Support	6-14
6.8	Asynchronous Operations	6-14
6.9	Client Using Response	6-15
6.10	Client Using AsyncHandler – AsyncHandler	6-15
6.11	Client Using AsyncHandler – Client	6-16
8.1	JAX-RS IDs – Simple Path	8-6
8.2	JAX-RS Path with Embedded Parameters	8-7
8.3	JAX-RS Path with Form Parameters	8-8
8.4	Use Standard HTTP Methods: GET	8-9
8.5	Use Standard HTTP Methods: POST	8-9
8.6	JAX-RS Support for Multiple Representations	8-10
8.7	JAX-RS Parameters	8-11
8.8	Explicit Declaration of Root Resources	8-12
8.9	Runtime Retrieval of Root Resources	8-13
8.10	Deploying Within a Java VM	8-14
9.1	Simplest JAX-RS Java Client	9-3
9.2	PathParam Java Client	9-3
9.3	FormParam Java Client	9-5
9.4	Simplest Jersey Client	9-7
9.5	QueryParam Jersey Client	9-8
9.6	Specifying Expected Return Type	9-9
9.7	Submitting Form Data	9-10

LIST OF CODE EXAMPLESLIST OF CODE EXAMPLES

9.8 Obtaining Reply Metadata – Client	9-10
9.9 Simple Client with Logging Filter	9-11
10.1 A Simple POJO Web Service Using JAXRS	10-2
10.2 Deploying a Web Service on JavaSE Using JAXRS	10-3
10.3 JAX-RS Deployment: Alternative Approach #1	10-5
10.4 JAX-RS Deployment: Alternative Approach #2	10-6
10.5 Logging in JAX-RS	10-7
10.6 A Secured JAX-RS Web Service	10-8
10.7 Securing Web Service URLs	10-10
10.8 Configuring Login in web.xml	10-10
10.9 Configuring sun-web.xml	10-11
10.10 Retrieving Security Information in a Servlet	10-12
10.11 Dependency Injection in JAXRS	10-13
10.12 Logging Callers in JAX-RS	10-14
10.13 Simple Jersey Client	10-14
10.14 Authenticating Jersey Client	10-15
10.15 A POJO Web Service Using a DAO	10-16
10.16 A More Complex POJO Web Service	10-17
10.17 Creating Web Services from EJBs	10-18
10.18 A More Complex EJB Web Service	10-19
10.19 Singleton Web Service EJBs	10-21
10.20 Concurrency Policy for Singleton Web Service EJBs using JAX-RS	10-22
11.1 Returning a Supported Complex Type	11-5
11.2 Returning a Supported JAXB Application Type	11-6
11.3 Providing Location of New Item	11-7
11.4 Another Complex Return Type	11-8

LIST OF CODE EXAMPLESLIST OF CODE EXAMPLES

11.5 Using JAXB, the REST Way	11-10
11.6 Another Complex Return Type – The REST Way	11-10
11.7 Default Response in JAX-RS Resource	11-11
11.8 A Complex Type Not Supported by JAX-RS	11-13
11.9 A MessageBodyWriter for Maps	11-14
11.10A MessageBodyWriter for Maps – isWriteable	11-15
11.11A MessageBodyWriter for Maps – writeTo	11-15
11.12A JAX-RS Resource that Throws Exceptions	11-18
11.13An ExceptionMapper	11-18
11.14A Simple JAX-RS Resource Class	11-20
11.15A More Complex Resource Class	11-21
11.16Selecting A Sub-Resource	11-22
11.17The Sub-Resource Class	11-23
11.18A Request-Scope Resource	11-25
11.19Singletons and Request Context	11-27
13.1 Dispatch Interface	13-20
13.2 Provider Interface	13-20
13.3 Sample Client Using JAXWS Dispatch API	13-22
13.4 Server Using JAX-WS Messaging API	13-23
14.1 Implementation of the Service-Specific Exception	14-12
14.2 Example of a Service Throwing a Service-Specific Exception	14-13
14.3 Error Message Defined in WSDL Document	14-15
D.1 Simple POJO WS Client	D-2
D.2 Authenticating POJO WS Client	D-3
D.3 Interface javax.xml.ws.handler.Handler	D-6
D.4 Types of JAX-WS Handler	D-6

D.5 An Authentication Handler	D-8
---	-----

Oscar Hernando Nivia Aragon (onivia@hotmail.com) has a
non-transferable license to use this Student Guide.

About This Course

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

- Understand and explain 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.
- Understand how to deploy web services that also leverage other JavaTM 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

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

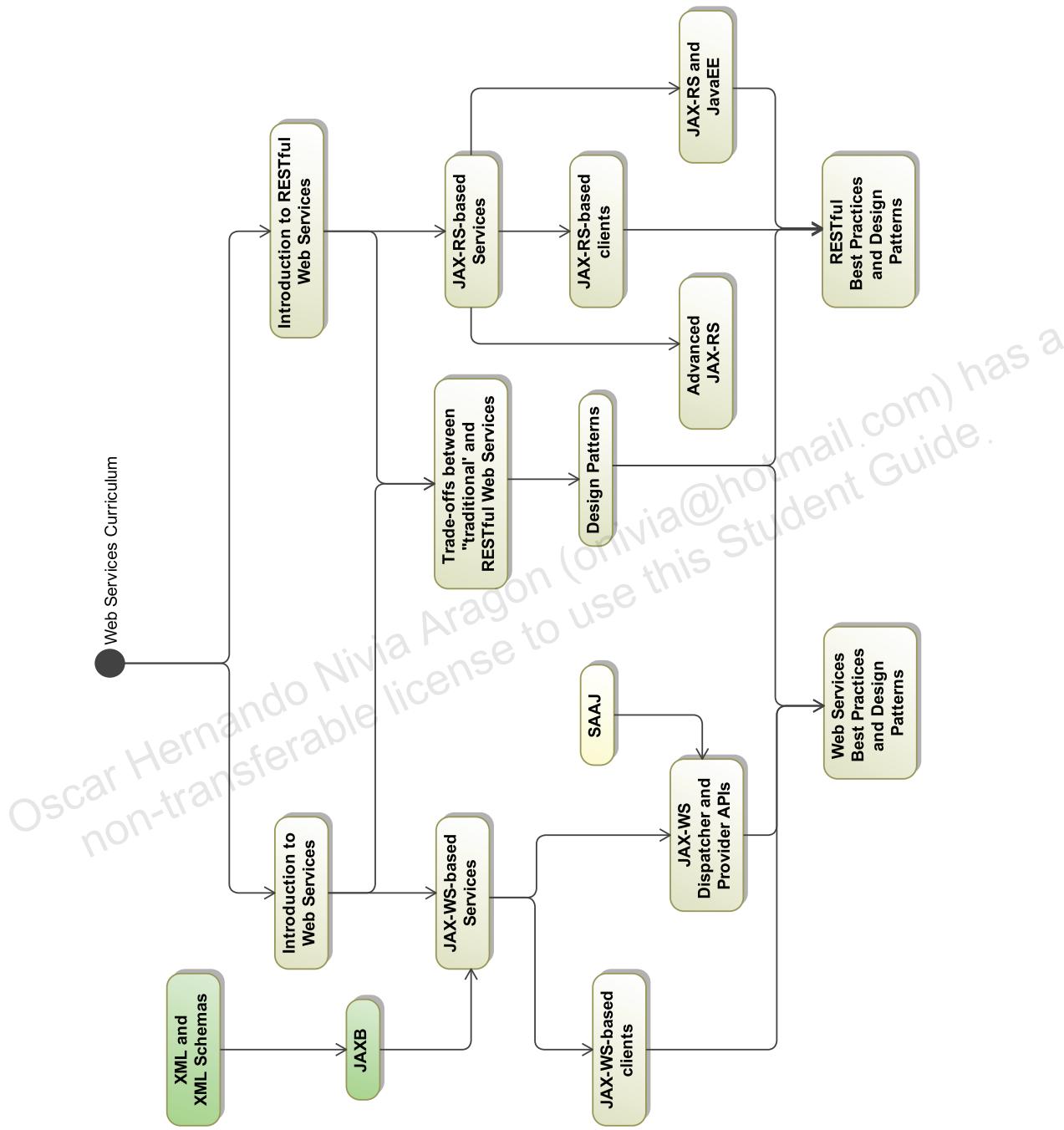


Figure 1: Course Map

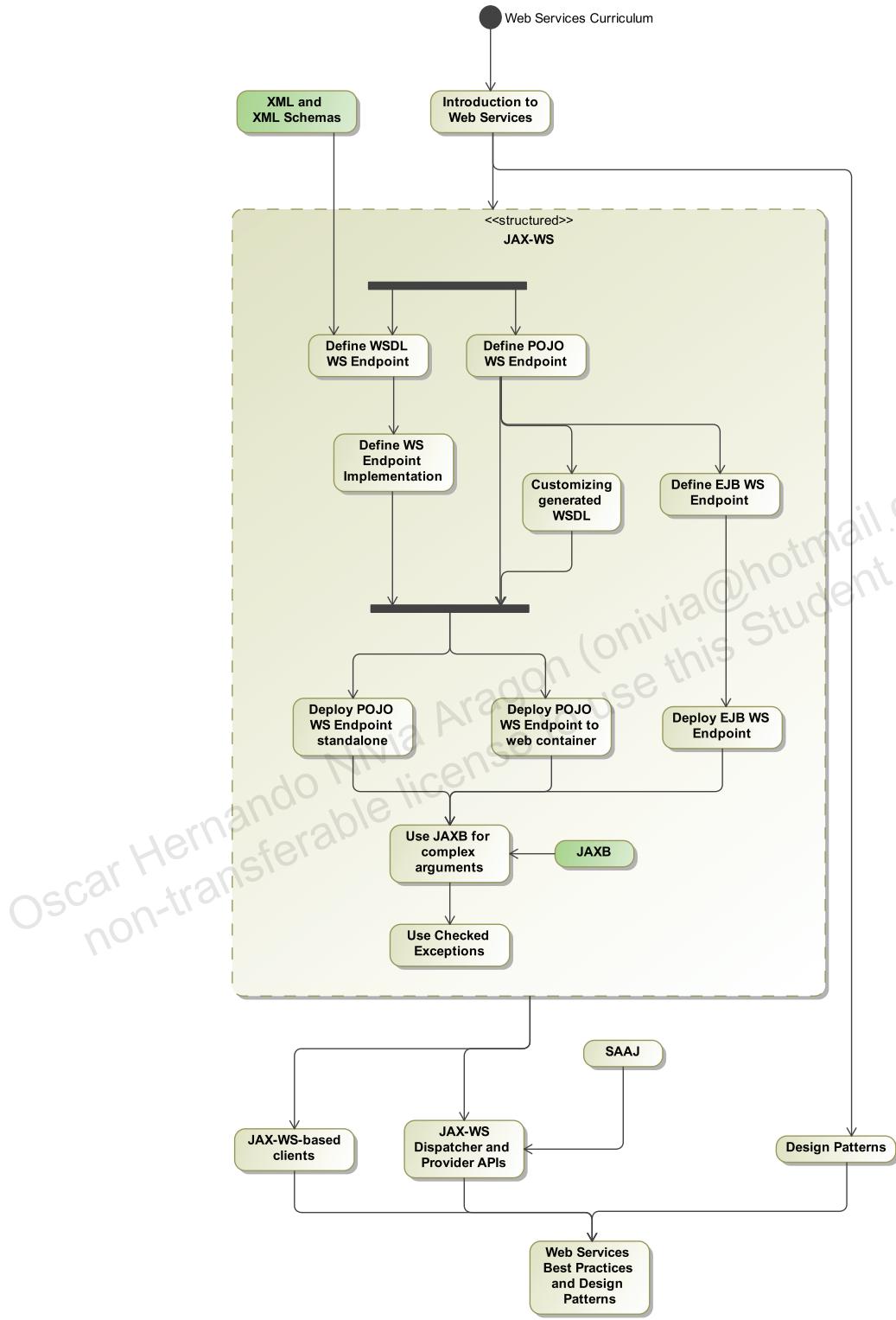


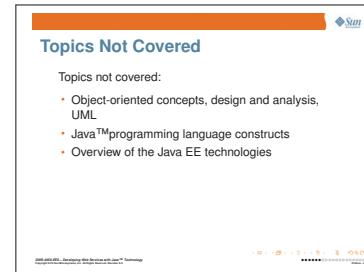
Figure 2: Course Map – JAXWS

Topics Not Covered

In order to concentrate on the topics directly relevant to this course, we make certain assumptions about other topics that students ought to be familiar with, before attending this course.

The list of these other topics includes:

- 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?

 Sun

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?

DWS-4050-EE6 - Developing Web Services with Java™ Technology
Copyright © 2010 Sun Microsystems, Inc. All rights reserved.



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

The screenshot shows a presentation slide with a blue header bar containing the Sun logo. The main title is 'Introductions'. Below it is a bulleted list of six items, which exactly matches the list provided in the adjacent text block. At the bottom right of the slide, there is a small set of navigation icons typical for a presentation slide.

How to Use Course Materials

To enable you to succeed in this course, these course materials contain a number of learning modules, which are designed to be equals parts lecture and discussion, on the one hand, and practical exercises to apply the concepts just learned. Each learning module is composed of the following components:

The screenshot shows a slide titled 'How to Use Course Materials' with the Sun logo in the top right corner. The slide content includes a brief introduction about the course's learning modules and their components, followed by two bulleted lists: 'Goals' and 'Objectives'. At the bottom of the slide are navigation icons for back, forward, and search.

- 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.
- 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.
- 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.

Conventions

The following conventions are used in this course to represent various training elements and alternative learning resources.



Self-check – Identifies self-check activities, such as matching and multiple-choice.



Additional resources - Indicates other references that provide additional information on the topics described in the module.



Discussion - Indicates a small-group or class discussion on the current topic is recommended at this time.



Note - Indicates additional information that can help students but is not crucial to their understanding of the concept being described. Students should be able to understand the concept or complete the task without this information. Examples of notational information include keyword shortcuts and minor system adjustments.



Code Listing – indicates that the complete source code listing associated with a code example is available by following this link.

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.

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.

Java Web Services • Developing Web Services with Java™ Technology
Copyright 2010 Sun Microsystems, Inc. All rights reserved. DWS-4050-EE6 Rev. A.0

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

- The `getServletInfo` method gets author information.

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.

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:

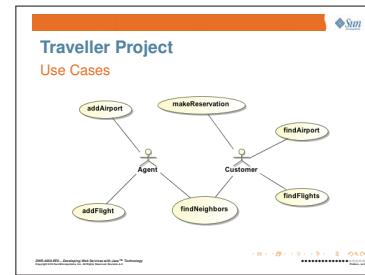
- To delete a file, use the `rm filename` command.

Java Web Services • Developing Web Services with Java™ Technology
Copyright 2010 Sun Microsystems, Inc. All rights reserved. DWS-4050-EE6 Rev. A.0

Course Projects

The Traveller Project

The Traveller application is the project from which the course draws all of its examples. The Traveller application supports a set of use cases that would necessary to provide an online airline reservation web site, as illustrated by Figure 3.



addAirport Adds a new airport to the system.

findAirport Finds an airport known to the system, by airport code, or by airport name.

findNeighbors Find all airports that can be reached from a given airport in one flight.

findFlights Find all (direct) flights between two airports.

makeReservation Make a reservation for a customer on a flight. In this system, a round-trip flight requires two reservations, one for each one-way trip.

addFlight Add a flight between the two airports specified to the system.

The uses cases listed above are just a sample of the uses cases that such an application should support – but they are enough to describe all the examples used throughout the course.



Figure 3: Use Cases for Traveller Project

Figure 4 is a sketch of the domain types used within the Traveller project to capture the information necessary to implement the uses cases described in Figure 3. In the design used by the course, all of these types are persistent – and all of these are implemented as JPA entity classes, to minimize the effort required to manage their persistence.

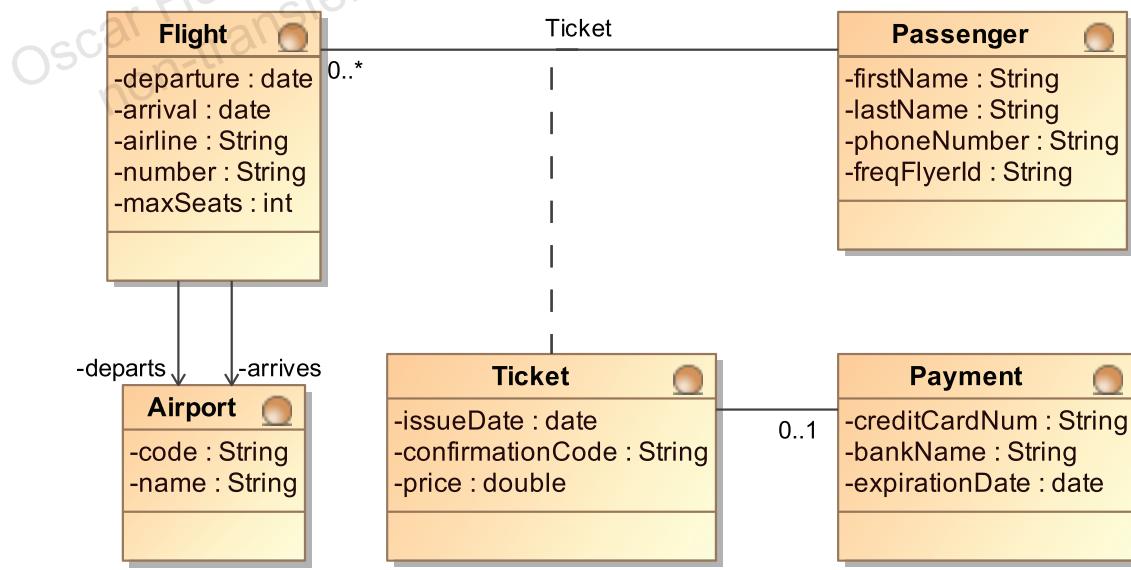
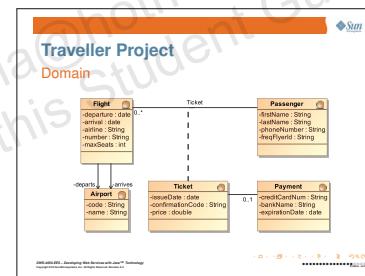


Figure 4: Domain Types for Traveller Project

Figure 5 presents a UML diagram of a number of candidate service types as *façades* to the business logic required by the application. (An SOA architecture for this application would probably further abstract some of the details exposed by the API presented in this diagram).

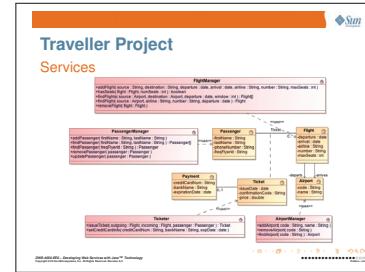
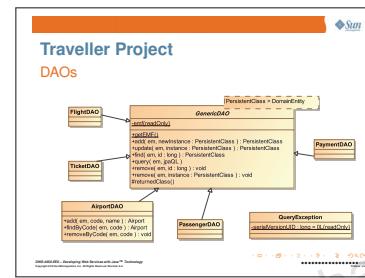


Figure 6 presents a UML diagram of a number of *Data Access Object* classes, which try to hide the JPA machinery that is used to manage the persistent instances used by the application. In the implementation offered for the examples, these DAO classes will be implemented two ways: as simple Java classes, and as Enterprise Java beans. The first implementation is less successful than the second one at hiding the JPA machinery: absent EJBs, the application logic must manage persistence contexts and transactions explicitly – and this may need to be exposed all the way to business methods...



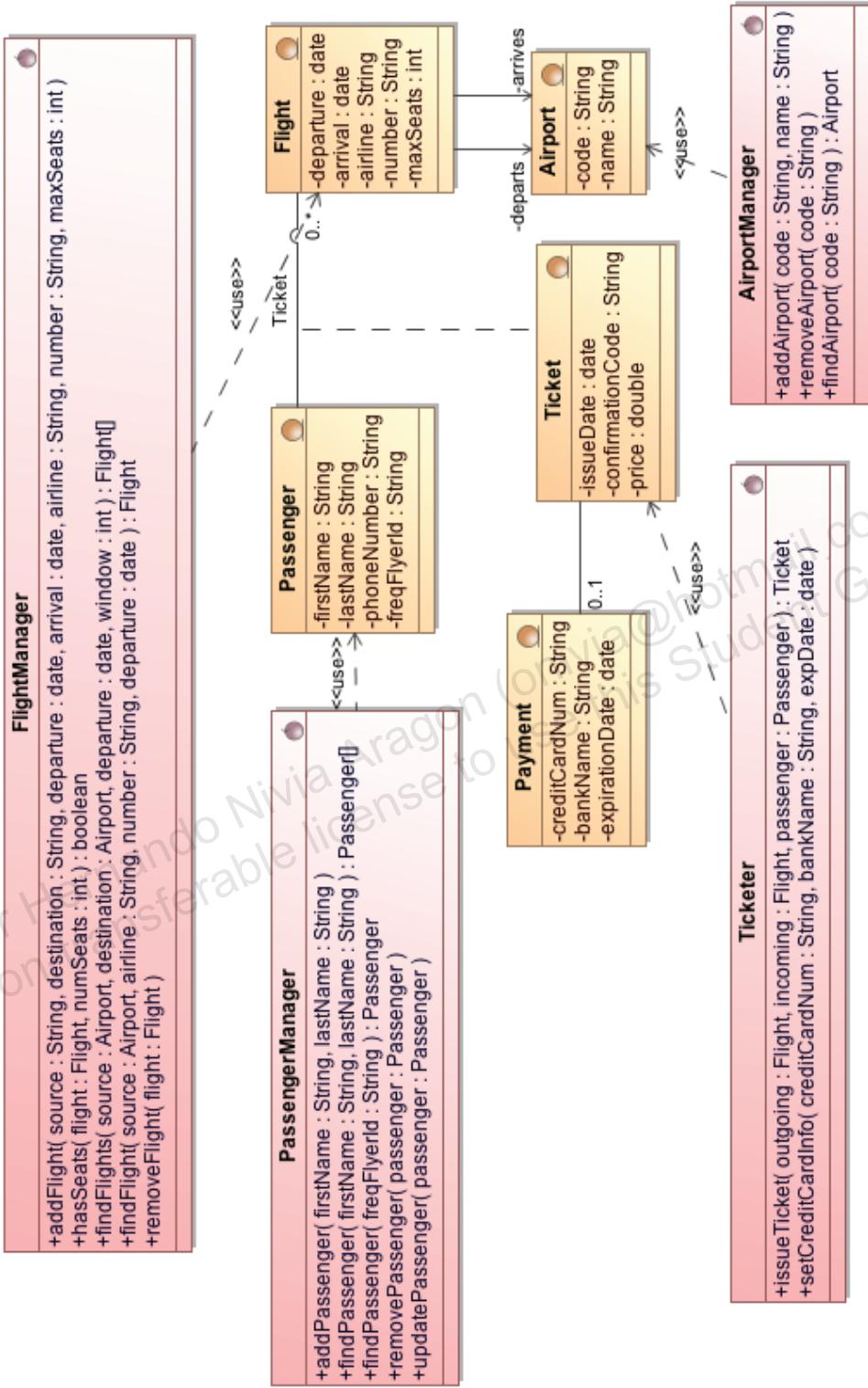


Figure 5: Application Services for the Traveller Project

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

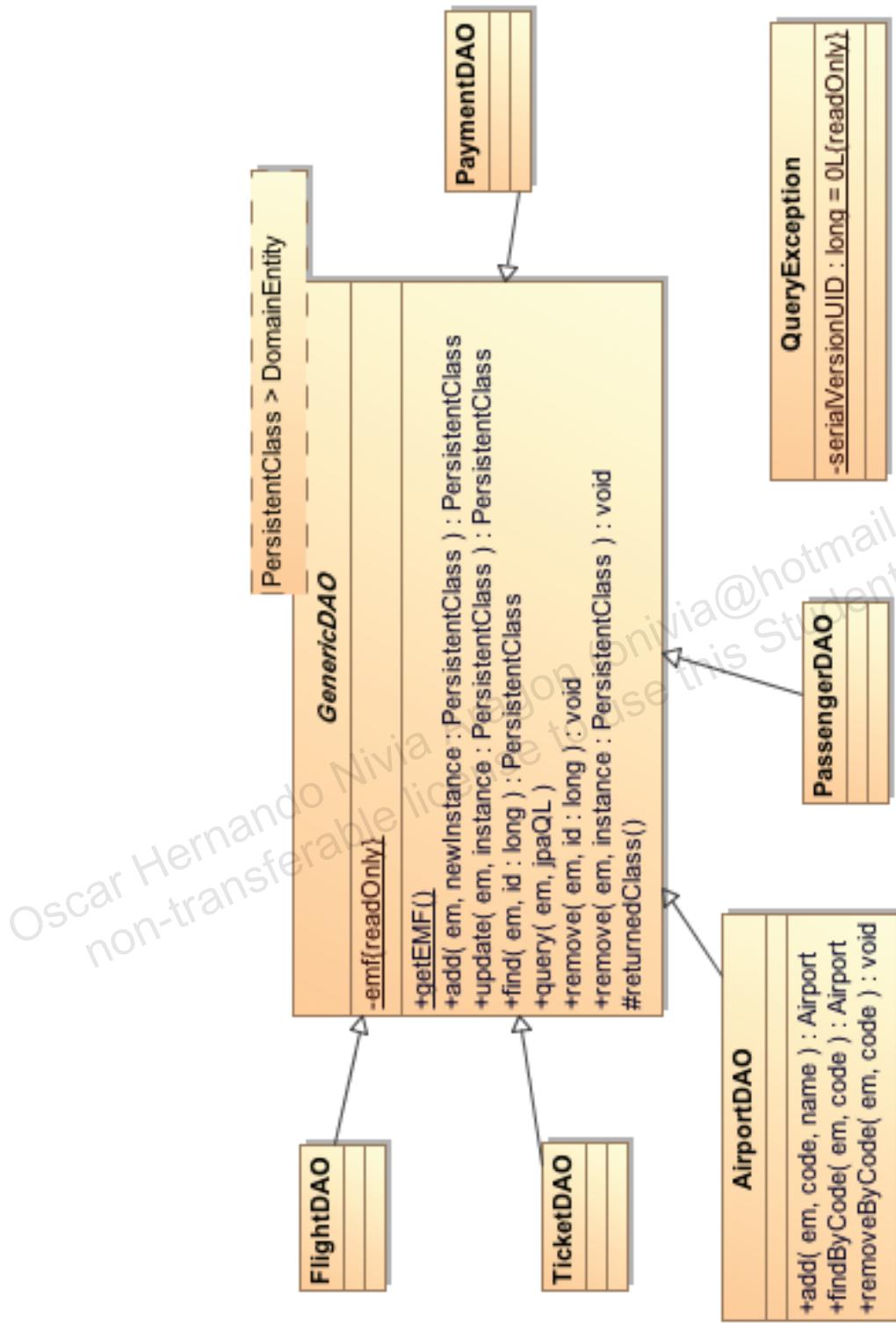


Figure 6: DAO Classes for the Traveller Project

The Auction Project

The Auction application supports the set of use cases necessary to provide an online auction web site, as illustrated by Figure 7:

login All users of the Auction system must first log on to the system, before they can invoke other operations. To log on, all users must authenticate themselves (perhaps by providing a userid and a password). This use case includes either of the **addUser** or **findUser** use cases.

It might be important to distinguish between sellers and bidders, since there are tasks in the Auction system that only one kind of user may request.

addUser Adds a new user to the system database. This use case is invoked by the **login** use case, when a new user accesses the Auction system for the first time. In addition to the user ID and password already supplied, the system asks the user for additional personal information: name, address (both location and email). Optionally, the user will be able to provide credit card information to be used as a default method of payment.

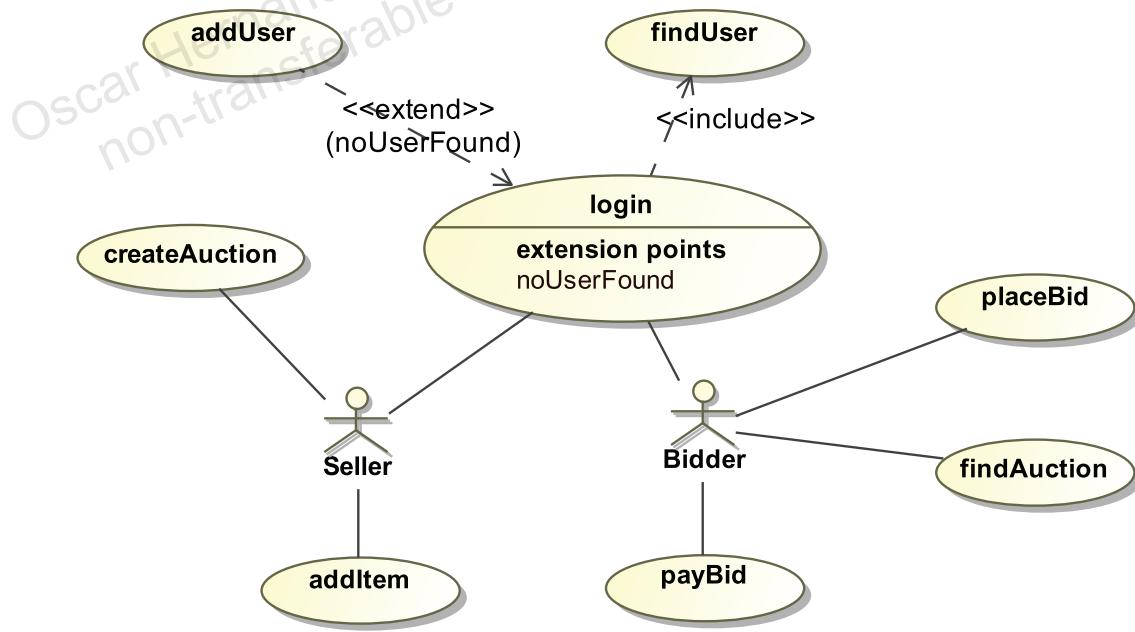
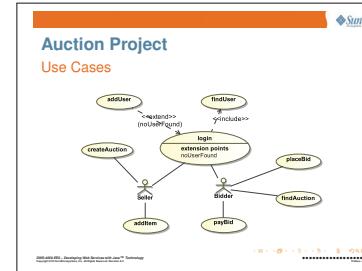


Figure 7: Use Cases for Auction System

`findUser` Finds an existing user in the system database. This use case is invoked by the `login` use case, when an existing user returns to the Auction system.

`addItem` Sellers are allowed to create items to be auctioned off. The item to be auctioned and the auction for the item are considered to be independent from each other. This allows a seller who has multiples of a given item to hold separate auctions for each copy, while reusing the same item for each auction. Items hold information, such as the name for the item, a description, and a picture.

`createAuction` Sellers are allowed to create auctions, to auction items off. Each auction holds information, such as the item being auctioned, the initial bid price, the date the auction starts and the date it will end (or how long the auction will last), and a list of all bids placed on it.

The Auction system should also allow the seller to create multiple auctions for a number of copies of the same item conveniently. This can be achieved by allowing the seller to create a first auction, and then allowing the seller to create “another just like it”, or by allowing the seller to specify the number of copies of an auction to create, as part of the process of creating each auction.

`placeBid` Bidders are allowed to bid on auctions. Given an auction, the bidder is allowed to enter a bid on that auction. The Auction system should not allow a bidder to place a bid on an auction that is lower than the current bid price for that auction. The Auction system should also allow the user to find out what the current bid price for the auction is, or even notify the user automatically when the current bid price for the auction changes, or when the bidder is no longer the current high bidder for the auction.

The bidder should be able to select a collection of auctions they are interested in bidding on, and to allow the bidder to place bids on any of the auctions in that collection. In this case, the Auction system should present the bidder with the current bid price for each of the auctions in the collection, plus a running total of the amount of money the bidder is currently bidding on all auctions in the list, taken together. Also, bidders should be able to find out what the current bid prices for all auctions in the list are, or be notified automatically when the current prices change, or when the bidder is no longer the high bidder for any of the auctions in the collection.

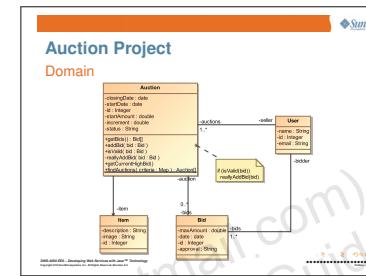
`findAuction` Users must be able to provide criteria to be used to retrieve a collection of bids in which the user might be interested. The user must also be able to narrow down the results of a previous query.

These query operations must be supported while other users are using

these same auctions. That is, users must be allowed to find auctions of interest while other users do the same, or bid on some of the same auctions.

payBid Bidders must be able to pay for their winning bid on an auction, once the action is over. The Auction system offers to use the credit card information the user entered when the user first registered with the system, although it also allows the user to provide an alternative means of payment. The system should also notify the seller of that auction, when the winning bidder actually pays for that auction.

The business model for the Auction system represents all the information necessary to support the use cases listed previously. Figure 8 illustrates the following abstractions, as a starting point to build the business tier for this application:



Auction Each instance represents an auction. It is responsible for the dates the auction starts and ends, the initial and winning bid price, and a list of all bids placed on this auction.

Item Each instance represents an item that can be auctioned off. It is responsible for information about each item, such as its name, description, or a picture of the item.

Bid Each instance represents a bid that a user has placed on an auction. It is responsible for information about the bid, such as the maximum amount the bidder was willing to bid on this bid, the time the bid was placed, and the bidder who placed the bid.

User Each instance represents a user of the Auction system. It is responsible for information about each user, such as the user's name, address, email, or credit card information.

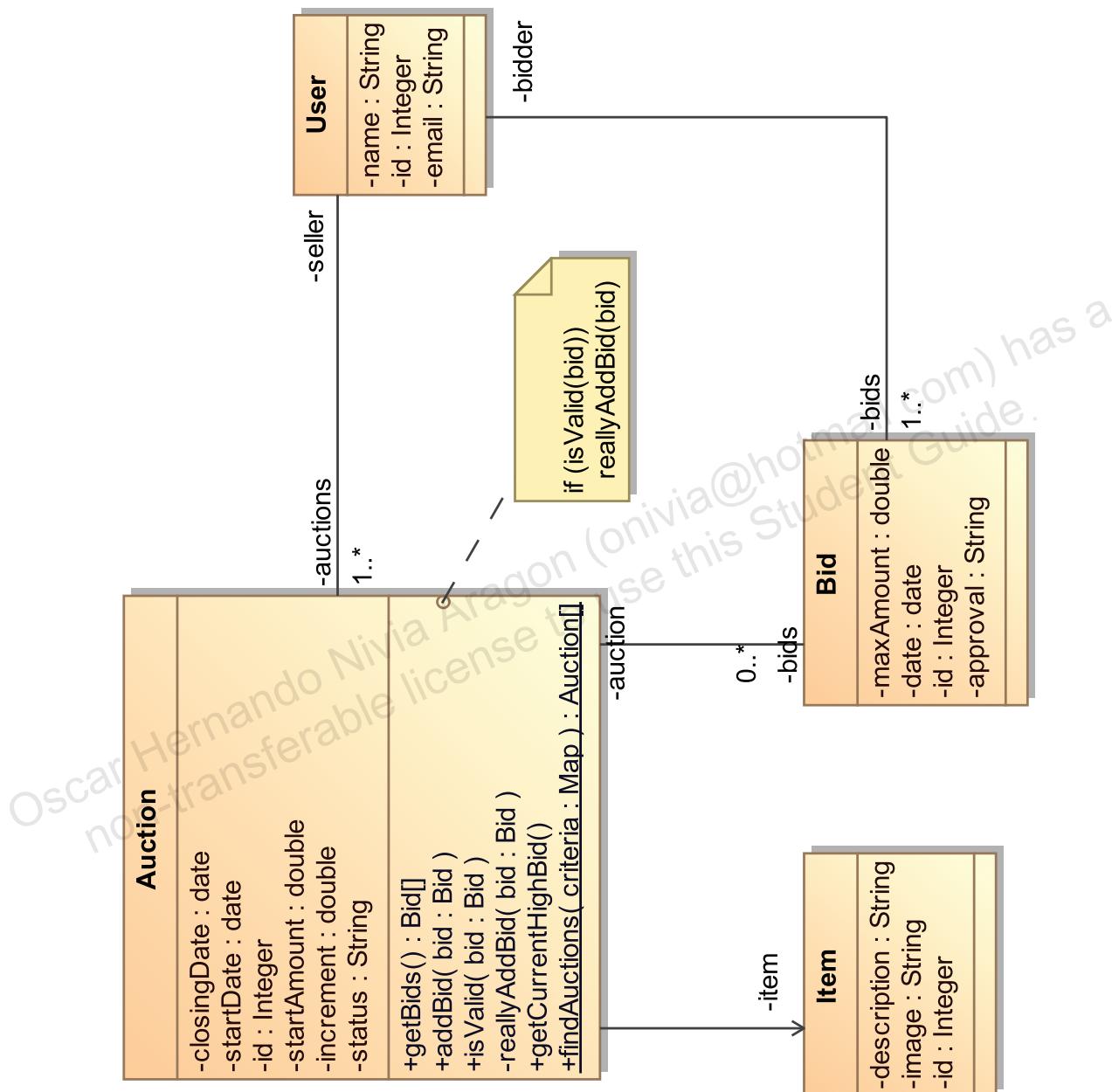


Figure 8: Domain Types for Auction System

Oscar Hernando Nivia Aragon (onivia@hotmail.com) has a
non-transferable license to use this Student Guide.

Chapter 11

Implementing More Complex Services Using JAX-RS and Jersey

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

The screenshot shows a slide with a blue header bar containing the word 'Objectives'. Below the header, there is a list of bullet points under the heading 'On completion of this module, you should:'. The list includes:

- 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

At the bottom of the slide, there is some small text and a set of navigation icons.

Additional Resources

Additional Resources

The following references provide additional information on the topics described in this module:

- Chapter 13, "Building RESTful Web Services with JAX-RS and Jersey", of the "JavaEE 6 Tutorial", at
<http://java.sun.com/javaee/6/docs/tutorial/doc/giepu.html>.
- JSR-311, JAX-RS: JavaTM API for RESTful Web Services specification, at:
<https://jsr311.dev.java.net/nonav/releases/1.1/spec/spec.html>

Parameters and Return Values

- 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.

 Sun

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.

Java Platform, Standard Edition 6 Technology
Java Platform, Enterprise Edition 6 Technology
... 1 2 3 4 5 6 7 8 9 10 11

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.

 Sun

Simple Types as Parameters

Valid parameter types in general include:

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

Java Platform, Standard Edition 6 Technology
Java Platform, Enterprise Edition 6 Technology
... 1 2 3 4 5 6 7 8 9 10 11

Resource class fields and bean properties can be annotated with the JAX-RS parameter annotations. Since injection occurs at object creation time, such annotated fields and properties will be initialized when the instance is initialized, at request processing time.



This ability to initialize resource class fields and bean properties as parameters is only supported for the default per-request resource class lifecycle.

`@QueryParam` and `@PathParam` can only be used on the following Java types:

Parameters and Return Values

- All primitive types except char.
- All wrapper classes of primitive types except Character.
- Those that have a constructor that accepts a single String argument.
- Any class with the static method named `valueOf(String)` that accepts a single String argument .
- Any class with a constructor that takes a single String as a parameter
- `List<T>`, `Set<T>`, or `SortedSet<T>`, where T matches the already listed criteria. Sometimes parameters may contain more than one value for the same name. If this is the case, these types may be used to obtain all values

The `DefaultValue` annotation may be used to supply a default value for parameters – see the Javadoc for `DefaultValue` for usage details and rules for generating a value in the absence of this annotation and the requested data. The `Encoded` annotation may be used to disable automatic URI decoding for `@MatrixParam`, `@QueryParam`, and `@PathParam` annotated fields and properties.

Cookie parameters (indicated by decorating them with `@CookieParam`) extract information from the cookies declared in cookie-related HTTP headers. Header parameters (indicated by decorating them with `@HeaderParam`) extract information from the HTTP headers. Matrix parameters (indicated by decorating them with `@MatrixParam`) extract information from URL path segments.

Form parameters (indicated by decorating them with `@FormParam`) extract information from a request representation that is of the MIME media type `application/x-www-form-urlencoded` and conform to the encoding specified by HTML forms. This parameter is very useful for extracting information that is POSTed by HTML forms.

Valid return types in general include:

- `void`
- `byte[]` and `String`
- Application-supplied JAXB classes and `javax.xml.bind.JAXBElement`
- `MultivaluedMap<String, String>`

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.

```

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

E-87



Code 11.1: Returning a Supported Complex Type

These return values result in an entity body mapped from the class of the returned instance. If the return value is not null a 200 status code is used; a null return value (or void) results in a 204 status code.

Additionally, return types can be one of Response and GenericEntity. These allow the application to control the HTTP return code and payload of the return message:

- The entity body is mapped from the entity property of the Response.
- The status code is specified by the status property of the Response. A null return value results in a 204 status code. If the status property of the Response is not set: a 200 status code is used for a non-null entity property and a 204 status code is used if the entity property is null.

Code 11.1 shows an example of a JAX-RS resource that returns an instance of an application domain type. Line 6 indicates the representations that the service will be willing to offer; the return type Airport in line 7 is a JAXB type – so Jersey can manage the marshalling of such instances into XML and JSON on its own.

Returning a Supported Complex Type

```

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

... 1 2 3 4 5 6 7 8 9 10 11 12

Parameters and Return Values

```

1 @Entity
2 @XmlRootElement
3 public class Airport
4     implements Serializable {
5     @Basic(optional=false) @Column(updatable=false, unique=true)
6     private String code;
7     private String name;
```

E-67

Code 11.2: Returning a Supported JAXB Application Type

The **@Produces** annotation is used to specify the MIME media types or representations a resource can produce and send back to the client. If **@Produces** is applied at the class level, all the methods in a resource can produce the specified MIME types by default. If it is applied at the method level, it overrides any **@Produces** annotations applied at the class level. The value of **@Produces** is an array of `String` of MIME types, for the representations to be supported.

If no methods in a resource are able to produce the MIME type in a client request, the Jersey runtime sends back an HTTP 406 Not Acceptable error.

If a resource class is capable of producing more than one MIME media type, the resource method chosen will correspond to the most acceptable media type as declared by the client.

Jersey can produce XML and JSON representations for any application domain type, as long as it is JAXB-compliant. Code 11.2 is an example of an application type, `Airport`, that is JAXB-compliant. Note that JAXB types are intended to be just POJOs, with a few annotations to indicate that, and how, JAXB is to manage marshalling and unmarshalling of such instances.

Methods that need to provide additional metadata with a response should return an instance of `Response`.

The `ResponseBuilder` class is used to build `Response` instances that contain metadata instead of or in addition to an entity. An initial instance may be obtained via static methods of the `Response` class; instance methods provide the ability to set metadata.

`ResponseBuilder` provides a convenient way to create a `Response` instance using a builder pattern:

The screenshot shows a slide titled "Returning a Supported Complex Type" under the "JAXB Application Type" heading. It contains a code snippet for a Java class named "Airport" with annotations for Entity and XML root element. The code includes fields for code and name, and a constructor that initializes them. The slide has a navigation bar at the bottom.

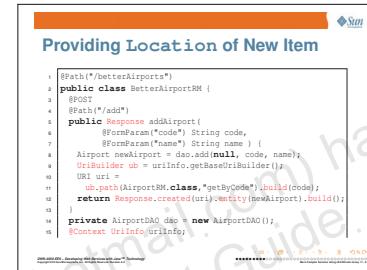
The screenshot shows a slide titled "Controlling Responses". It lists several points about Response instances, including the implementation of the Builder pattern and the use of UriInfo and UriBuilder for building URLs. The slide has a navigation bar at the bottom.



- 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.

Code 11.3 shows a typical use case for the two types Response and ResponseBuilder: the web service shown wants to return the newly created instance as the result of the operation in line 12 – but it is not enough to just return that instance. To obey HTTP semantics, it is also necessary to indicate to the client that a new instance was indeed created by this request, and how to refer to it from that point forward. The `created()` method in ResponseBuilder takes care of both: the call in line 12 indicates that the status code must indicate the creation of a new item, and the parameter to the call will be provided as the Location of that newly-created item.

UriBuilder is an URI template aware utility class for building URIs from their components.



```

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;

```

E-118



Code 11.3: Providing Location of New Item

Linking To Other Resources

Linking To Other Resources

In principle, a JAX-RS web service can return any type of object that is JAXB-compliant. Figure 11.4 shows another version of the Airport web service, BetterAirportRM. This version supports the ability to produce, on demand, a list of all known airports. The version shown in Figure 11.4, however, returns a list of “complete” instances of Airport (see line 6), each containing all the information available to the server about each airport. This will work – but it is not the REST way.

Figure 11.1 shows what the client might receive as a return value, if they were to ask for a list of all airports. As it is the complete representation of each Airport, it is arguably *too much* information.

The REST way involves the principle of “progressive disclosure” – clients should receive only a summary, first; they should have to “drill down” if more detail is required. Figure 11.2 illustrates what the answer to the initial request could look like, to obey this principle.

```

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();

```

- 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>

```

```

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();

```

E-118

Code 11.4: Another Complex Return Type



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

Figure 11.1: Returning Complex Values via XML

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

Figure 11.2: Returning Complex Values the REST Way

Unfortunately, support for customizing the marshalling of JAXB instances on a per-operation basis is not very sophisticated within JAXB – so there is no convenient way to customize what the representation of an Airport will be, on a per-operation basis. Code 11.5 shows an alternative approach to generate a custom representation of an Airport: since there is no way to customize the representation of the “generic” Airport, the example opts for creating a *custom* Airport, and thus control what the representation of the Airports for this operation will be. The Airport class shown in Code 11.5 is constructed from a “generic” `listinline-Airport`, but results in the XML structure illustrated in Figure 11.2.

Code 11.6 shows how the `BetterAirportRM` resource shown before would have been modified, in order to produce the XML response that obeys “progressive disclosure” through use of the custom `Airport` class. Note how the service operation must create instances of the custom `Airport` class and return a list of them, to control the format of the XML structure that will be produced.

```
1 package com.example.user.resources.helpers;
2 import javax.xml.bind.Unmarshaller;
3 import javax.ws.rs.core.UriBuilder;
4 import java.util.List;
5 import java.util.Map;
6 import org.json.simple.JSONObject;
7 import org.json.simple.parser.JSONParser;
8 import org.json.simple.parser.ParseException;
9
10 public class UsingJAXB {
11     public static void main(String[] args) throws ParseException {
12         String json = "[{" +
13             "code": "MCO", "name": "Orlando" +
14         "}]";
15         JSONParser parser = new JSONParser();
16         JSONObject jsonObject = (JSONObject) parser.parse(json);
17         Map map = (Map) jsonObject.get("list");
18         List<Airport> list = (List<Airport>) map.get("list");
19         Unmarshaller unmarshaller = Unmarshaller.unmarshal(list);
20         UriBuilder ub = UriBuilder.fromUri("http://localhost:8080/jaxrs/resources/airports");
21         System.out.println(unmarshaller.build(ub));
22     }
23 }
```

```
1 package com.example.user.resources;
2 import javax.ws.rs.core.UriBuilder;
3 import java.util.List;
4
5 @Path("airports")
6 public class BetterAirportRM {
7     @GET
8     @Path("{code}")
9     @Produces({"application/xml", "application/json"})
10     public List<Airport> getByCode(@PathParam("code") String code) {
11         UriBuilder ub = UriBuilder.fromUri("http://localhost:8080/jaxrs/resources/airports");
12         ub.path("byCode");
13         ub.queryParam("code", code);
14         List<com.example.jaxrs.resources.helpers.Airport> result =
15             UriBuilder.fromUri("http://localhost:8080/jaxrs/resources/airports").queryParam("code", code).build();
16         for (com.example.jaxrs.resources.helpers.Airport proxy : result) {
17             proxy.setUriBuilder(ub);
18         }
19         return result;
20     }
21 }
```

You can think of this custom `Airport` class as an application of the *Transfer Object* pattern, to control the representation of the information that the client will receive.



Linking To Other Resources

```

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}

```

E-122

Code 11.5: Using JAXB, 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    }

```

E-118

Code 11.6: Another Complex Return Type – The REST Way



```

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         }

```

E-118



Code 11.7: Default Response in JAX-RS Resource

It is possible to introduce a method that constitutes a *default response*. Given the service implementation shown in Code 11.7, a GET request with path /betterAirports will result in the JAX-RS runtime dispatching the operation to the getDefault() method in line 5: dispatchable methods in the resource class need not have each their own @Path declaration – it is enough that the runtime can unambiguously select a method for the URL path that it has to process.

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         }

```

Custom Marshalling and Unmarshalling

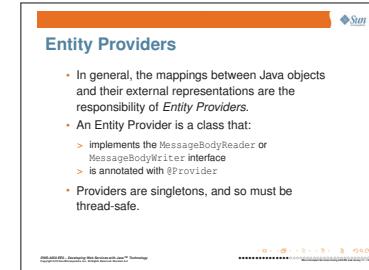
Entity providers supply mapping services between representations and their associated Java types.

There are two types of entity providers:

MessageBodyReader and MessageBodyWriter. For HTTP requests, the MessageBodyReader is used to map an HTTP request entity body to method parameters. On the response side, a return value is mapped to an HTTP response entity body using a MessageBodyWriter. If the application needs to supply additional metadata, such as HTTP headers or a different status code, a method can return a Response that wraps the entity, and which can be built using ResponseBuilder.

Entity providers are annotated with @Provider. They are singletons, and so must be thread-safe.

The following list contains the standard types that are supported automatically for entities. You only need to write an entity provider if you are not choosing one of the standard types shown in Table 11.1.



Java Type	Media Types	MIME Type
byte[]	All media types	*/*
java.lang.String	All text media types	text/*
java.io.InputStream	All media types	*/*
java.io.Reader	All media types	*/*
java.io.File	All media types	*/*
javax.activation.DataSource	All media types	*/*
javax.xml.transform.Source	XML types	text/xml, application/xml, and application/*+xml
javax.xml.bind.JAXBElement and application-supplied JAXB classes	XML media types	text/xml, application/xml, and application/*+xml
MultivaluedMap<String, String>	Form content	application/x-www-form-urlencoded
StreamingOutput	All media types	*/* (MessageBodyWriter only)

Table 11.1: Standard Entity Types in JAX-RS

```

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         }

```

E-123



Code 11.8: A Complex Type Not Supported by JAX-RS

Code 11.8 shows a slightly more complex resource, PlannerRM. This service supports the ability to produce a descriptions of all the flights departing from a given airport, along with each flight's destination. The return type used to capture this information is a map from flight numbers (strings) to airport codes (also strings). The problem to solve is that JAX-RS does not support by default the marshalling of Map instances into XML.

Instances of MessageBodyWriter can be used to extend the set of types that the JAX-RS runtime can marshall into XML, and MessageBodyReader instances to extend the set of types that can be unmarshalled. A MessageBodyWriter has to implement:

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

- `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.

- `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`

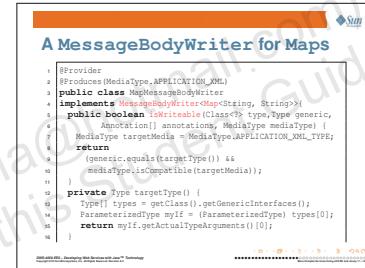
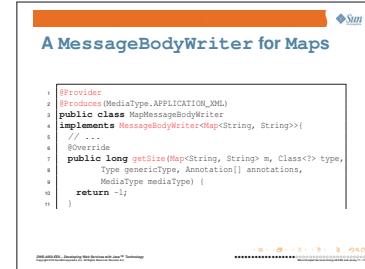
Custom Marshalling and Unmarshalling

Called to actually convert the value to be returned to its appropriate representation by way of this writer.

MessageBodyReaders need equivalent functions.

Code 11.9 shows what the getSize method for a MessageBodyWriter for Map might look like. In principle, this method must return the marshalled length of the representation of a Map. However, it can also return -1 if the resulting length is yet unknown – which might be preferable to have to compute the marshalled representation of the Map just to figure out its length.

Code 11.10 shows what the isWriteable method for a MessageBodyWriter for Map might look like. In principle, this method must return true if this MessageBodyWriter can accept responsibility for marshalling the parameter it's offered – which boils down to verifying that said parameter is of a type consistent with the type of object that this MessageBodyWriter was designed to handle.



```

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    }

```

E-126

Code 11.9: 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      }

```

E-126



Code 11.10: A MessageBodyWriter for Maps – isWriteable

```

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  }

```

E-126



Code 11.11: A MessageBodyWriter for Maps – writeTo

Custom Marshalling and Unmarshalling

Code 11.11 shows what the `writeTo` method for a `MessageBodyWriter` for `Map` might look like. In principle, this method must marshall the parameter it's offered into the representation that this `MessageBodyWriter` is responsible for. The implementation of the `writeTo` method is free to do this as it will.

```
A MessageBodyWriter for Maps
+-----+
| public void writeTo(MessageContext message, 
| Class<?> type, Type genericType,
| Annotation[] annotations, MediaType mediaType,
| MultivaluedMap<String, Object> httpHeaders,
| OutputStream entityStream)
| throws IOException, WebApplicationException {
|     StringWriter sb = new StringWriter();
|     sb.append("<map>"); // writer starts with <map>
|     for (Map.Entry<String, String> entry : m.entrySet()) {
|         String key = entry.getKey(), val = entry.getValue();
|         sb.append("<entry>"); // start of entry
|         sb.append("<key>").append(key).append("</key>"); // key
|         sb.append("<value>").append(val).append("</value>"); // value
|         sb.append("</entry>"); // end of entry
|     }
|     sb.append("</map>"); // writer ends with </map>
|     entityStream.write(sb.toString().getBytes());
| }
+-----+
```

The implementation of the `MessageBodyWriter` shown here is a bit more complex than it might need to be. In particular, `isWriteable` in this example is written to use reflection on itself to figure out whether the parameter it's offered is compatible with this writer's intent. But it could more easily have hardcoded the knowledge needed, rather than using reflection.



Exception Handling in JAX-RS

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.

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.

Code 11.12 shows the `add` operation on a JAX-RS `BetterAirportRM` resource. Although never mentioned before, it is possible for the `add` operation on the data access object used by this resource (line 8) to throw an exception – for example, if the database were down, or if the application attempted to add a duplicate airport.

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 = urlInfo.getBaseUriBuilder();
10         Uri uri = ub.path(AirportRM.class, "getByCode").build(code);
11         return Response.created(uri).entity(newAirport).build();
12     }
13 }

```

• DAO methods may throw `PersistenceException`

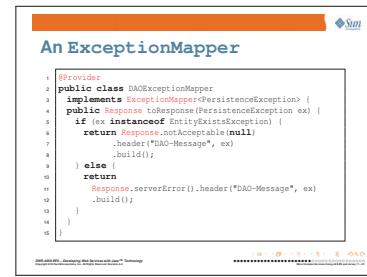
The specification of a `Response` return type from the operation (line 5) allows the application to explicitly handle the representation to return to the client, if it so desired. In this example, however, no steps are taken to handle exceptions within this operation.

An application can supply exception mapping providers to customize the mapping between exceptions and the HTTP response to deliver back to the client. Exception mapping providers map a checked or runtime exception to an instance of `Response`.

An exception mapping provider `ExceptionMapper<T>` and is annotated with `@Provider`. When a resource method throws an exception for which there is an exception mapping provider, the matching provider is used to obtain a `Response` instance. The resulting `Response` is processed as if the method throwing the exception had instead returned the `Response` normally.

Exception Handling in JAX-RS

Code 11.13 shows an example of an exception mapper for PersistenceException – the type of exception that the AirportDAO data access object could throw. The Response instance created by this mapper captures information about the exception, including the proper HTTP status code, to be returned back to the caller.



```

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    }

```

E-118

Code 11.12: A JAX-RS Resource that Throws Exceptions

```

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

```

E-128

Code 11.13: An ExceptionMapper



Using Resources and Sub-resources

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. This allows web service implementations to honor good object-oriented practices: “maximize cohesion, minimize coupling”.

Figure 11.3 recaps the domain model for the Traveller application.

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.

2010-EE6-003 - Developing Web Services with Java™ Technology
Copyright © 2010, Sun Microsystems, Inc. All rights reserved. DWS-4050-EE6 Rev. A.0

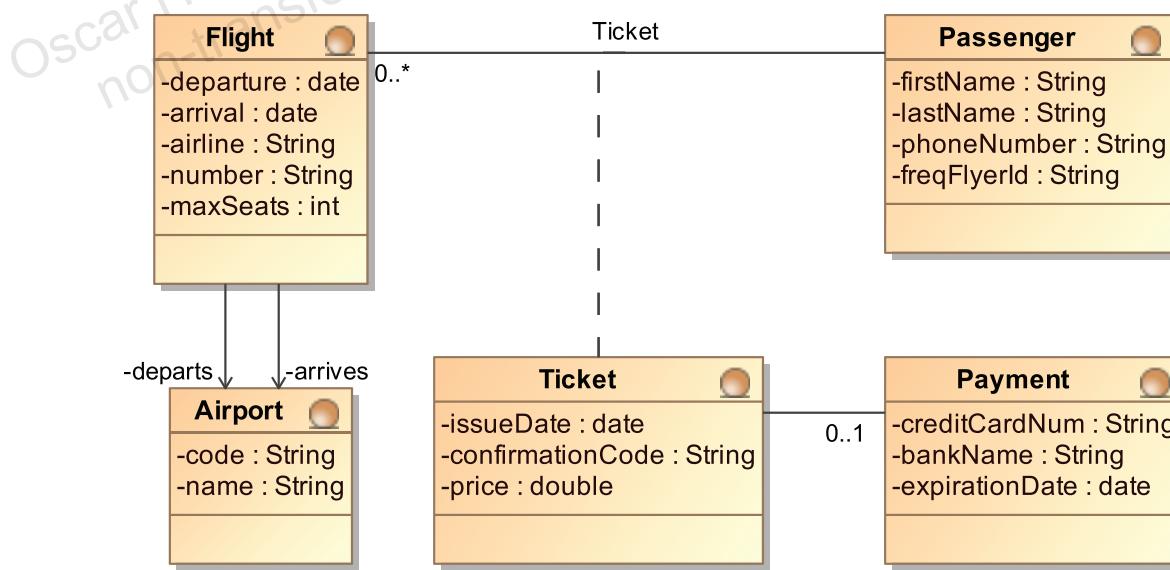
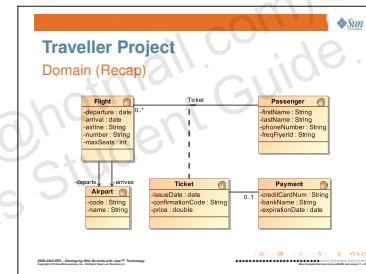


Figure 11.3: Traveller Project Domain Recap

Using Resources and Sub-resources

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*.

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.

Not every resource class can be implemented so.

Code 11.14 shows a simple JAX-RS root resource class. It can process two different requests itself.

A Simple Resource Class

```

@State("airports")
public class AirportRM {
    @GET
    @Path("/airports")
    public String getNumAirports() {
        List<String> codes = dao.getAllCodes( null );
        return (codes == null) ? "0" : "" + codes.size();
    }
    @GET
    @Path("/byCode/{code}")
    @Produces({ "application/xml", "application/json" })
    public Airport getByCode(
        @PathParam("code") String code ) {
        return dao.findByCode( null, code );
    }
    private AirportDAO dao = new AirportDAO();
}
  
```

```

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 }
  
```

E-87

Code 11.14: A Simple JAX-RS 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 }

```

E-129



Code 11.15: A More Complex Resource Class

- 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 that 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.

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 root resource class is anchored in URI space using the **@Path** annotation. The value of the annotation is a relative URI path template whose base URI is provided by the combination of the deployment context and the application path (see the **@ApplicationPath** annotation). The code in listing Code 11.15 shows a root resource class associated with the URI "/flights".

However, the resource class in listing Code 11.15 returns a complete Flight instance. The JAX-RS way, on the other hand, suggests that clients ought to be able to navigate their way to the more detailed flight-related information they are interested in.

A More Complex Resource Class ...

```

@Path("/flights")
public class FlightRM {
    @GET @Path("/{byNumber}/{number}")
    @Produces({"application/xml", "application/json"})
    public Flight getByNumber(
        @PathParam("number") String number) {
        return dao.findByNumber(number);
    }
    private FlightDAO dao = new FlightDAO();
}

```

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

Using Resources and Sub-resources

```

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      }

```

E-129

Code 11.16: Selecting A Sub-Resource

Methods of a resource class that are annotated with **@Path** are either sub-resource methods or sub-resource locators.

Sub-resource methods handle a HTTP request directly; sub-resource locators return an object that will handle a HTTP request.

A sub-resource locator method is characterized by the lack of a request method annotation (annotations such as **@GET**). Such methods are used to dynamically resolve the object that will handle the request. Any returned object is treated as a resource class instance and used to either handle the request or to further resolve the object that will handle the request.

Code 11.16 shows a code listing for a root resource class – FlightRM – that delegates processing of a request to a different resource class – AirportResource – via the sub-resource locator method `getDepartsByNumber()`.

Code 11.17 shows the corresponding sub-resource class, `AirportResource`:

- The class itself is not annotated with a **@Path** annotation.
- The path annotations on its methods are resolved relative to the parent resource class that delegated to an instance of this class.

Figure 11.4 presents several valid queries that the root resource `FlightRM` (and its subresource `AirportResource`) could handle.

```

@Path("/flights")
public class FlightRM {
    @Path("/byNumber/{number}/departs")
    public AirportResource getDepartsByNumber(
        @PathParam("number") String number) {
        Flight flight = dao.findByNumber(number);
        return new AirportResource(flight.getDeparts());
    }
}

```

... and A Sub-Resource

- 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`

```

public class AirportResource {
    ...
    @GET
    @Produces("application/json")
    public com.example.jaxrs.resources.helpers.Airport getDefault() {
        UriBuilder ub = UriInfo.getUriBuilder();
        ub.path("airports");
        new com.example.jaxrs.resources.helpers.Airport(airport, ub);
    }

    @GET("code/{code}")
    public String getCode() {
        return airport.getCode();
    }

    @GET("name")
    public String getName() {
        return airport.getName();
    }

    private Airport airport;
    @Context UriInfo uriInfo
}

```

The Sub-Resource Class

```

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

/flight/byNumber/AA123/departs
<airport><code>MCO</code><name>Orlando</name></airport>

/flight/byNumber/AA123/departs/code

```

Valid Queries



```

1 public class AirportResource {
2     AirportResource(Airport airport) {
3         this.airport = airport;
4     }
5     @GET @Produces({"application/xml","application/json"})
6     public
7         com.example.jaxrs.resources.helpers.Airport getDefault() {
8             UriBuilder ub = uriInfo.getBaseUriBuilder();
9             return
10                 new com.example.jaxrs.resources.helpers.Airport(airport,ub);
11             }
12     @GET @Path("/code") public String getCode() {
13         return airport.getCode();
14     }
15     @GET @Path("name") public String getName() {
16         return airport.getName();
17     }
18     private Airport airport;
19     @Context UriInfo uriInfo;
20 }
```

E-131



Code 11.17: The Sub-Resource Class

- /flights/byNumber/AA123


```

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


```

<airport><code>MCO </code><name>Orlando </name></airport>
```
- /flights/byNumber/AA123/departs/code


```
MCO
```

Figure 11.4: Valid Queries

Resource Scopes

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.

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.

By default the life-cycle of root resource classes is per-request, namely that a new instance of a root resource class is created every time the request URI path matches the root resource. This makes for a very natural programming model where constructors and fields can be utilized without concern for multiple concurrent requests to the same resource:

- 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.

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!

In general this is unlikely to be a cause of performance issues. Class construction and garbage collection of JVMs has vastly improved over the years and many objects will be created and discarded to serve and process the HTTP request and return the HTTP response.

Instances of singleton root resource classes can be declared by an instance of **Application**.

```

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;

```

E-102



Code 11.18: A Request-SScoped Resource

By default, JAX-RS resources are request-scoped. The SecureAirport resource shown in Code 11.18 describes root resource instances that are request-scoped, since it has no scope annotations at all.

Jersey supports two further life-cycles using Jersey specific annotations. If a root resource class is annotated with **@Singleton** then only one instance is created per-web application. If a root resource class is annotated with **@PerSession** then one instance is created per web session and stored as a session attribute.

Applications can request that resource instances have *singleton* scope, in one of two ways:

- **Application** can return a list of singleton resources.
- Using a Jersey extension, resource classes may be annotated with **Singleton**

A Request-SScoped 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?

Resource Scopes

Characteristics of singleton resources include:

- A single instance of the class is allocated, at the time the application is initialized.
- The singleton instance must be thread-safe, since it will be shared by all requests.
- Field-level dependency injection is performed once, when the instance is allocated.
 - No request-related @Context entity may be injected as a field of a singleton resource.

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.

The screenshot shows a section titled "Injecting Parameters" from the Java API documentation. It discusses two points of dependency injection: object initialization and method invocation. For object initialization, it notes that annotated fields can be injected but their values are not request-specific. For method invocation, it notes that annotated parameters can be injected with request-specific entities. The code example shown is for a singleton resource named "Singleton".

Code 11.19 shows an example of a singleton root resource, given the **@Singleton** annotation on line 1. Note the two opportunities for injection illustrated by this example:

- Instance variables, such as `webContext` (in line 19), are injected at the time the instance of the resource is created.
- Parameters to a resource method can be injected, too. `secContext` (in line 9) is injected as an extra parameter, every time a request is dispatched to the `addAirport` method. This mechanism allows for request-specific context objects to be injected, when processing a request, even if the resource itself is longer-lived than that request.

The screenshot shows a code example for a singleton resource named "Singleton". The code includes annotations like `@Singleton`, `@Path("singleton")`, and `@POST`. It defines a method `addAirport` that takes a `Airport` object and a `SecurityContext` object. The `SecurityContext` is used to get the principal user. If the principal is null, the method logs a warning and returns an error message. Otherwise, it creates a new `AirportDAO` and adds the new airport to the `webContext`. The code also handles exceptions and logs errors.

```

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;

```

E-132



Code 11.19: Singletons and Request Context

Table 11.2 lists the values that can be injected into a JAX-RS resource, indicating whether they are request-specific values or not.

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

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

Table 11.2: Injectable Values

Resource Scopes

Oscar Hernando Nivia Aragon (onivia@hotmail.com) has a
non-transferable license to use this Student Guide.

Chapter 12

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

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.

 **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.



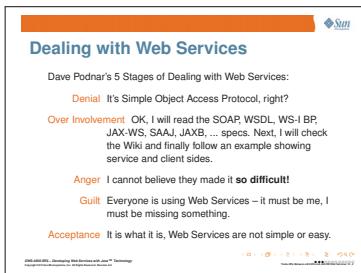
Additional Resources

Additional Resources

The following references provide additional information on the topics described in this module:

- Information was taken from an article published by Dion Hinchcliffe.
“REST vs. SOAP: The Battle of the Web Service Titans”.

Comparing SOAP and REST



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.

Figure 12.1: Dave Podnar's 5 Stages of Dealing with Web Services



You can find a reference to Dave Podnar's list at:

http://www.theserverside.com/news/thread.tss?thread_id=40064 .

It is also referenced in page 3 of Mark Hansen's book, "SOA Using Java™ Web Services", published by Prentice-Hall.

Comparing SOAP and REST

Impedance Mismatch

An issue that developers need to address when building web services has to do with what one could call *impedance mismatch*: the difference between the two representations that they will have to manage within their application: the Java-based, object-oriented representation that the application services are written in, and the XML-based, HTTP-based communications between client and server used to invoke operations on those services. This is illustrated by Figure 12.2.

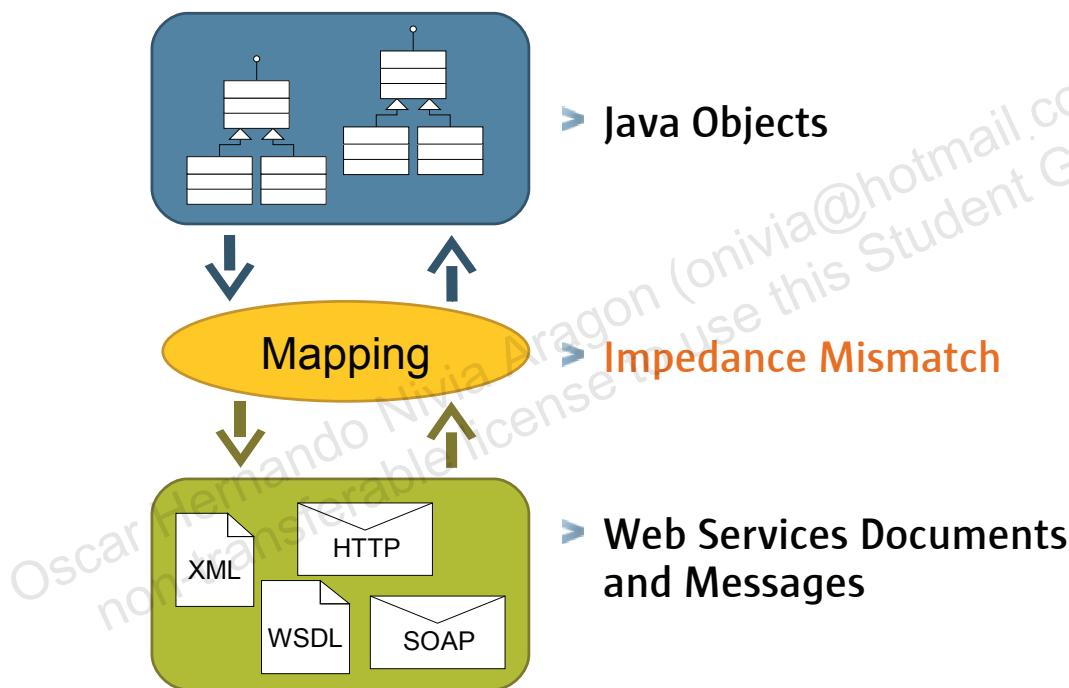
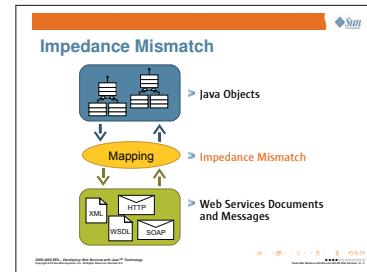
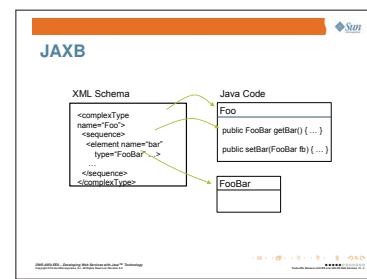


Figure 12.2: Impedance Mismatch

Java provides developers with the Java API for XML Binding (JAXB), a technology that can be used to alleviate some of this concern. JAXB can map between Java and XML representations of the information used by an application – so it is a natural way to address this impedance mismatch. Figure 12.3 illustrates the mapping provided by JAXB.



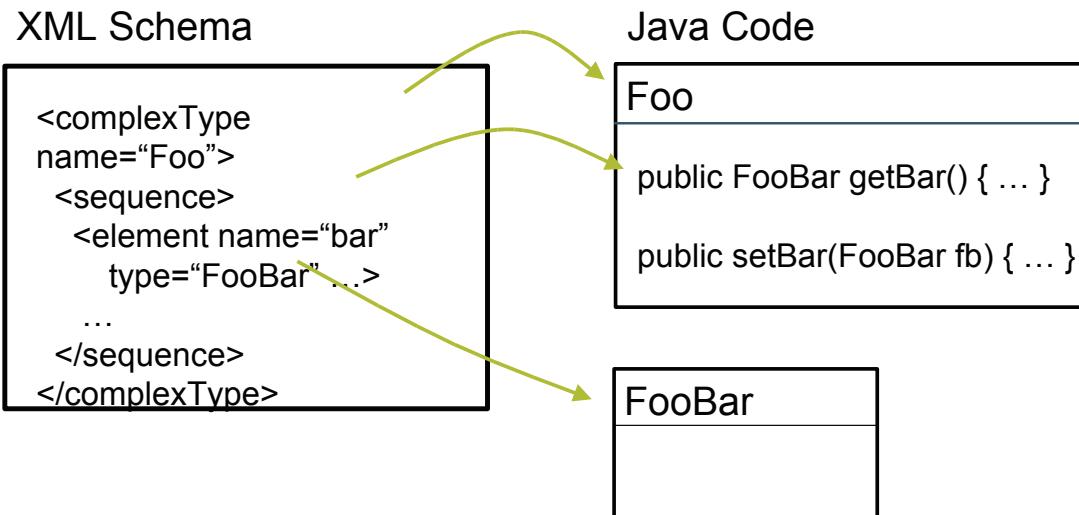


Figure 12.3: JAXB Mapping XML to Java

Unfortunately, JAXB is not a perfect solution:

- It is not transparent – the application level source code must be modified to incorporate JAXB annotations, needed to guide the mapping process.
- It is not complete – there are object-oriented models that JAXB cannot map conveniently to an XML form: object-oriented models support arbitrary networks of objects, while XML makes the assumption that data will be represented as a tree structure; JAXB does not provide any mechanism to address this mismatch, conveniently or completely.

Annotations `@XmlID` and `@XmlIDREF` together allow JAXB to create a customized mapping of a JavaBean property's type by containment or reference. This allows the JAXB runtime to preserve referential integrity of an object graph across XML serialization followed by a XML deserialization, requiring that object references be marshalled by reference or containment appropriately. There are certain constraints on what the object graph structure can be, so that the marshalling and unmarshalling behaves correctly.



Comparing SOAP and REST

JAX-WS Web Services

JAX-WS technology provides mechanisms to map standard descriptions for web services (written in terms of WSDL descriptions and XML schema types) into Java (and vice versa), leveraging the features built into JAXB. The mapping from WSDL to Java – which is the approach to developing web services considered a “best practice” – is illustrated in Figure 12.4.

Communications between client and server for JAX-WS-based web services is built in terms of SOAP messages. Figure 12.5 shows the mapping between a SOAP envelope and the Java entities that that JAX-WS will use to dispatch processing of that request.

SOAP enables applications to access any object on any platform:

- It is an open and standard protocol. SOAP standards are simple compared to those of COM, CORBA, and COM+.
- Debugging can be done. Numerous tools are available that support the development of standardized SOAP-based web services.

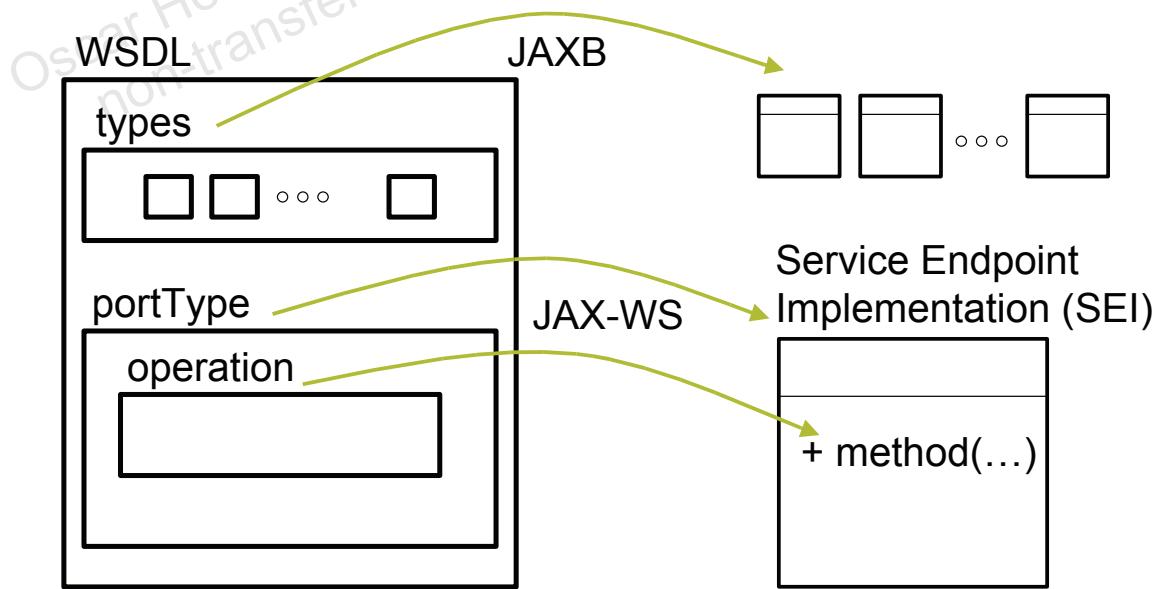


Figure 12.4: JAX-WS – WSDL to Java

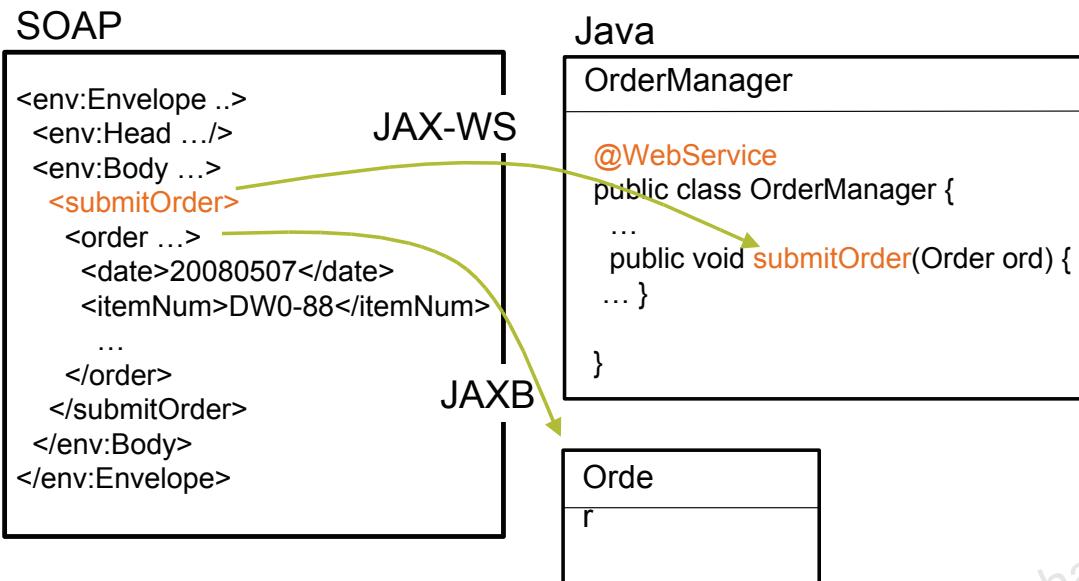


Figure 12.5: JAX-WS – SOAP to Java

- SOAP version 1.2 is cleaner, has better web integration, is more versatile, and is faster.

Disadvantages of SOAP include the following:

- SOAP adds weight at every stage, from server-side processing time to build the message, to the bandwidth to carry it, to the client-side effort, and to unpack it.
- SOAP almost always adds a layer of indirection to queries because the primary noun in a query could be an individual URI; instead, a SOAP-based service forces the query through a single portal URI, a façade, and then finds the requested resources behind that virtual resource.

Most developers, however, consider the functionality provided by SOAP headers and the SOAP encoding valuable additions provided by SOAP-based messages.

Beyond being overweight, SOAP is sometimes viewed as a step backward and away from existing solutions to web service problems:

- For example, HTTP already has a known and widely implemented security mechanism.
- SOAP requires a security mechanism of its own. The much-touted extensibility of SOAP headers is used to solve a problem that SOAP itself creates.

Comparing SOAP and REST

- Where a single URI is a façade for multiple addressable resources, you must implement a policy to protect those individual resources. The security of these resources could easily have otherwise been administered by traditional means.
- SOAP complicates the reuse and consolidation of existing presentational web applications. SOAP typically has separate B2C and B2B interfaces. The advances in XML presentation, such as CSS, Extensible Style Language (XSL), and XLink/XPointer appear to make separation an unneeded complication.

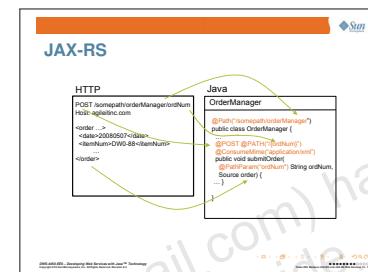
JAX-RS Web Services

XML, HTTP, and URIs can be fused into a robust messaging system without SOAP. An alternative to SOAP might be a resource-based service distributed over URIs, as the existing, secure, and scalable web is now distributed.

The alternative architecture, REST, is an older architecture from which the SOAP initiative departs. REST relies on direct HTTP to resources by URI with XML as a semantic enhancement.

Figure 12.6 illustrates the scheme JAX-RS describes to dispatch incoming HTTP-based requests to the proper Java resource to handle that request.

RESTful web services offer the following advantages:



- Simpler and smaller messages without the weight of the SOAP envelope.
- Better scalability and security with the leverage of existing and broadly implemented HTTP and URI standards.
- Native support for HTTP-based session management.
- Better performance due to established page-caching capabilities, which SOAP-based services cannot reuse.

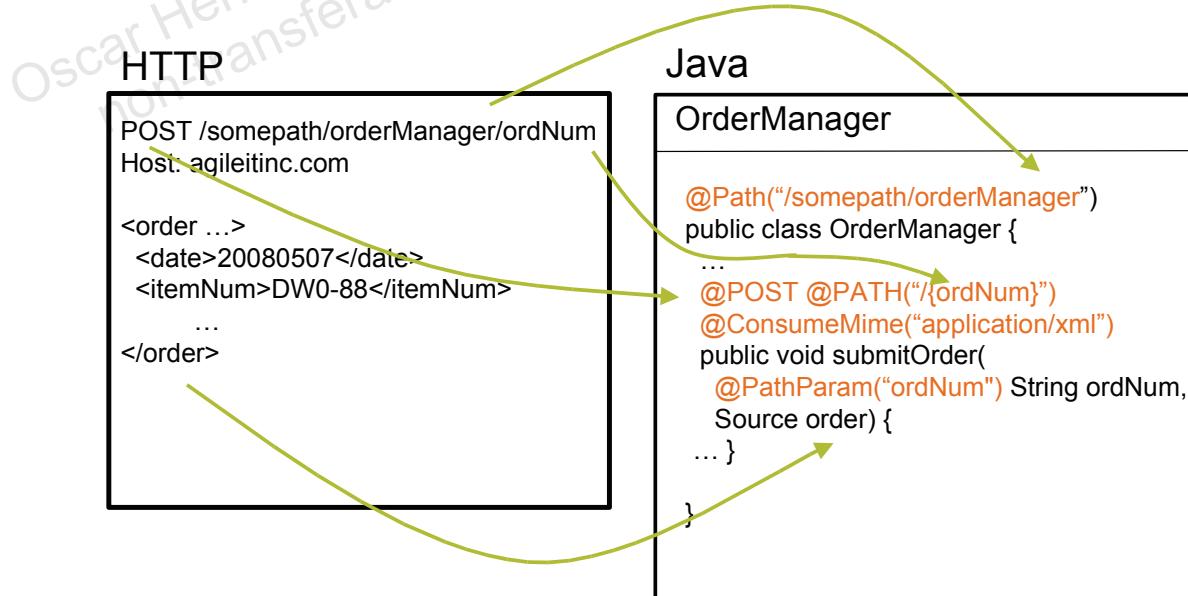


Figure 12.6: JAX-RS Mapping Message to Java

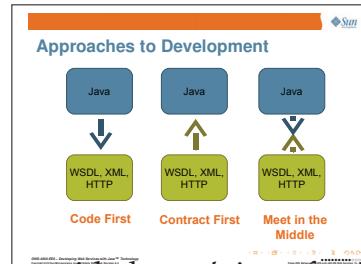
Comparing SOAP and REST

- Easier, even trivial, integration of web services with web applications. The browser-based B2C client and the more automated B2B client make the same request, and then carry the XML response in different directions.
- Easier and tighter integration and extensibility using direct resource linking with the leverage of XLink and XPointer.

Although there are benefits to the REST architecture, SOAP has gained support among developers looking for a more robust mechanism for creating and maintaining interoperable services accessed across the Internet. Indeed, it has become a cornerstone in the current web services framework.

SOAP Compared to REST

Decide upfront, based on the requirements and constraints, what approach for Web Service development best suits your situation:



- Top-down or contract first. The starting point here is the contract of the Web Service: its WSDL description. You either design it or it is provided as a 'given fact'. From the WSDL you generate the implementation. If the contract frequently changes, regeneration of the code can cause difficulties since the implementation is overridden. If you use this method, make sure you don't change the generated artifacts.
- Bottom-up or code first. The starting point is the implementation; all Web Service artifacts, such as its WSDL descriptions, are generated. This is a fast approach when you want to expose existing components as Web Service. However, you need to be careful because you have limited control over the generated Web Service artifacts and it is therefore easy to break an interface if the Web Service is regenerated.
- Meet-in-the-middle approach. Here you define both contract and implementation yourself and later on create the glue between them. In case of Java you can use JAX-WS and JAXB APIs and code to create this glue. This is a very flexible approach: you can change both the WSDL description and the implementation. It requires more work in the beginning, but is easier to change later on.



This section taken from a blog entry by Ronald van Luttikhuizen on best practices for web services, at:

<http://www.approach.nl/2009/07/best-practices-2-web-services/>.

Contract-first approaches may be harder to follow, if the developer chooses to go for a RESTful architecture: neither the overall REST approach, nor the standard APIs that support REST (like JAX-RS), include standards for a abstract document that could describe the features of the web services to be created, the way that WSDL descriptions capture web services that can be implemented then in Java using JAX-WS. There are proposals for documenting RESTful services after the fact, like WADL – but they are not in common use.

Comparing SOAP and REST

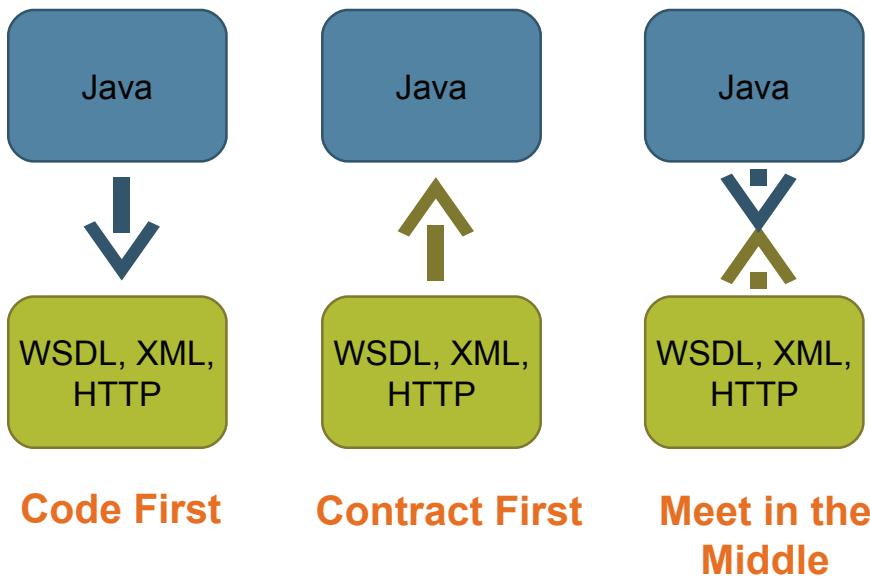


Figure 12.7: Approaches to Web Services Development

One way to compare JAX-WS and JAX-RS web services considers the application-level design approach that needs to be taken:

- JAX-WS web services are described in terms of *activities*, where each application is free to define any operations that it considers appropriate. This approach is illustrated in Figure 12.8.
- JAX-WS web services are described in terms of *resources*, which are then manipulated (in principle) in terms of a fixed set of operations that correspond to HTTP methods. This approach is illustrated in Figure 12.9

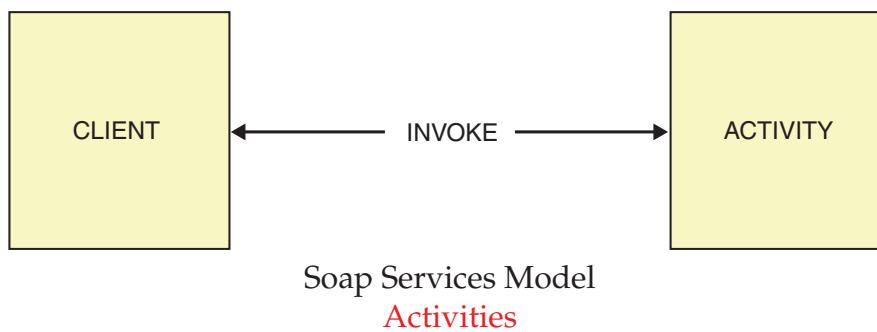
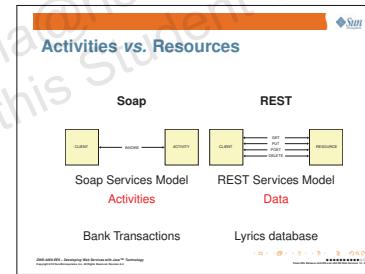


Figure 12.8: JAX-WS Activities

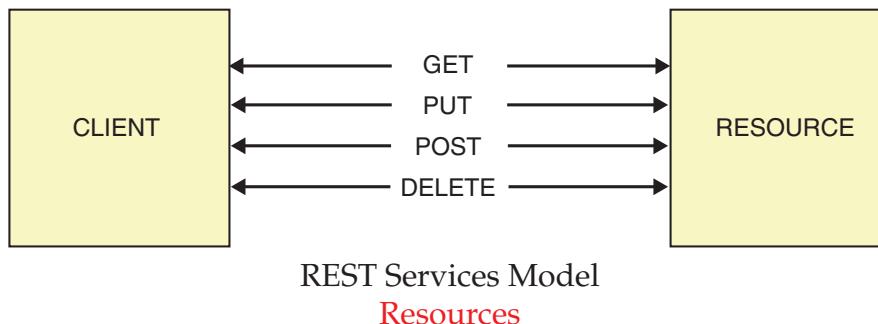
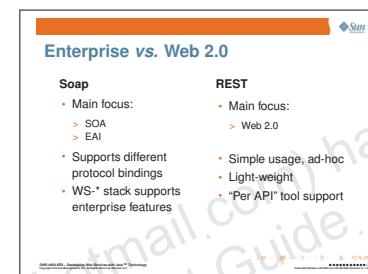


Figure 12.9: JAX-RS Resources

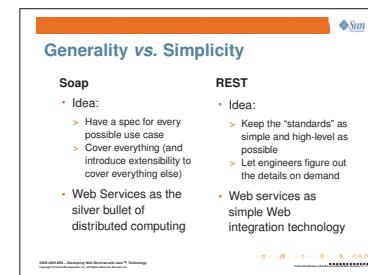
Another way to compare JAX-WS and JAX-RS web services is based on their characteristics:

- SOAP
 - Main focus is SOA and EAI.
 - Supports different protocol bindings
 - WS-* stack supports enterprise features
- REST
 - Main focus is Web 2.0
 - Simple usage, ad-hoc
 - Light-weight
 - “Per API” tool support



Yet another way to compare JAX-WS and JAX-RS web services is based on the intent behind each of the two technologies:

- 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



Comparing SOAP and REST

SOAP-based services represent a method that can be invoked, compared to REST, with which you use URIs to uniquely identify resources in a message. These resources are available for edit, addition, and deletion.

Selecting a web service is based on the situation. REST is an attractive choice for more basic applications that involve high levels of interoperability among multiple platforms. SOAP is appropriate for larger, formal applications that require advanced capabilities between relatively homogenous systems. SOAP's ability to leverage standard web protocols, such as WS-Security, WS-Policy, and WS-Transactions, make rich functionality easy to deliver quickly.

Chapter 13

Web Services Design Patterns

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

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

The slide has a blue header bar with the Sun logo. Below it, the word "Objectives" is in bold blue text. A bulleted list follows: "On completion of this module, you should be able to:" with two items: "Describe web services-based design patterns" and "Describe web services-based deployment patterns". At the bottom right, there is a small navigation bar with icons for back, forward, and search.

Additional Resources

Additional Resources

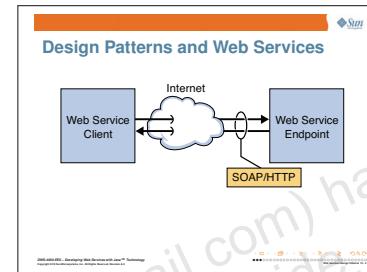
The following references provide additional information on the topics described in this module:

- Lai, Ray, "J2EE Platform Web Services". Prentice-Hall PTR; 1st edition (August 15, 2003).
- Sameer Tyagi, "Patterns and Strategies for Building Document-Based Web Services: Part 1 in a Series," September 2004,
<http://java.sun.com/developer/technicalArticles/xml/jaxrpcpatterns>
- Roberto Chinnici and Marc Hadley, "Next Generation Web Services in the Java Platform," May 2005,
http://gceclub.sun.com.cn/java_one_online/2005/TS-7230/ts-7230.pdf
- Russell Butek, "Which style of WSDL should I use?," October 31, 2003,
<http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl>
- Chris Peltz and Mark Secrist, "Using XML schemas effectively in WSDL design," March 2004,
http://devresource.hp.com/drc/slides_presentations/schemaWSDL
- Ayesha Malik, "Create flexible and extensible XML schemas," October 1, 2002,
<http://www.ibm.com/developerworks/library/x-flexschema>

Design Patterns in the Context of Web Services

Design patterns provide a solution to recurring problems in a given context. A design pattern describes the arrangement of classes and objects in a software design that helps you solve a known problem. Each structure or pattern of classes applies only in a certain context. The 23 patterns enumerated by the authors known as the Gang of Four (GOF) are considered standard design patterns. Sun has published an additional 21 patterns for the Java EE platform.

Web services add an additional abstraction that can be used to describe the interface exposed by Java EE applications to their clients. Web services expose the application interface to the clients in terms of XML, which is a platform, language, and network-neutral representation. However, when using XML, the interaction might include the following operations, as shown in :



- Client code must encapsulate the request, which can be a SOAP request, in an XML document.
- The request is delivered synchronously or asynchronously to the server.
- The server application parses the request.
- After disassembling the request to figure out what the request requires and what data it provides, a call is delegated to the business logic.
- The business logic processes the request and generates a response to the client.
- The response, which can be a SOAP response, must be encapsulated in an XML document.
- The response is delivered to the client as a message.
- The client disassembles the message to extract the response.

The preceding operations can degrade performance metrics for the entire application. However, using the appropriate design patterns, you can minimize this effect.

Figure 13.1 illustrates a simple web services interaction.

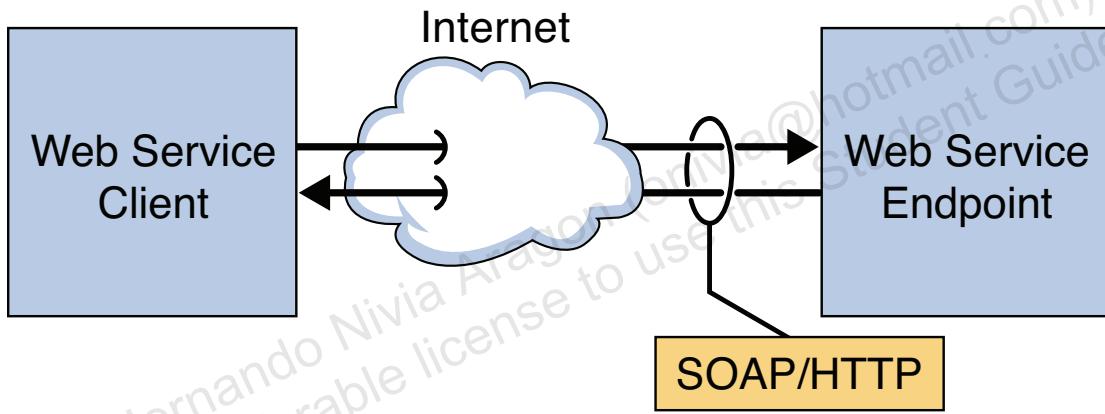
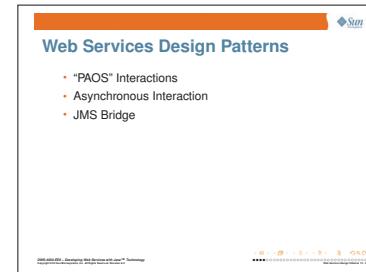


Figure 13.1: Simple Web Services Interaction

Web Services Design Patterns

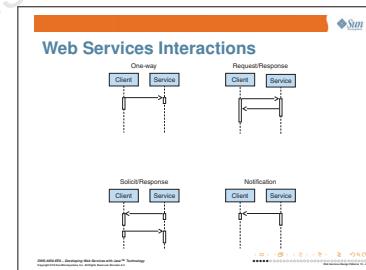
There are various design patterns that are relevant to the design of web services-based applications. The patterns help to enhance the maintainability of the solution, or to minimize the Quality of Service (QoS) impact associated with building applications using web services frameworks. Some basic web services-based design patterns include:

- “PAOS” Interactions
- Asynchronous Interaction
- JMS Bridge



“PAOS” Interactions

The authors of the original web services specifications did not want to lock developers into specific modes of interaction between clients and web service providers. In the WSDL 1.1 specification, they present a number of different modes of interaction that they thought applications would find useful (illustrated in Figure 13.2):



One-Way The client makes a request of the server, not expecting to receive a response.

Request-Response The client makes a request of the server, then waits for the matching answer to be delivered back to the client.

Solicit-Response The server queries the client for information, then waits for the client to deliver the matching answer back to the server.

Notification The server delivers an unsolicited notification to the client, without expecting any response from the client.

The first two modes of interaction are very easy to implement on top of any framework that supports client-initiated request-response interactions; in particular this includes the typical distributed objects frameworks (such as CORBA, or RMI), and many other client-server protocols (such as HTTP).

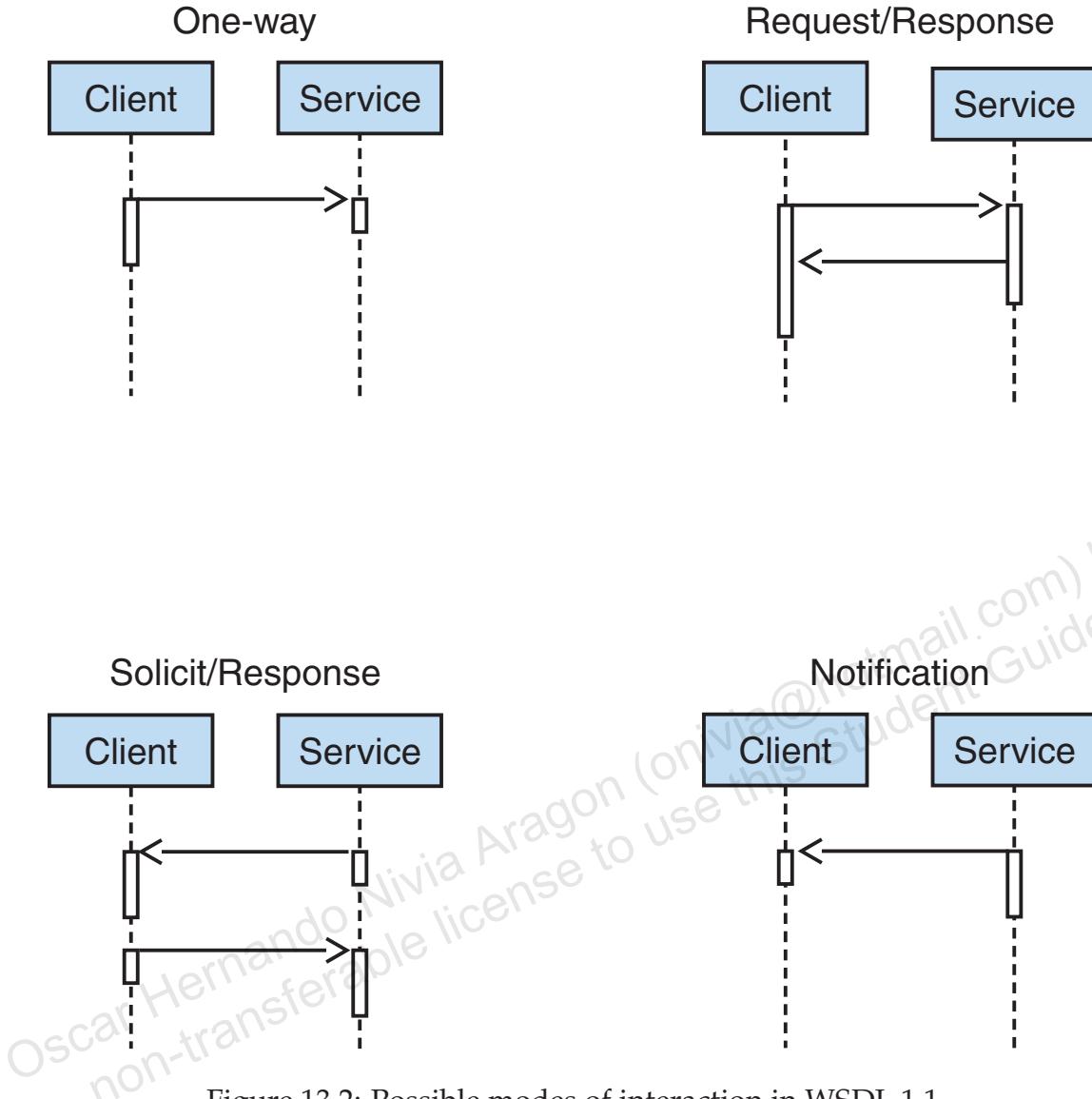


Figure 13.2: Possible modes of interaction in WSDL 1.1

The second two modes of interaction, on the other hand, are quite cumbersome to implement in any client-initiated protocol – yet they are considered useful enough to merit support. Although one could imagine leveraging protocol-specific features to support these modes of interaction, it would be best if a solution were to exist that would be portable to any underlying communications protocol.

The “PAOS” pattern proposes one such portable implementation. Since client-initiated interaction is a mode of interaction that all distributed frameworks support, it seems like it would be a good candidate for a portable design for

“PAOS” Interactions

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

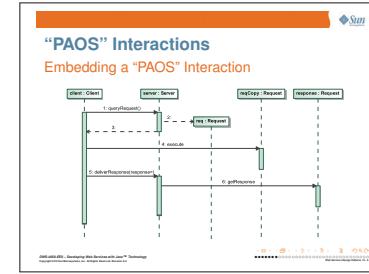
Forces The implementation of this service interaction needs to be portable.

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 - Developing Web Services with Java™ Technology

the two server-initiated modes. The “PAOS” pattern suggests that one could capture a server-initiated interaction within a larger client-initiated conversation, as illustrated in Figure 13.3:

- Since the server cannot initiate a conversation, the client must do so. But, to support server-initiated interactions, the initial request from the client to the server could simply poll whether the server has any requests pending. If so, the reply from the server to the client’s first request would just be the request from the server to the client.
- Once the client has processed the server request, it could issue a second request to the server. But, this second request would simply carry from client to server the client’s response to the server’s earlier request. The server could simply acknowledge receipt of that response.



This way, a server-initiated solicitation-response would be embedded within two client-initiated request-response interactions.

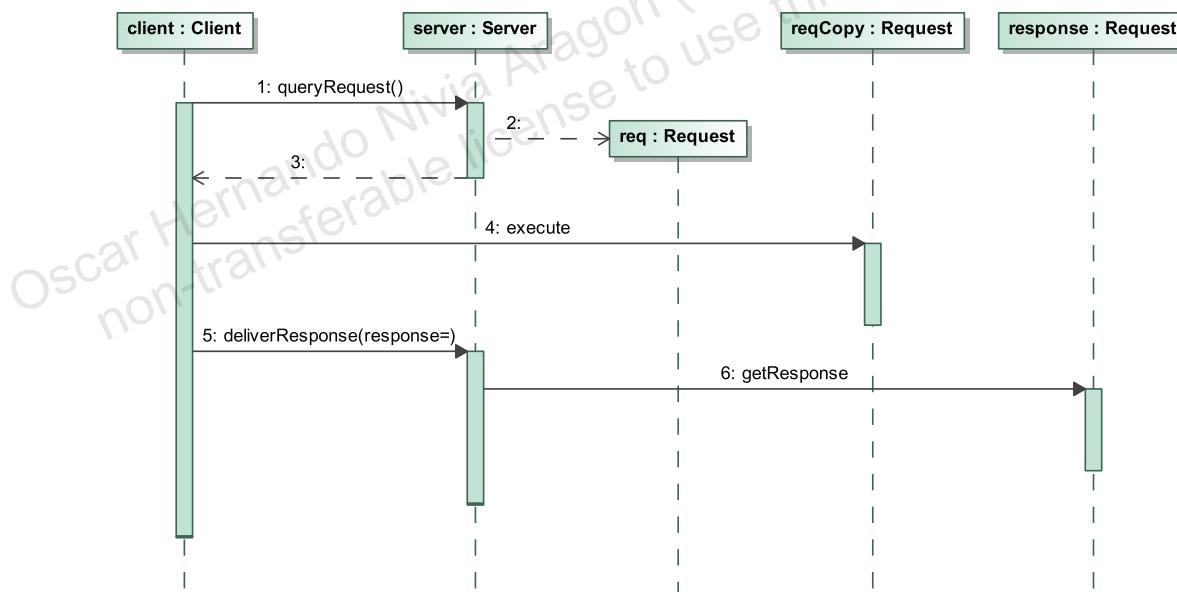


Figure 13.3: “PAOS” Interactions

Advantages and Disadvantages

Web Services Design Patterns

The main advantage to the “PAOS” Interaction pattern is that it allows server-initiated interactions to be captured within two interactions of any client-initiated framework, thus providing a portable way to implement such interactions.

However, 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, not to mention the additional load on the server, when it has nothing to request.

“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

There are various reasons why a web services-based interaction might be too expensive for the client:

- The operation itself is too expensive, in terms of the server-side processing required
- Communications overhead between client and server is too high, either in terms of the amount of data to transfer, or due to bandwidth limitations
- Encoding and decoding of the messages exchanged is too expensive

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.

Some of these reasons might be related to the particular framework, such as web services, used to build the application. But it could also be that the nature of the interaction is such that the time required to process the request would have been unacceptably high, regardless of framework. It is the responsibility of the application designers to address this concern with alternatives.

The first observation is to check whether the client, issuing one of these expensive requests, actually needs to wait for the processing of the request to be finished. When the interaction is described as a web service, the author of the description can choose whether:

Alternative Scenarios

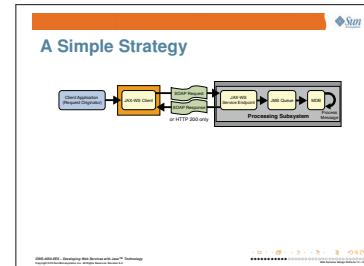
- Client needs nothing from processing request:**
 - Web service contemplates input with no output
 - Can require WS-ReliableTransport
- Client needs information that results from processing request:**
 - Must decouple request from response
 - Must deliver responses to clients
 - Must match requests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

- This interaction requires a message from the client to the server that carries the request to be processed, followed by a reply message from server

to client, carrying information back to the client, or an acknowledgment that the work was done.

- This interaction requires only a message from the client to the server that carries the request to be processed, with the implication that there is nothing additional for the client.

If the second interaction applies, then you can see whether your framework enables you to capture the notion that the client that makes the request can trust that the request is processed. The web services framework can implement this interaction so that the framework carries the request from client to server, then allows the client to proceed without waiting for an answer. Some web services frameworks insist that the method called by the framework return, before allowing the client to proceed. In this case, it might be possible to implement the server-side method so that it returns immediately, while the business process proceeds independently. This last scenario is illustrated in Figure 13.4.



The first scenario requires that you build an application-level structure to capture your intent. This is the case that is addressed by the Asynchronous Interaction pattern.

The first scenario requires that you describe the interaction with a pair of messages: one carries the request from the client to the server, along with any data necessary for processing the request; the other carries information that must be provided to the client, once processing of that request from that client is complete. When describing this as a synchronous interaction, you can take advantage of the fact that the framework naturally associates the reply message to the proper request message, something the framework can do because the client was waiting for that reply.

When describing this first scenario as an asynchronous interaction, there are three goals to achieve:

- Decouple the input and the output messages
- Deliver the output message from the server to the client
- Associate an output message to the corresponding input message

While it might be possible to describe this scenario using the Web Service Description Language (WSDL), there is no portable way to have a web services

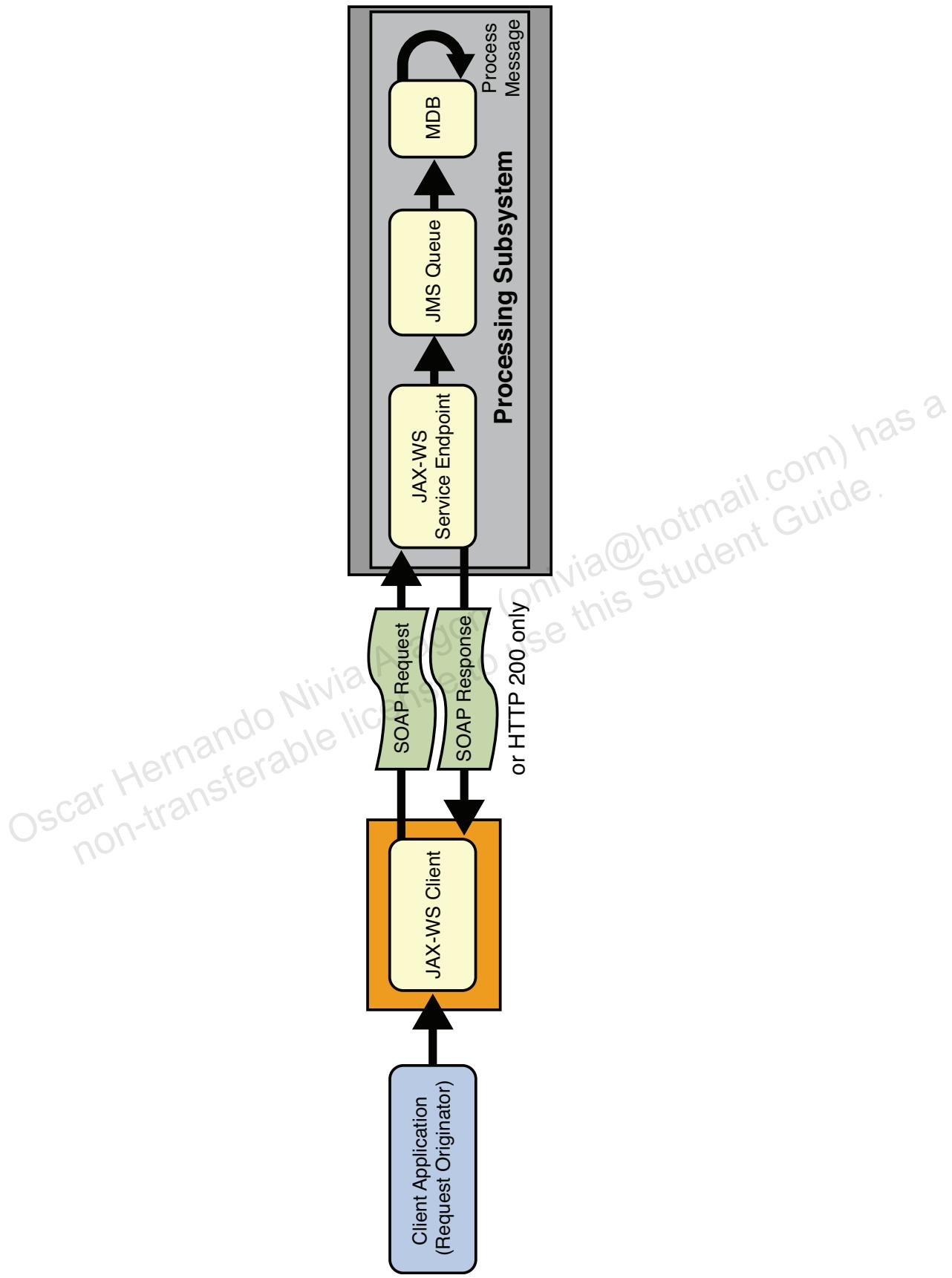


Figure 13.4: A Simple Strategy
Developing Web Services with Java™ Technology

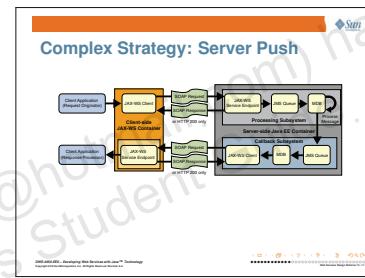
Copyright 2010 Sun Microsystems, Inc. All rights reserved. DWS-4050-EE6 Rev. A.0

framework provide an implementation that can achieve all three of these transparently. The Asynchronous Interaction pattern describes several application-level designs to achieve these goals:

- Server-side push
- Client-side pull
- JMS-based
- JAX-WS-based

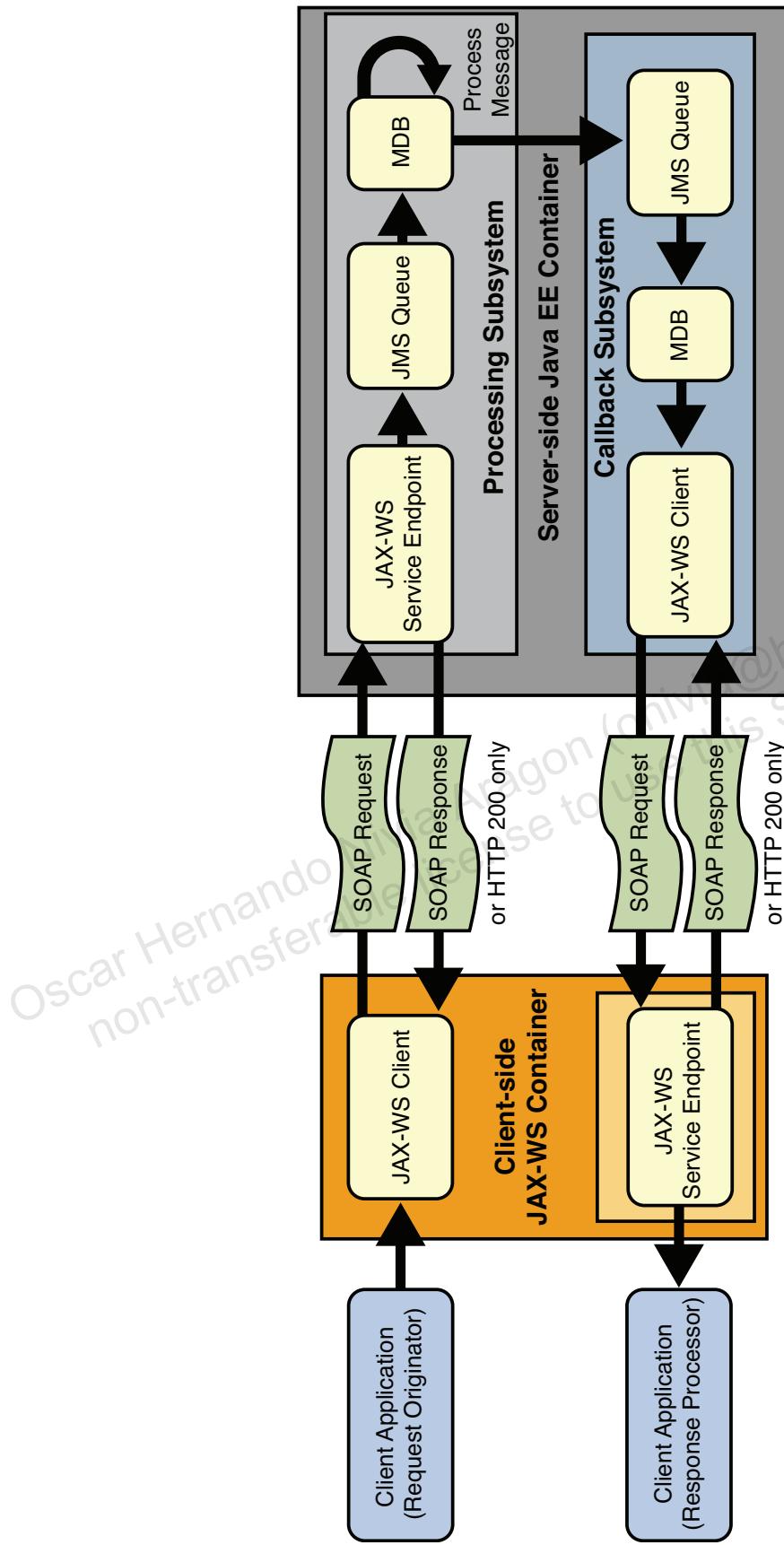
Server-Side Push Implementation

The server-side push implementation decouples the input and output messages for the interaction. In this implementation, the design is converted so that there are two separate interactions; one from client to server that the client uses to issue its request to the server, and another from server to client that the server uses to notify the client of its response, as illustrated in .



In some web services frameworks, a web service operation with a single input message results in an implementation that allows the client to continue execution as soon as the input message is delivered to the server-side application, because the client does not have to wait for anything else. However, other implementations wait for the method that is called to complete execution, to deliver an acknowledgment back to the client that allows the client to proceed. Such implementations are of no use because the client ends up waiting anyway, even after the return message has been separated into its own interaction. illustrates how to ensure that the input message delays the client as little as possible. In the figure, notice how the server-side implementation is written in terms of a web service provider that receives the method call, and immediately forwards the request using JMS to a separate provider. This decouples the method called by the web service from the actual execution of the service. This scheme allows the method called by the web service to return immediately, which results in the acknowledgment being sent back to the client as soon as possible. The client is then free to continue execution.

The web service client can execute additional logic while the service request is processed by the server side. However, in the given scenario, the client needs to receive an answer to its request from the server. In the server push scenario, this notification becomes the responsibility of the server side. However, for the client to properly process this response, the client must recognize this request



made by the server as the response to an earlier request by this client. The following alternatives are possible:

- The client could supply the address of a web service dedicated to process the specific response to this request, as part of the request. In this case, the server contacts this dedicated web service to supply the answer to this query back to the client. The fact that the server contacts a particular dedicated web service serves to identify to which request the server is replying. Figure 13.6 illustrates the server-side push implementation using a UML class diagram

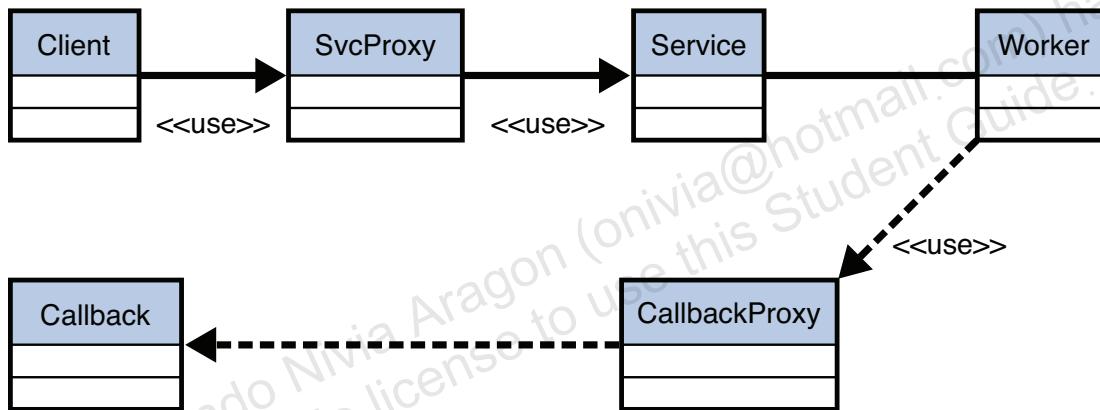
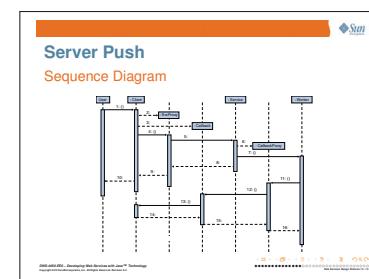
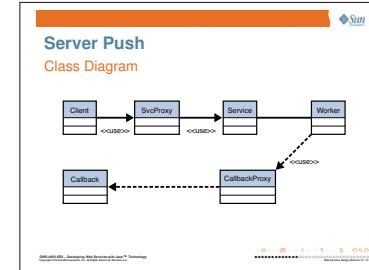


Figure 13.6: Server Push Class Diagram

Figure 13.7 illustrates the server-side push implementation using a UML sequence diagram.



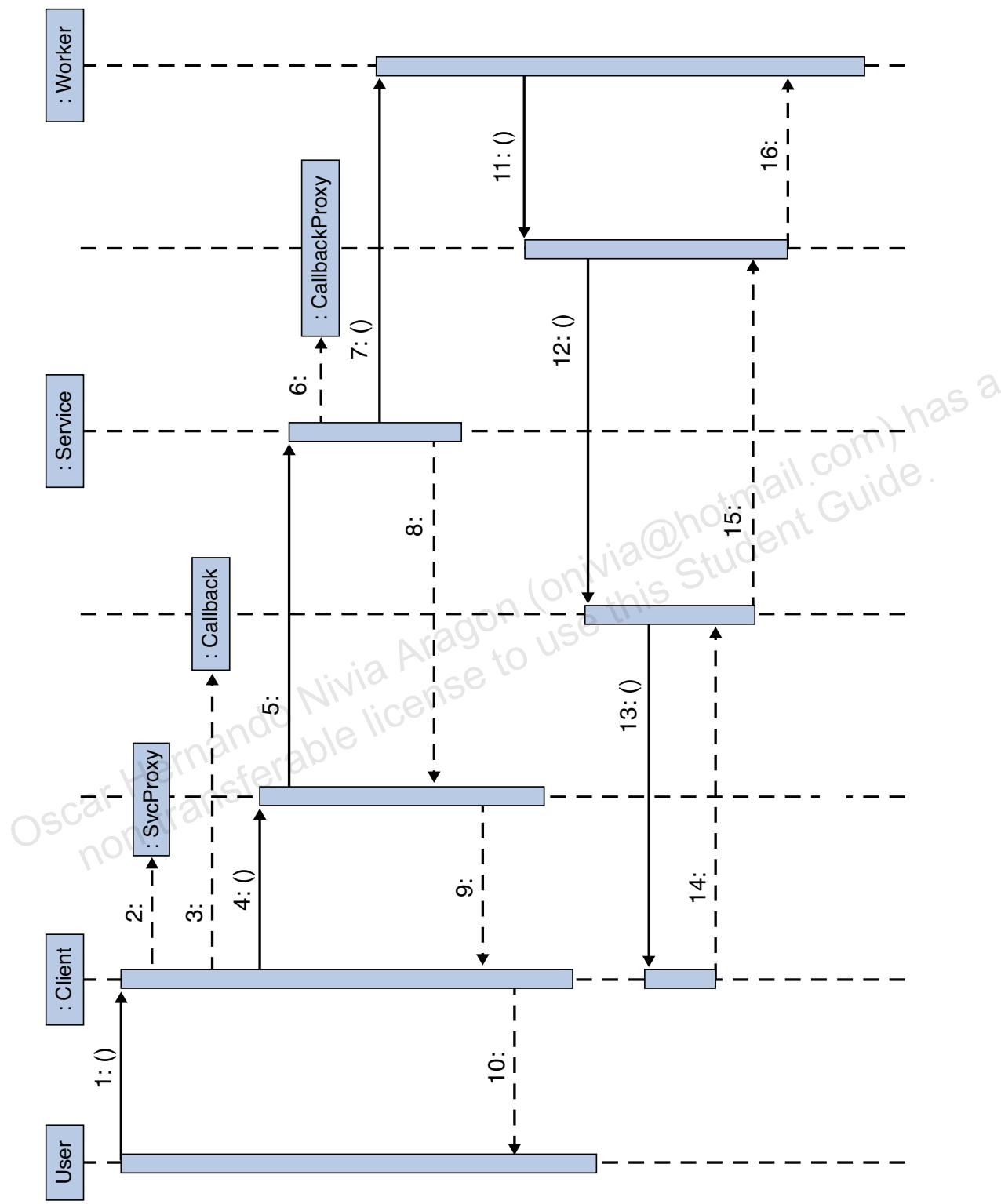


Figure 13.7: Server Push Sequence Diagram

- The client could supply the address of a generic web service that can be invoked to supply the results of operations back to that client, along with a unique token, also part of the input message, that can be used later to identify this request. When the server later invokes this generic web service to deliver an answer to this client, the server uses the same token as part of the response message, to identify the response to the client. The WS-Addressing specification provides a portable means to identify entities, which you could use to define these tokens portably. Figure 13.8 illustrates a second approach to the server-side push implementation using a UML class diagram.

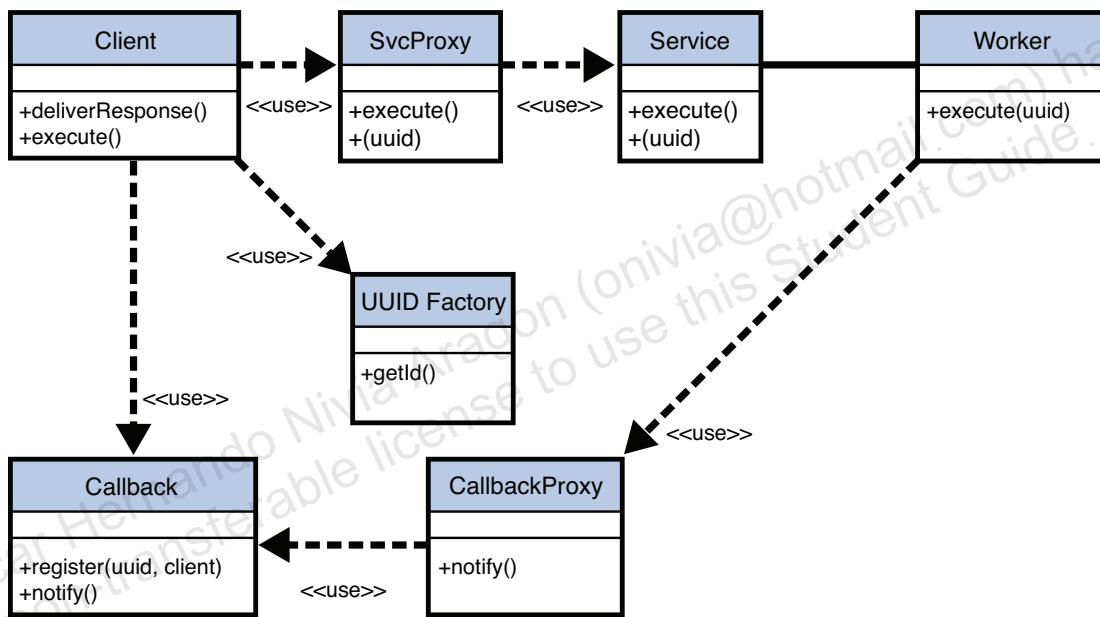
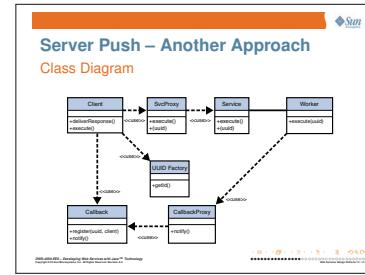
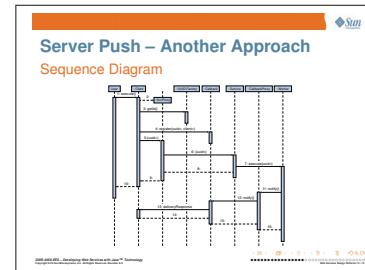


Figure 13.8: Server Push Class Diagram - Second Approach

Figure 13.9 illustrates the second approach to the server-side push implementation using a UML sequence diagram.



Client-Side Pull Implementation

Sometimes, the client cannot set up a web service endpoint for the server to use for the second half of the interaction. In these cases, a variation of one of the schemes described previously could be that the client issues a request to the server, including a unique token that identifies the particular request to the server. The server accepts the request, allowing the client to continue

Web Services Design Patterns

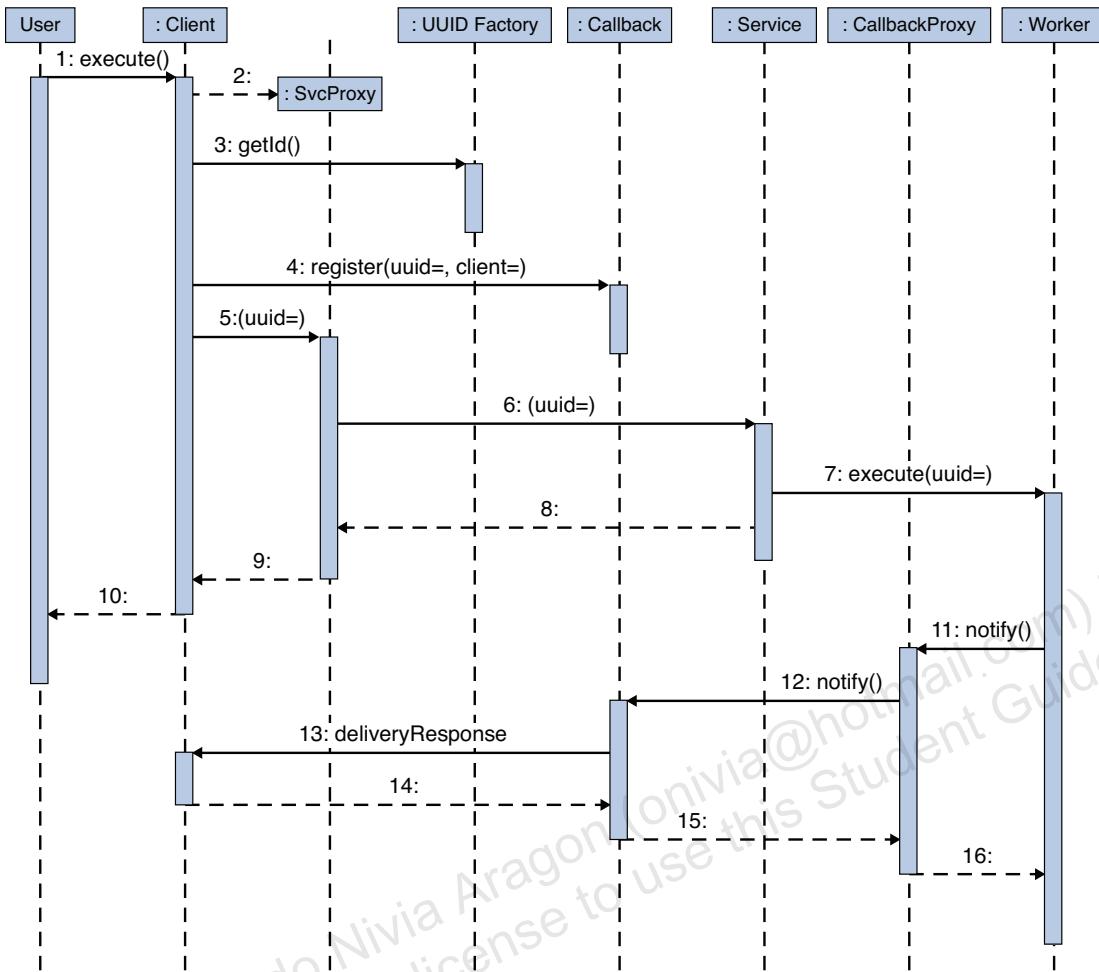


Figure 13.9: Server Push Sequence Diagram - Second Approach

processing by means similar to the ones described previously. However, in this variation, the server processes each request to obtain the appropriate response. Then, the server stores all responses, indexed by the tokens supplied by the client in each request, in a data structure accessible to a new web service, called the response web service. This web service is exposed to all clients. Each client can query this new web service for the answer to earlier requests using the same token the client used to identify each request originally. This token allows the response web service to identify the appropriate answer for this client. If that answer is not yet available, the response web service indicates that the response is unavailable, and the client can ask again later.

This scheme requires storage, enough to hold all the responses computed for each client until each client retrieves their response. For reliability, this storage might need to be persistent, so that clients can retrieve their response regardless of a transient server or network failure. This scheme might also increase network overhead, because clients might need to poll the response web service.

periodically until they retrieved their answer.

JMS-based Implementation

The portable implementations described previously require the web service implementation to be built so that the method call invoked on the server in response to a client request returns immediately to the client, while the server continues to process that client's request in a separate thread. This is required so that the solution is portable, because the web services specifications only require support for request-response based interactions using HTTP. Another portable strategy to achieve this concurrent execution within the Java EE specification relies on JMS. The sending of a message using JMS is decoupled from the receiving of that same message. This allows the thread responsible for processing the client's request within the web service implementation to return to the client once the message is sent. JMS delivers the message independently to a message-driven bean (MDB) to process, separately.

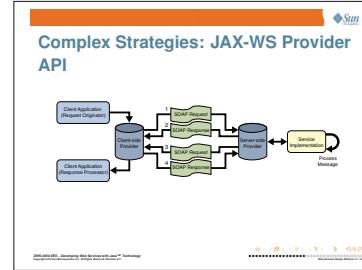
If you are building a non-portable web services solution within the Java EE specification, then you have an alternative. JAX-WS allows the application to define web services that are implemented using JMS as the message transport, instead of HTTP. You can build an alternative implementation of either the server-push or client-pull solution described previously. With the new approach, instead of using the web services framework to deliver a request using the request-response paradigm to a web service implementation, you can implement a web service using the web services framework, but configure it to use JMS as its transport. This strategy allows the client to issue a web service call to a remote web service, but the web service framework could deliver the request using JMS. Because you are using JMS, the web services framework is not required to wait for a reply. If the web service is defined to not expect a response, then the implementation can deliver the request, and allow the client to move on. Figure 13.4 illustrates this strategy for a server-push implementation.

In the client-pull implementation, the client issues an initial request using a web service where the client does not want to wait for a reply, but follows up later with a second request to the response service, to retrieve the result for the initial request, if available. This second request is a synchronous operation: The client requests an answer and expects a reply that is either the response desired, or an indication that the response is not yet available. The client's initial request can be implemented using JMS; the later request can be implemented using any portable request-response alternative. Figure 13.8 and Figure 13.9 present UML diagrams for a client-pull implementation that can be implemented using JMS for the initial client request.

JAX-WS-Based Implementation

The web services machinery included in the Java EE specification provides support for non-portable asynchronous interactions, but only for JAX-WS-based interactions. JAX-WS introduces the `Dispatch<T>` and `Provider<T>` interfaces that are used to describe the client and server sides to the interaction.

On the client side, JAX-WS introduces the ability to indicate the intent associated with an interaction; whether the interaction is to be synchronous or asynchronous, and if asynchronous, whether it is client-pull or server-push. Effectively, the API allows the client to tell JAX-WS to send the message that represents the client request, leaving the client to package the request as an instance of the actual underlying message representation. However, the way in which the client asks for this request message to be sent captures the client's intent, as far as this interaction is concerned:



- In the case of a synchronous interaction, you must indicate that an interaction occurs using `invoke()`, declaring the request and return message type when declaring the interface that the client uses. The message type can be the underlying communications message type, such as `SOAPMessage`, or the payload of the underlying message, using JAXB, for instance. When such a method is declared, JAX-WS assumes the client waits for the return message to be produced, whenever the caller initiates an interaction.
- When the interaction is to be asynchronous and client-pull, the method call to make is `invokeAsync()`, which is declared to return a value of type `Response<T>`. This `Response<T>` type eventually produces the return value; its API allows the client to ask whether there is an answer already available to retrieve, or to actually retrieve the answer.
- When the interaction is to be asynchronous and server-push, the method to call is another `invokeAsync()`, which is declared to accept a parameter of type `AsyncHandler<T>` and to return a value of type `Future<?>`. The parameter with type `AsyncHandler<T>` must define a method called `handleResponse()`, which is invoked when the return value for this call is available. The return value of type `Future<?>` allows the client to ask whether the call has completed. No other use of this instance is allowed, when used in a web service context.
- The application can also describe *one-way* interactions. These are interactions in which the client issues a request of the server, but in which the client is not expecting to receive a response, at all. These interactions

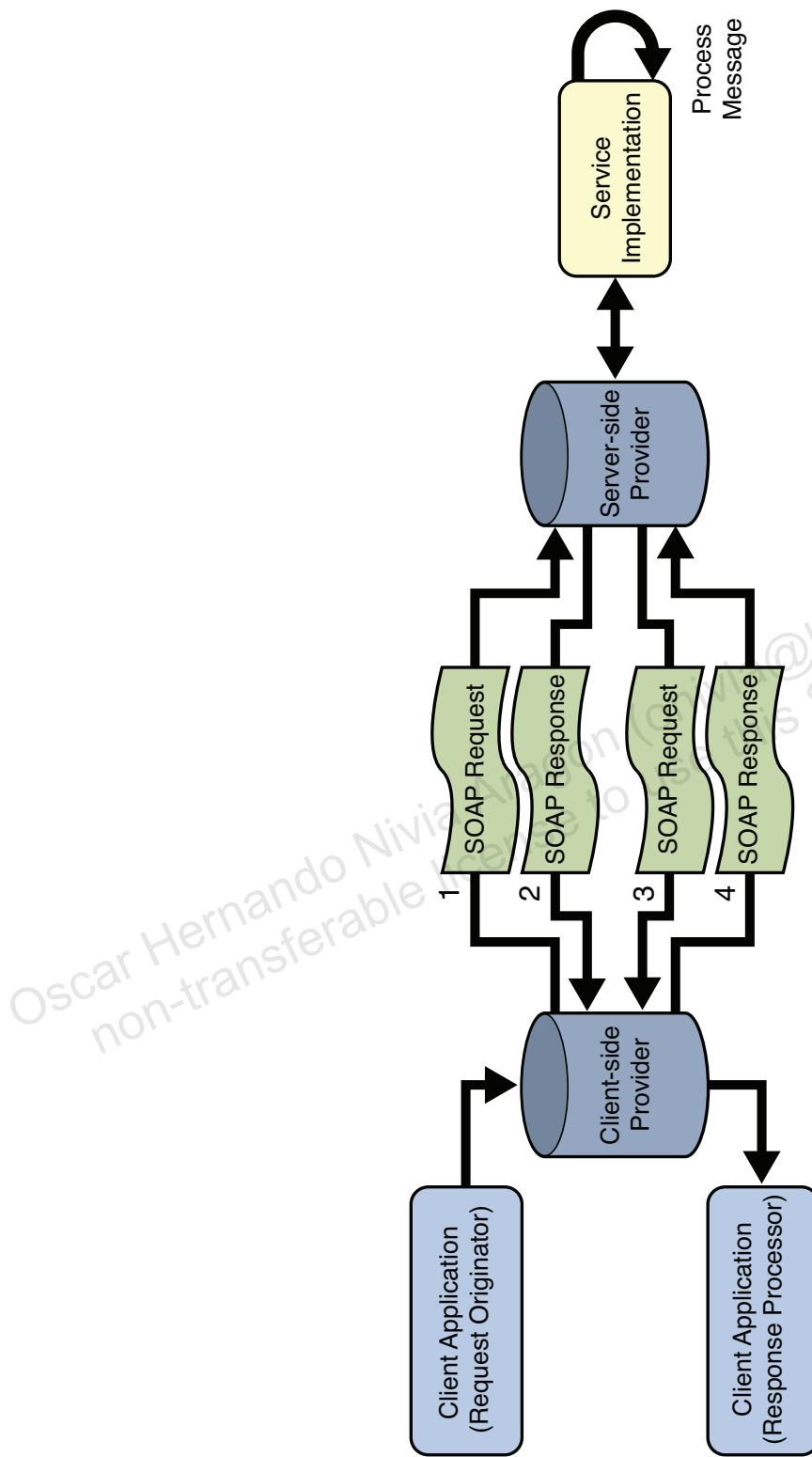


Figure 13.10: Complex Strategies: JAX-WS Provider API

might be implemented by allowing the client to continue, once they issue the request, while the server side processes this request in parallel, but it depends on whether the underlying binding protocols allow it. For example, web services using HTTP as its communications protocol require that the client wait until the message finishes processing on the server side, even though the client was not expecting any response.

This intent is not something that can be captured for application-defined methods on application-defined objects. Clients must call one of `invoke()`, `invokeAsync()` or `invokeOneWay()` on an instance of a `Dispatch<T>` to use this new API. As a result, the caller might have to package the actual request and its parameters into the outgoing message, and extract the response out of the incoming message. Code 13.1 illustrates the client-side messaging interface definition.

On the server side, JAX-WS enables you to define the application-level object responsible for processing requests issued through this dynamic invocation API. Such objects must implement the `Provider<T>` interface to implement the method that handles requests as they are received. These instances must extract the actual request and its parameters out of the incoming messages they received, and dispatch it to the appropriate processing method explicitly themselves. Code 13.2 illustrates the server-side messaging interface definition.

```

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 }
```

Code 13.1: Dispatch Interface

```

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

Code 13.2: Provider Interface

Code 13.3 provides a small example of what the code might look like, on the client side. The API is a more verbose and lower-level than the simpler JAX-WS approach. Rather than obtaining a Java proxy that already implements functions modeled after those defined in the web service's WSDL description, the Dispatch API provides the caller with a generic `Dispatch<T>` proxy that offers a fixed number of `invoke` methods, regardless of WSDL definition. As far as arguments and return values go, these `invoke` functions all accept a single argument representing the content of the SOAP body, and return a single value, representing the content of the SOAP body for the response message.

Code 13.4 provides the matching code for the server side. The Provider API for the server side is equally low-level: service provider classes need only implement a single `invoke` method; it is the responsibility of the service provider to decode the incoming content object, and to assemble together the proper response object to send back to the caller.

```

1 public class MessagingAPIClient {
2     public static void main(String[] args) throws Exception {
3         QName portName = new QName("urn:examples", "Hello");
4         Service service = Service.create(servicename);
5         String url = "http://127.0.0.1:8081/MessagingAPI";
6         service.addPort(portName, SOMPIBinding, SOMPIHTTP_BINDING,
7                         url);
8         Class msgClass = MessagingAPIMessage.class;
9         JAXBContext jaxbCx = JAXBContext.newInstance(msgClass);
10        DispatchObject port =
11            service.createDispatch(portName, jaxbCx, Mode.PAYLOAD);
12        port.invoke();
13    }
14 }
```

```

1 @ServiceMode(Mode.PAYLOAD)
2 #ServiceProviderProvider(portName="Hello", servicename="Examples",
3 #public class MessagingAPIServer implements ProviderSource {
4     MessagingAPIServer() throws JAXBException {
5         JAXBContext jaxbCx = JAXBContext.newInstance(Clazz.class);
6         jaxbCx.setSchema(JAXBContext.newInstance(Clazz));
7         jaxbCx.setSchema(jaxbCx.createSchema());
8     }
9     // ...
10    private JAXBContext jaxbContext;
11    public void start() throws IOException, ServletException {
12        port = createDispatch();
13        String url = "http://127.0.0.1:8081/MessagingAPI";
14        MessagingAPIServer server = new MessagingAPIServer();
15        Endpoint endpoint = Endpoint.publish(url, server);
16    }
17 }
```

Web Services Design Patterns

```

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);
15        MessagingAPIMessage request =
16            new MessagingAPIMessage("sayHello", "Tracy");
17        // synchronous
18        MessagingAPIMessage response =
19            (MessagingAPIMessage) port.invoke(request);
20        System.out.println("Response: " + response.getResult());
21        // asynchronous
22        AsyncHandler<Object> responseHandler =
23            new AsyncHandler<Object>() {
24                public void handleResponse(Response<Object> resp) {
25                    try {
26                        MessagingAPIMessage result =
27                            (MessagingAPIMessage) resp.get();
28                        System.out.println(result.getResult());
29                    }
30                    catch (Exception e) {
31                        {}
32                    }
33                };
34                port.invokeAsync(request, responseHandler);
35            }
}

```

E-135

Code 13.3: Sample Client Using JAXWS Dispatch API



```

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 = JAXBContext.newInstance( msgClass );
8      }
9      public Source invoke(Source payload) {
10         try {
11             Unmarshaller u = jaxbContext.createUnmarshaller();
12             MessagingAPIMessage message =
13                 (MessagingAPIMessage) u.unmarshal( payload );
14             message.setResult("Hello, " + message.getArgument());
15             return new JAXBSource( jaxbContext, message );
16         }
17         catch( Exception ex )
18         { throw new WebServiceException( ex ); }
19     }
20     private JAXBContext jaxbContext;
21     public static void main(String[] args) throws Exception {
22         String url = "http://127.0.0.1:8081/MessagingAPI";
23         MessagingAPIServer server = new MessagingAPIServer();
24         Endpoint endpoint = Endpoint.publish( url, server );
25     }
26 }
```

E-137

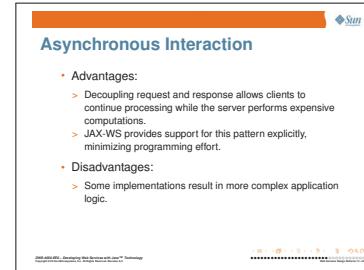


Code 13.4: Server Using JAX-WS Messaging API

Advantages and Disadvantages

Application designs that apply the Asynchronous Interaction pattern can continue processing on the client side while performing expensive operations on the server side. This allows the user running the client application to continue to work while this expensive processing takes place, resulting in applications that seem more responsive.

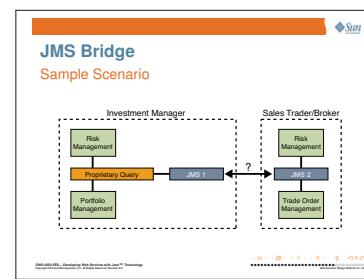
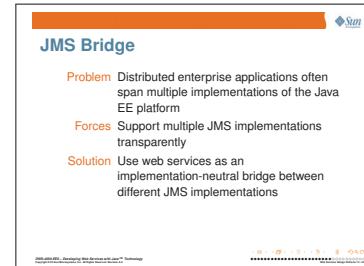
When the application can use a JAX-WS-based implementation of this pattern, this advantage can be realized without increasing the complexity of the design, because JAX-WS can transparently provide the support. When JAX-WS is not an option, the Asynchronous Interaction patterns suggests a number of alternative implementations; unfortunately, these implementations require a more complex application-level design to achieve the necessary goals at the application level.



JMS Bridge

Enterprise applications often use messaging-based protocols for asynchronous interactions within the application, or between it and other legacy applications. In the context of Java EE applications, this typically means that the enterprise application uses JMS to write the application-level machinery that constructs and sends messages to their destinations. JMS allows the application-level logic to ignore the particulars of the messaging product actually used.

Unfortunately, the JMS specification only defines a set of standard Java interfaces for interaction between the application-level logic and the messaging service. Interoperability between two different JMS implementations is not part of the JMS specification. If the enterprise application is complex enough to run on more than one hardware platform, it is not uncommon for there to be two or more JMS implementations in play, and interoperability then becomes an issue, as illustrated in Figure 13.11 . While it might certainly be possible to



simply replace the multiple JMS implementations with a single cross-platform implementation, this might be intrusive to the applications in question.

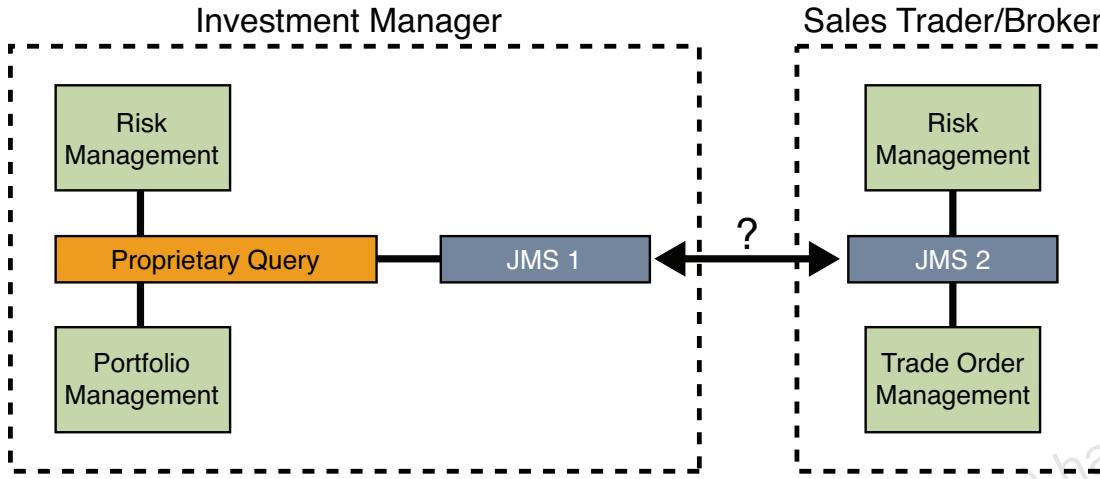
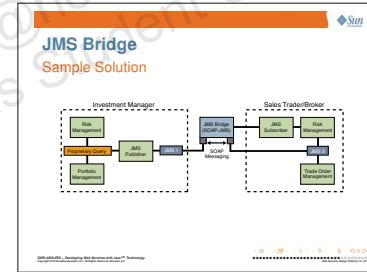


Figure 13.11: JMS Bridge Scenario

The JMS Bridge pattern suggests keeping the different subsystems using their own JMS implementations, but advocates the introduction of a client into the enterprise application that can *relay* messages from one JMS implementation to the next. This requires JMS clients for each of the subsystems, so they can interact with the local JMS implementation in that subsystem, and that are also able to interact with each other to relay messages that appear on one JMS subsystem to the next. The JMS Bridge pattern also suggests to implement each of these subsystem-specific JMS clients as a web service, because the web services framework guarantees interoperability between any two of these JMS clients. Figure 13.12 illustrates the approach proposed by the JMS bridge pattern.



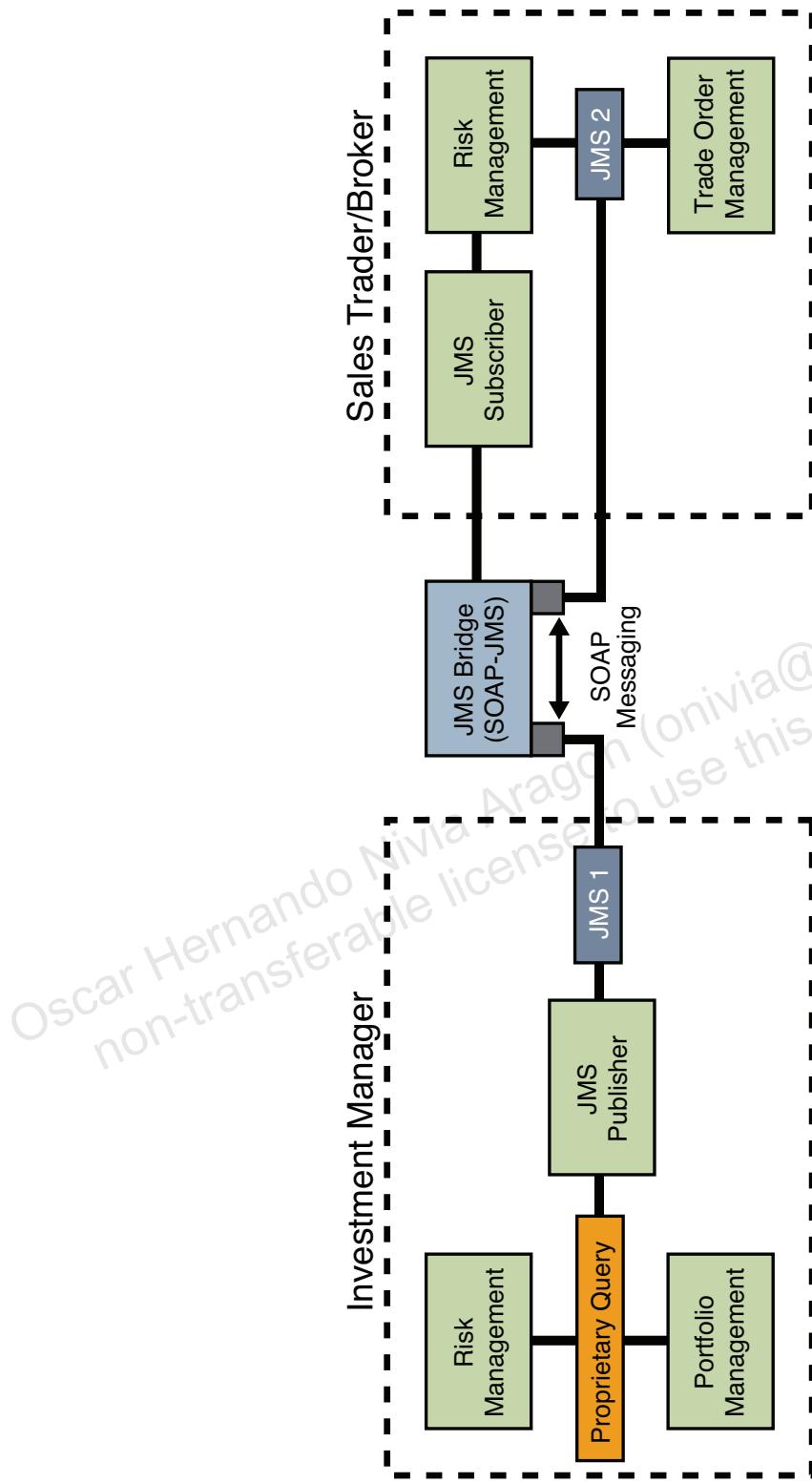


Figure 13.12: JMS Bridge Solution

Advantages and Disadvantages

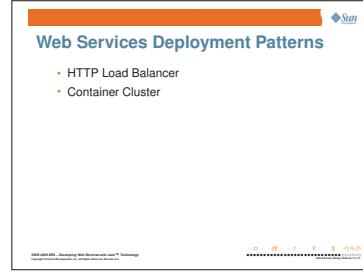
There is no need to develop a vendor-specific adapter to bridge two underlying middleware vendors: a JMS bridge is vendor and messaging API-independent, so it can guarantee interaction between any two JMS implementations.

The main disadvantage of the JMS Bridge pattern is that there is overhead associated with the act of relaying messages across JMS implementations. One of the advantages of using a messaging service is that it can optimize network traffic. However, the presence of a JMS bridge in a design means there is overhead corresponding to the XML encoding, transmission, and XML decoding associated with the relaying of the message.

The screenshot shows a presentation slide with the title 'JMS Bridge' in bold. Below the title, there are two sections: 'Advantages:' and 'Disadvantages:'. The 'Advantages:' section contains one bullet point: 'JMS bridge is vendor/JMS-independent, so it can guarantee messaging between any two JMS implementations.' The 'Disadvantages:' section contains one bullet point: 'JMS bridge incurs overhead corresponding to the XML encoding, transmission, and XML decoding associated with the relaying of the JMS message'. At the bottom right of the slide, there is a small navigation bar with icons for back, forward, and search.

Web Services Deployment Patterns

When considering the architecture of any enterprise application, one has to consider quality of service (QoS) issues, such as scalability or availability of the application. Web services-based applications are no exception, but the characteristics implicit in a web service can help choose strategies to address these issues.

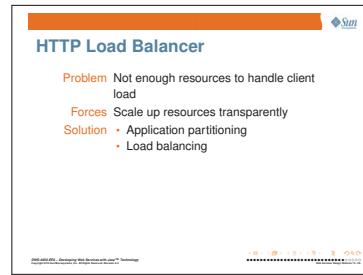


This section describes the following deployment patterns:

- HTTP Load Balancer
- Container Cluster

HTTP Load Balancing

Any enterprise application should consider how to ensure that response times remain acceptable as load on the application increases. There are two simple strategies to consider as starting points, vertical scalability and horizontal scalability.



Vertical scalability suggests that you deploy the enterprise application as a single application, but on a single computer powerful enough to handle increases in load. You can always buy more hardware, though computer hardware that effectively supports the ability to plug in additional CPUs, or memory, or I/O controllers tends to be expensive. On lower-end hardware, you can run into design limitations, such as bus bandwidth, or 32-bit addressing limitations in either the hardware or operating system.

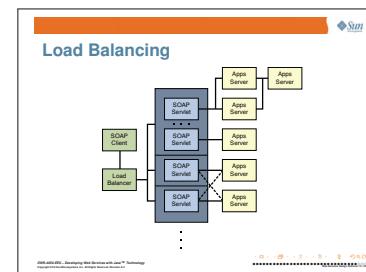
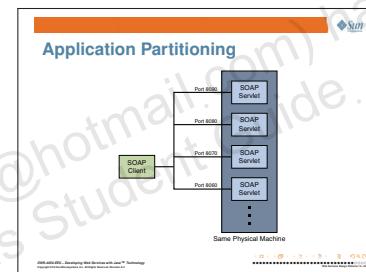
Horizontal scalability suggests that you deploy multiple instances of the enterprise application that are then made to look like a single system, from the point of view of any client. The number of resources available is the sum of resources available to all the multiple instances of the application.

You can deploy multiple instances of the enterprise application on a single host server, or across multiple servers, depending on which resource needs to be addressed:

- If you run into resource limits associated with 32-bit addressing, one possibility would be to deploy multiple copies of the application on a single host server. Each of the copies can allocate the maximum virtual memory available to an application and the multiple copies can then use all the physical memory available to the server.
- If the application is CPU-bound on a single server, one possibility is to deploy multiple copies of the application on separate servers. Each of the copies can then use all the CPU resources available on its server.

One of the problems in choosing horizontal scalability is to route incoming client requests to the appropriate server to be processed. Several possibilities are common:

- Application partitioning suggests that the proper way to route requests to individual servers is by partitioning the space of requests according to some rule. For example, clients whose names begin with A-K are routed to one server, while clients whose names begin with L-Z are routed to the other; or requests for a specific subset of services are routed to one server, and the remaining requests to the other. Figure 13.13 is an example of such an architecture.
- Load balancing suggests that the proper way to route requests is to try to balance load over the multiple copies of the application that are deployed, so that load distribution remains even over time. Figure 13.14 is an example of such an architecture.



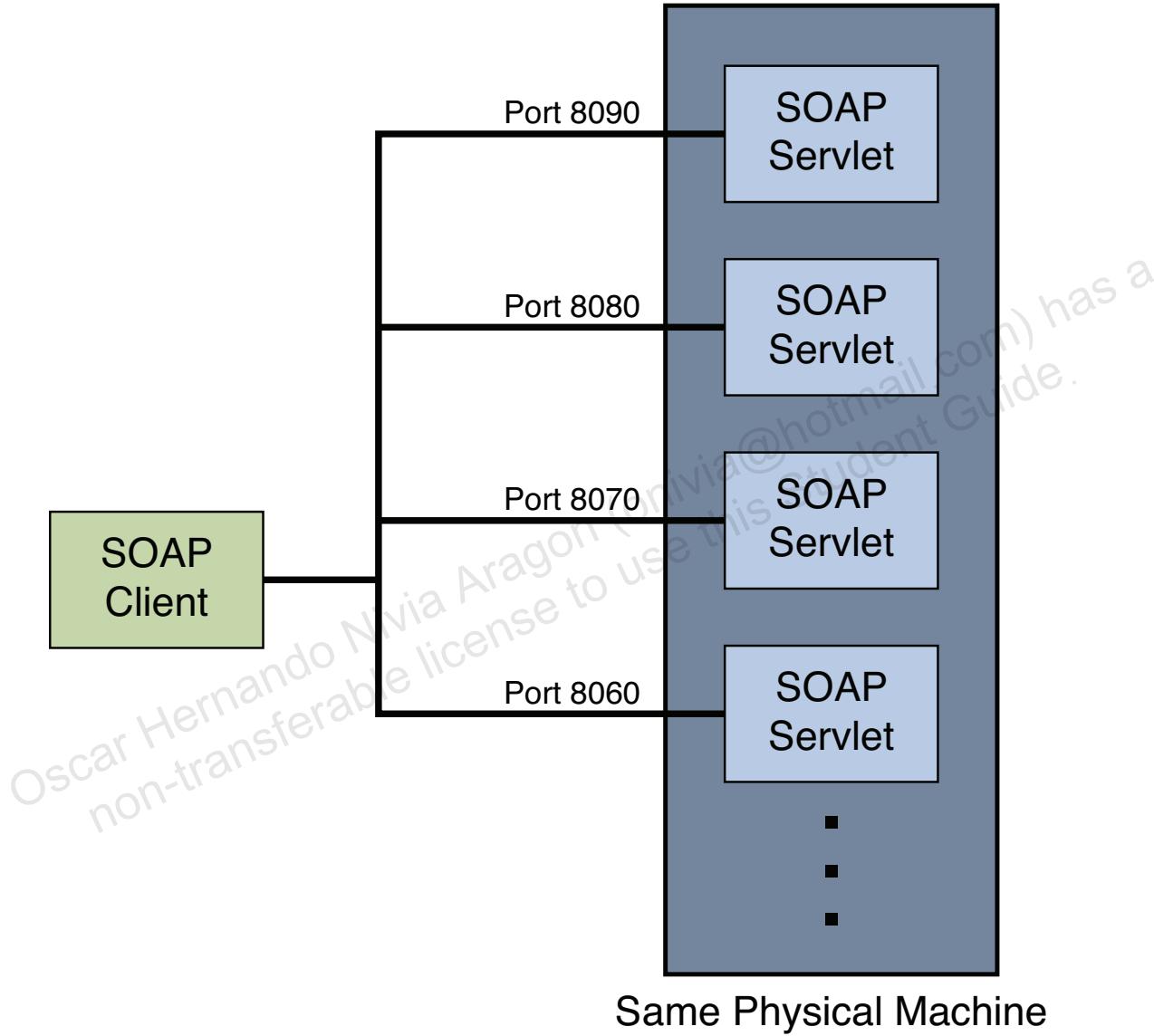


Figure 13.13: Application Partitioning

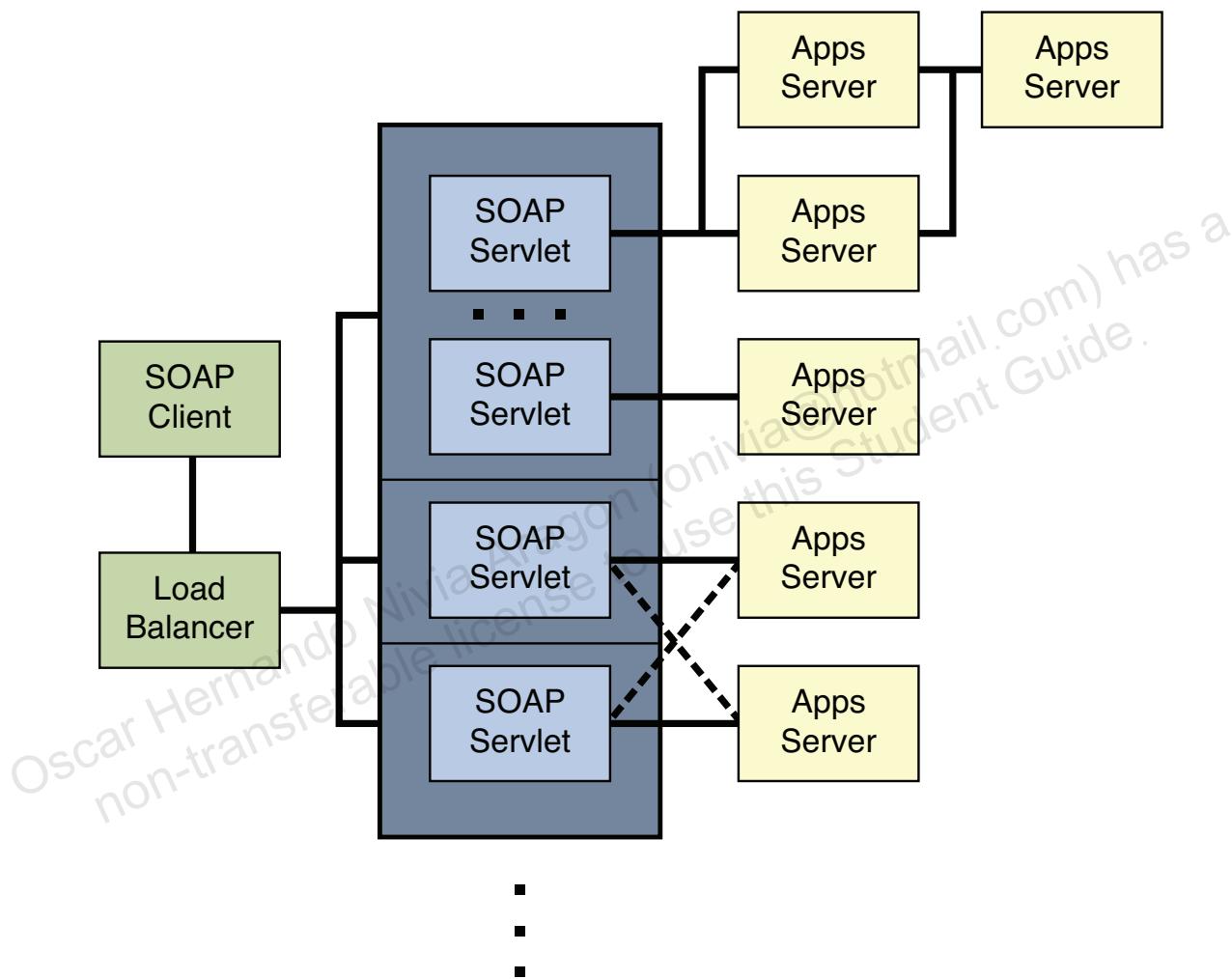


Figure 13.14: Load Balancing

Advantages and Disadvantages

The load balancing schemes share the common advantage that they make applications using such schemes more scalable as load on the application increases, you can address additional loads by adding more servers to the deployment. As load decreases, you can reduce the number of servers available.

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

The traditional disadvantage of such schemes is the overhead associated with the need to synchronize state across servers. This is not relevant to these scenarios, because web services are by definition stateless. Although, additional network traffic is required to deliver any relevant conversational state to the web service on every call, this overhead is required even if there is a single remote server.

Container Cluster

A common non-functional requirement that applications must address is availability. There are a number of common strategies to be applied to address availability concerns. Architects often decide that a set of servers running concurrently should be used for scalability purposes, using load balancing. However, they often notice that this set of servers that they had already committed to deploying for scalability purposes can also be used to deliver better availability numbers. If one of the servers in the set goes down, the application can redirect any requests that were addressed to the downed server to one of its peers, clients do not need to know that such a switch took place.

Container Cluster

- Problem** Ensure availability of application even during system failures
- Forces** Improve availability transparently
- Solution** Failover strategies in a clustered deployment

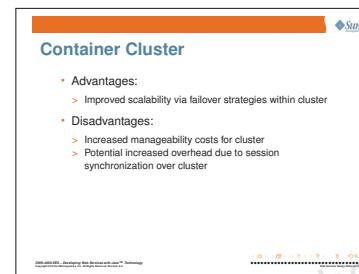
The concerns associated with introducing such a fail-over scheme into an application's architecture parallel those mentioned earlier with regards to availability. If one server is to be addressed instead of a peer that went down, it must be true that any conversational state that would have been available to the first server would be available to the peer server. That would be the only way to ensure that the client in question would receive the same answer, regardless of which server was involved. Just as before, however, this concern might not be relevant when dealing with web services, because the web service is by definition stateless. There should not be any state in the server that a client relies on, across invocations. Any state that a client needs to have available to process a

request, the client should have supplied as part of that same request. Therefore, in principle, there would be no problem in addressing that request to a server other than the one used previously.

Advantages and Disadvantages

Clustering web service endpoints across a cluster can improve the availability of the services, because failure of some of the servers in the cluster results in redirecting any clients that might have been affected to some of the remaining servers.

Assuming that the web services are indeed stateless, the conventional concern (session synchronization) that is associated with setting up such a cluster is not an issue. However, manageability of the cluster is still more expensive than manageability of individual servers.



Web Services Deployment Patterns

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Oscar Hernando Nivia Aragon (onivia@hotmail.com) has a
non-transferable license to use this Student Guide.

Chapter 14

Best Practices and Design Patterns for JAX-WS

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
- Describe best practices associated with exception handling in web services.

Objectives

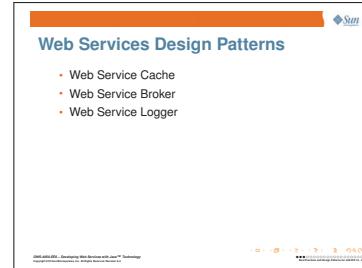
On completion of this module, you should be able to:

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

Java EE 6 API - Developing Web Services with Java™ Technology

Web Services Design Patterns

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



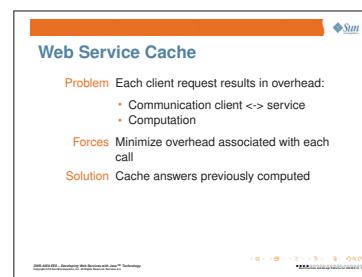
Web Service Cache

The web services framework is just one of several frameworks that an architect can choose to use, to turn the design of an enterprise application into a distributed design. The reason architects are often interested in so doing has to do with quality of service concerns, such as how to provide an application with enough resources to deal with the expected load the application will must manage. Distributing the enterprise application over a cluster of machines allows the architect to enlist the resources of all the nodes in the cluster in service to the application.

Unfortunately, introducing a distributed framework into a design is not without its drawbacks. For example, to have all nodes in a cluster collaborate to compute the answer to a query, these nodes must communicate with each other to deliver requests and its arguments in one direction, and results in the other. These communications occur over the network that ties the nodes in the cluster together. However, communications over this network incur overhead due to limitations in network bandwidth or latency, or due to encoding and decoding overhead, when the web services framework sends and receives messages across applications.

Even if you ignore the framework-induced overhead, you still have to deal with the increasing load that the clients place over the application as time goes on. However, architects observe that the actual load that clients place on an application is actually greater than it could be, in principle, because clients often issue requests that duplicate work that has already been done.

The idea behind caching is that the application builds a data structure, the cache, that records the answers to requests already computed. When a second client issues a request that has already been processed, and when the answer for the second client is the same as the one that was computed for the first client, then the application can



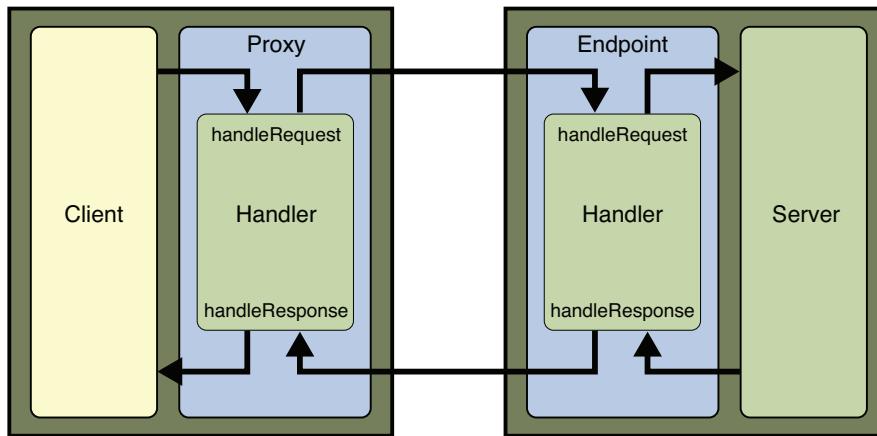
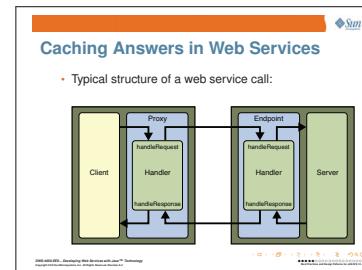


Figure 14.1: WS Handlers

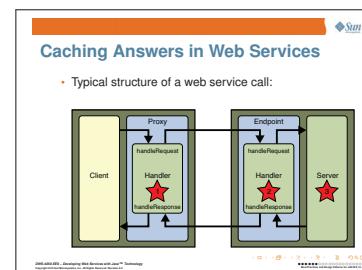
retrieve that answer from this cache. Rather than incur the penalties associated with both communications with the server over the network, and computation of the proper response once on the server, the application client can build a cache, and retrieve answers from that cache when appropriate.

The Web Service Cache pattern describes ways in which an application can build on the web services framework an application-independent transparent cache. It is based on the observation that, when the web services framework describes the machinery allowing a client to request that a service invoke an operation on behalf of that client, it allows for the presence of handlers.

These handlers exist on either side of a web-services interaction. They can examine requests as they leave the client, or as they arrive at the server, and execute logic before the request is actually processed by the server-side service provider, as illustrated in Figure 14.1.



Given this framework, the Web Service Cache pattern argues that there are two places where you can introduce a cache that is transparent to the application, but yet would reduce overhead by short-circuiting requests that do not need to be executed again, as illustrated in Figure 14.2. And, if an application-specific cache was more appropriate, it could be accommodated, as well.



Web Services Design Patterns

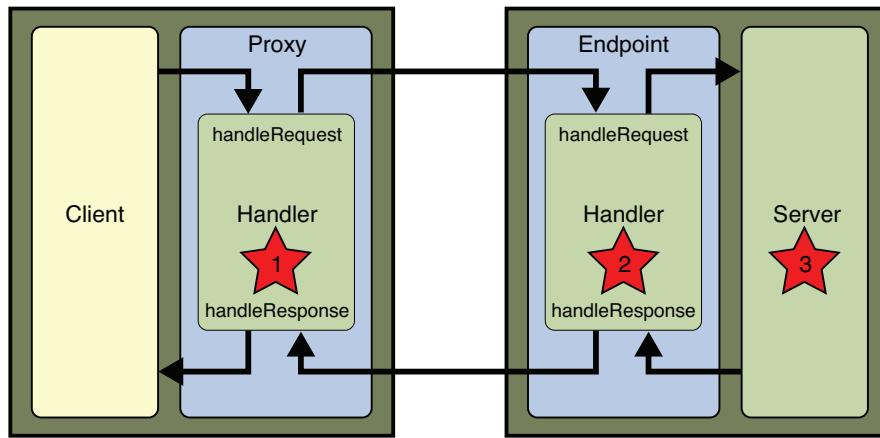


Figure 14.2: Opportunities for Caching

Advantages and Disadvantages

The advantage of introducing the Web Service Cache pattern is that it minimizes the overhead associated with executing operations.

In the context of a web service, there are two sources of overhead to consider:

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

Java EE 6 API - Developing Web Services with Java™ Technology

- Communications overhead, which includes both the cost of actually transmitting information over the network and the cost associated with encoding and decoding the messages that are sent over the network according to the protocol used by the web service, typically XML.
- Processing overhead to actually execute the client's request.

Depending on the semantics required by the application, some caching can occur on one side of the network, some on the other. However, every time the cache is used, you reduce the cost of satisfying a request.

The disadvantages of introducing the Web Service Cache into a design are also those associated with any caching scheme:

- The application must realize when to invalidate or refresh the cache. Sometimes, it is necessary to bypass the cache and actually perform the operation one more time. However, the logic to determine whether that is the case, on each call, could be complex to write, and expensive to run.

- Increases memory footprint to hold cached information. In practice, introducing a caching scheme into a design is useful when the likelihood of the cache being used, is high. It is a space and time trade-off.

Web Service Broker

Web services provide an attractive mechanism to support the integration of Java EE and non-Java EE peers in a distributed application. The web services framework can take care of translations that might be necessary for any interaction to take place. However, it is important to recognize the limitations implicit in the web services framework, so as to recognize when they would hinder rather than help the service to be implemented. You can use the Web Services Broker pattern to implement some services as a web service, and still address the concern that web services do not propagate transactions, at least not portably.

An important question to be considered while offering a service is whether the candidate service is to be coarse-grained or fine-grained. One of the characteristics that determines this difference is whether the service can only be used as a self-contained operation and so could manage its own transaction internally, or whether the service might need to be a participant in a larger, client-initiated transaction, in which the service commits or rolls back as part of that larger transaction. The first of these two scenarios is a natural one for a web services implementation; the second one could be a problem, since it might not be possible to propagate the client's transaction into the web service implementation of our candidate service, thus making it impossible for that candidate service to integrate into the larger transaction.

There is a simple solution, if there are other valid reasons to implement your candidate service as a web service. There are extensions to the basic web services framework, such as the Web Services Atomic Transaction (WS-AT) extension, that allow a web service to join its client's transaction. One could simply choose a framework that already implements this extension.

The Web Services Broker pattern proposes an alternative implementation that would emulate the behavior intended, with just the basic web services machinery. Consider a simple design of an application that wants to integrate two remote services into its own workflow. Such a simple design assumes that a transaction begun by the client automatically propagates to both remote services. This allows both remote services to join that same transaction. Once

 **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

Downloaded from www.oreilly.com/catalog/webservicesguide

Web Services Design Patterns

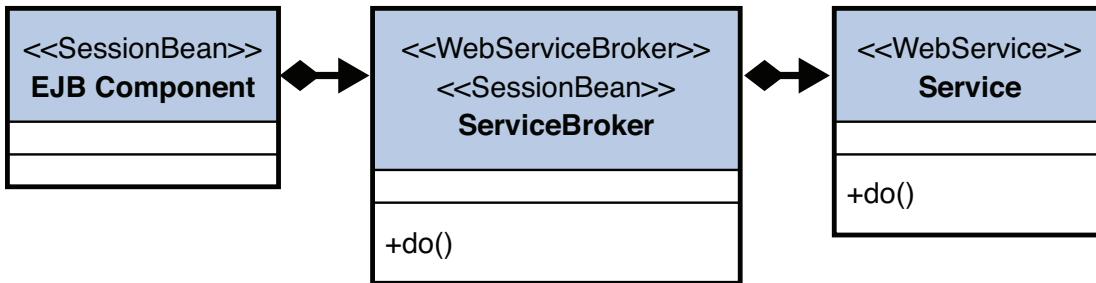


Figure 14.3: Web Service Broker

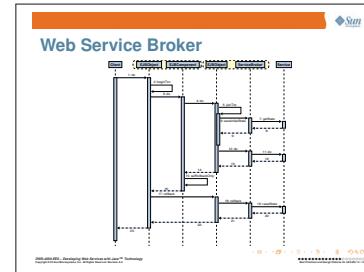
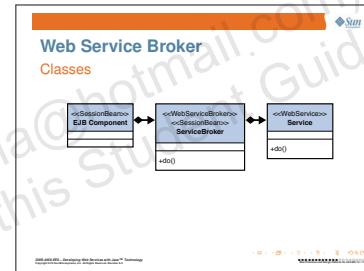
they have done so, a failure during processing in the second remote service allows it to roll back its own state by rolling back the current transaction. However, this automatically rolls back the work done by the first remote service, and any work done by the client, during that same transaction.

The question to answer is, how can you achieve the same effect, if it is impossible to propagate the client's transaction to any remote services? Figure 14.3 illustrates the strategy proposed by the web service broker pattern. If the remote service cannot join the client's transaction, then the web service broker is introduced to support the transactional behavior that is desired. This web service broker is introduced as a middle-man, between the client and the remote service in which the client is interested in.

The broker is defined as a stateful session bean to:

- Join the transaction that the client had already initiated, at the time the client calls the broker.
- Keep the initial state of its associated web service, in case a rollback is needed.

It is the broker's responsibility to record any relevant state associated with the remote service, before that service is used by the client within the transaction. This would be the remote service's initial state. The broker then propagates any requests that the client makes of the broker across to the remote service. Because the broker is a participant in the client's transaction, it is notified when that transaction attempts to commit, commits successfully, or rolls



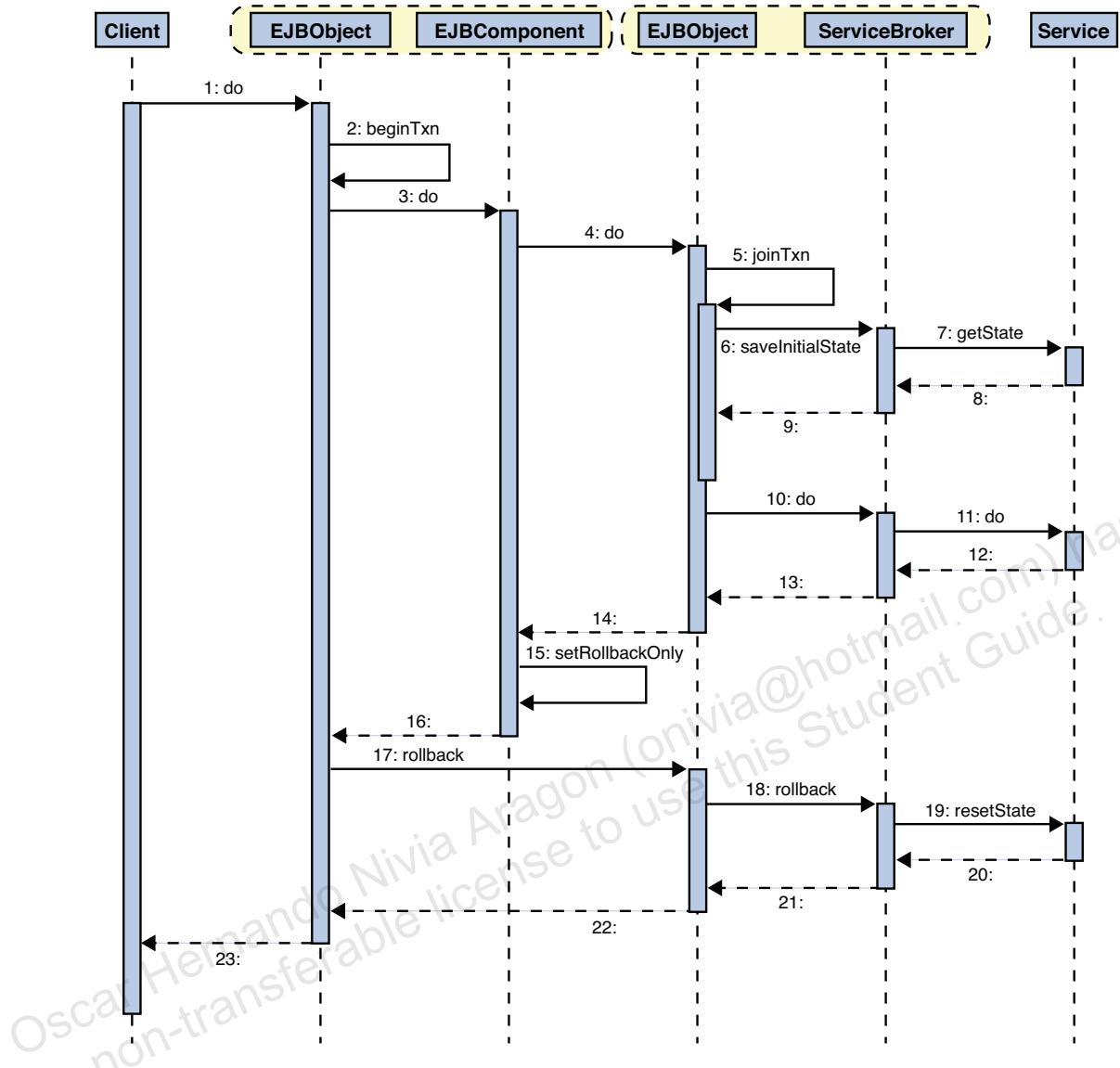


Figure 14.4: Web Service Broker Interaction

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

back. And because the broker saved that initial state, it can also implement explicitly any logic that might be needed to actively return the remote service to that initial state, if a roll back becomes necessary. The remote service initiates a compensating transaction internal to that service, controlled by the web service broker, at that time. Figure 14.4 illustrates an interaction that results in a transaction rollback.

Advantages and Disadvantages

The advantage associated with the web service broker pattern is that it allows the application to preserve the simpler design associated with the idea that all participants in an atomic operation can join the corresponding transaction, and delegate to the transactional engine the correct operation.

Web Service Broker

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

The drawback associated with the pattern is that it assumes that the web service broker can both save an initial state for the remote web service before using it in an operation and return that remote web service to that same initial state using a compensating transaction. However, this is not guaranteed under all circumstances.

Web Service Logger

Applications often need to maintain operational logs, for example, to track operations performed for security purposes. To improve maintainability and flexibility for the application, architects should incorporate logging functionality in such a way that the machinery used to support logging would be orthogonal to the core functionality offered by the application.

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

A common approach to introduce abilities, such as logging, into the design of an application involves the application of the Decorator pattern. In the Decorator pattern, an additional object is introduced as a wrapper around the actual service provider, and the logging functionality is captured in this wrapper. A different approach is possible in the web services world, by taking advantage of the functionality built into the web services framework.

The web services infrastructure includes mechanisms to extend the machinery responsible for processing requests, using SOAP handlers. These are objects that can be configured to examine web service requests as they make their way from the client who initiates a request to the web service endpoint responsible for processing the request. Logging capabilities are easy to build into a web service interaction using this infrastructure.

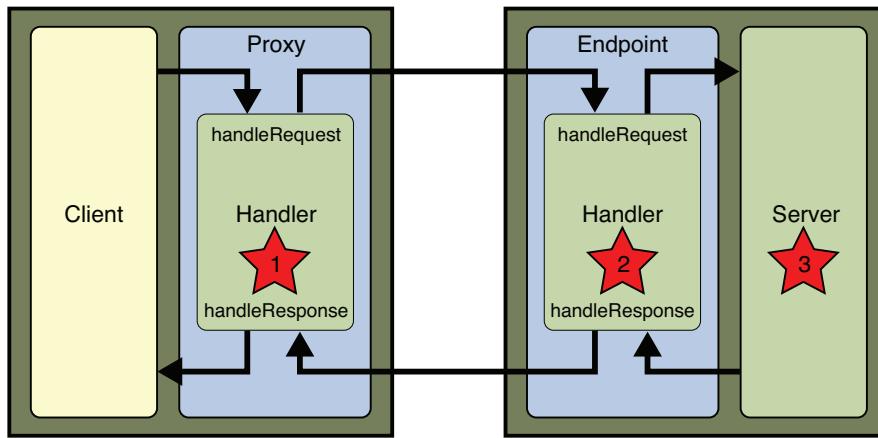
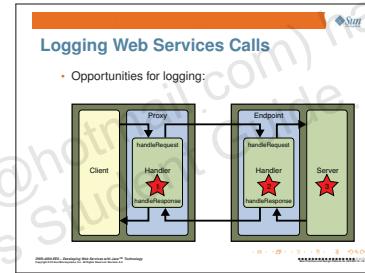


Figure 14.5: Opportunities for Logging

Figure 14.5 illustrates where there are opportunities to implement a logger, within the web services framework. In the figure, 1 and 2 are points in the processing where a SOAP handler could be inserted to provide logging functionality, while 3 corresponds to the introduction of logging functionality as part of the provider's application-level processing.



SOAP handlers are objects that must implement the JAX-WS Handler interface. They must provide a `handleMessage` method, which the JAX-WS framework invokes to give the handler an opportunity to examine the outgoing or incoming messages. Although handlers are expected to examine and perhaps modify messages and their headers before passing them on, a handler can terminate processing of a particular message. In such a case, the handler is responsible for generating the proper response.

Handlers are involved in the processing of messages in both directions. A client handler examines the outgoing message first, and then later examines the incoming response. A server handler examines the incoming message first, and then later examines the outgoing response. However, these opportunities are issued using a call to the `handleMessage` method provided by the handler. To distinguish between the two cases, handlers can use a standard property in the `MessageContext` argument to `handleMessage`. A standard property `javax.xml.ws.handler.message.outbound` has a boolean value to identify the case.

Web Services Design Patterns

```

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

Figure 14.6: Sample Service Using Handlers

```

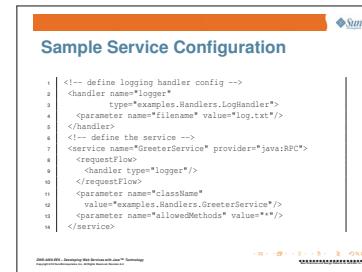
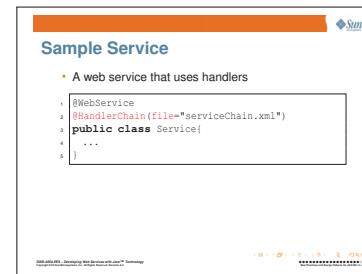
1 <!-- define logging handler config -->
2 <handler name="logger"
3         type="examples.Handlers.LogHandler">
4     <parameter name="filename" value="log.txt"/>
5 </handler>
6 <!-- define the service -->
7 <service name="GreeterService" provider="java:RPC">
8     <requestFlow>
9         <handler type="logger"/>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

Figure 14.7: Handler Configuration

Multiple handlers can be associated with a particular web service. This handler chain is typically specified in JAX-WS declaratively through an annotation, as illustrated by Figure 14.6.

You can specify the XML file referenced by the HandlerChain annotation as an absolute URL, or as a path relative to the class path for the web service class. The content of this file is illustrated by Figure 14.7.

Handlers in JAX-WS can be of two types, logical handlers or SOAP handlers. The key difference is that a logical handler is given access to the payload of the message alone, while a SOAP handler is given access to the raw SOAP message, including its headers. There are two interfaces defined in JAX-WS, LogicalHandler



and SOAPHandler. Application-level programmers choose what kind of handler they mean to define by the type of interface that they choose to implement. An additional constraint built into the specification is that logical handlers always process outgoing messages before any SOAP handlers, but logical handlers always process incoming messages after any SOAP handlers.

Advantages and Disadvantages

The ability to specify tasks, such as logging in such a way that they are decoupled from normal application-level logic leads to more flexible and maintainable application designs. Such cross-cutting concerns can be introduced using these SOAP handlers transparently, reinforcing the principle of separation of concerns. Deploying these handlers requires a bit of additional configuration, but the alternative of integrating such logic into the application-level service provider is worse.

The screenshot shows a slide titled 'Web Service Logger' from the 'JAX-WS Best Practices and Design Patterns' presentation. The slide has a blue header bar with the Sun logo. Below the title, there are two sections: 'Advantages:' and 'Disadvantages:'. The 'Advantages:' section lists three items: 'Decoupling logging from other application-level concerns leads to more maintainable designs', 'Strategy can be used to introduce other "cross-cutting" concerns', and 'Additional configuration not evident in code is needed to incorporate logger into application'. The 'Disadvantages:' section is currently empty. At the bottom of the slide, there is a footer with the text 'JAX-WS-EE6 - Decoupling the Service and Java™ Platform' and navigation icons for back, forward, and search.

Handling Exceptions in Web Services

A web service application can encounter two types of error conditions. One type of error might be an irrecoverable system error, such as an error due to a network connection problem. When an error such as this occurs, the JAX-WS runtime on the client throws the client platform's equivalent of an irrecoverable system exception. For Java clients, this translates to a `WebServiceException` or a subclass of `ProtocolException`.

A web service application can also encounter a recoverable application error condition. This type of error is called a service-specific exception. The error is particular to the specific service.

To illustrate the web service exception-handling mechanism, we will examine it in the context of the weather web service example. When designing the weather service, you want the service to be able to handle a scenario in which the client requests weather information for a non-existent city. You might design the service to throw a service-specific exception, such as the domain-level `CityNotFoundException`, to the client that made the request. Service-specific exceptions must be checked exceptions that directly or indirectly extend `java.lang.Exception`. They cannot be unchecked exceptions. Code 14.1 shows a typical implementation of a service-specific exception, such as for `CityNotFoundException`.

Code 14.2 shows the service implementation for the same weather service interface. This example illustrates how the service might throw an exception of type `CityNotFoundException`.

Remember the following while handling exceptions in the web service:

- Convert application-specific errors and other Java exceptions into meaningful service-specific exceptions and throw these service-specific exceptions to the clients.

```
1 public class CityNotFoundException extends Exception {  
2     private String message;  
3     public CityNotFoundException(String message) {  
4         super(message);  
5         this.message = message;  
6     }  
7 }
```

Code 14.1: Implementation of the Service-Specific Exception

```

1  public class WeatherService {
2      public String getWeather(String city)
3          throws CityNotFoundException {
4              if (!validCity(city))
5                  throw new CityNotFoundException(city+"_not_found");
6                  // Get weather info and return it back
7                  return (getWeatherInfo(city));
8          }
9      }

```

Code 14.2: Example of a Service Throwing a Service-Specific Exception

- Exception inheritances are lost when you throw a service-specific exception.
- The exception stack trace is not passed to the client.

Handling Exceptions in Web Service Client

Two types of exceptions occur for client applications that access web services: system exceptions and service-specific exceptions, which are thrown by a service endpoint. System exceptions are thrown for such conditions as passing incorrect parameters when invoking a service method, service inaccessibility, network error, or some other error beyond the control of the application. Service exceptions, which are mapped from faults, are thrown when a service-specific error is encountered. The client application must deal with both types of exceptions.

System exceptions generally involve unanticipated errors, such as a network timeout or an unavailable or unreachable server, that occur when invoking a web service. Although system exceptions are usually out of the control of the client developer, you should still be aware that these errors can occur, and you should have a strategy for your client application to deal with these situations. For example, with these types of errors, it might be useful to have the client application prompt the user to retry the operation or redirect the user to an alternative service, if possible. Many times the only solution is to display the exception occurrence to the end user. Sometimes it might be appropriate to translate the system exception to an unchecked exception and devise a global way to handle these exceptions. The exact solution is particular to each application and situation.

System exceptions, which the client receives as a `WebServiceException` or a subclass of `ProtocolException`, arise for a variety of reasons. Often system ex-

Handling Exceptions in Web Services

ceptions happen because of network failures or server errors. They also can be the result of a SOAP fault. Because a `WebServiceException` or the subclass of `ProtocolException` usually contains an explanation of the error, the application can use that message to provide its own error message to the user and can prompt the user for an appropriate action to take. If an EJB component client is doing its work on behalf of a Web tier client, the EJB client should throw an exception to the Web tier client. The Web tier client notifies the user, giving the user a chance to retry the action or choose an alternative action.

Web components can leverage the servlet error messaging system and map the `WebServiceException` or the subclass of `ProtocolException` to some exception through handling instructions in the `web.xml` file. The client can throw its own `javax.servlet.ServletException` and can display a general application error message. You can extend the `ServletException` to create a set of application-specific exceptions for different errors. It might also use the file to define a specific error JSP to handle the error.

Although they do not directly interact with the user, workflow clients implemented as EJB components can benefit from transactional processing, particularly if you are using container-managed transactions. However, because you cannot assume that the web service is also transactional, you might have to manually back out some of your changes if other web services are involved in the processing. Keep in mind that backing out changes can be difficult to accomplish and is prone to problems.

A Java EE component that receives a `WebServiceException` or a subclass of the `ProtocolException` when accessing a service can retry connecting to the service a set number of times. If none of the retries are successful, then the client can log an error and quit the workflow. Or, the client can forward the unit of work to another process in the workflow and let that component access the same service at a later point or use a different service.

Service exceptions occur when a web service call results in the service returning a fault. A service throws such faults when the data presented to it does not meet the service criteria. For example, the data might be beyond boundary limits, it can duplicate other data, or it might be incomplete. These exceptions are defined in the service's WSDL file as operation elements, and they are referred to as `wsdl:fault` elements. These exceptions are checked exceptions in client applications. For example, a client accessing the order tracking service can pass to the service an order identifier that does not match orders kept by the service. The client can receive an `OrderNotFoundException`, because that is the error message defined in the WSDL document, as shown in Code 14.3.

You can use the JAX-WS tools to map faults to Java objects. These tools generate the necessary parameter mappings for the exception classes and generate the necessary classes for the mapping. Generated exceptions classes extend

```
1 <fault name="OrderNotFoundException"
2   message="tns:OrderNotFoundException" />
```

Code 14.3: Error Message Defined in WSDL Document

java.lang.Exception. The client application developer is responsible for catching these checked exceptions in a try/catch block. The developer should also try to provide the appropriate application functionality to recover from such exceptions. For example, an order tracking client might include the code shown in to handle cases where a matching order is not found. The order tracking service threw an OrderNotFoundException, and the client presented the user with a GUI dialog indicating that the order was not found.

A Java EE web component client can handle the exception using the facilities provided by the Java EE environment. The client can do one of a couple of things: it can wrap the application exception and throw an unchecked exception, such as a javax.servlet.ServletException; or it can map the exception directly to an error page in the web deployment descriptor.

Exception Management in Web Services

Exception management in web services requires more consideration than for standalone applications.

Standalone applications are monolithic and are typically managed by a single business. The changes to the software are controlled centrally. It is also assumed that the application can generate only limited exceptions and these are well-defined.

Applications based on web services are typically distributed and might be composed of services running on heterogeneous platforms. Many users can concurrently share the constituting web services. When an exception occurs in a federation of web services, it is usually localized and does not give any useful information to the client. If the server maintains a log of exception events, it might not help the administrator to elaborate the business exception because the log can only give a cause of localized exception.

Therefore, you need a more elaborate exception management system to manage exceptions in web services, especially if the application contains federated web services.

The success of a web service design largely depends on the adoption of an appropriate exception management strategy. A good exception management

Handling Exceptions in Web Services

strategy in web service design should consider the following:

- Detect all the possible exceptions in a web service
- Keep a log of all error conditions
- Abstract logging of error conditions from the business logic
- Enable monitoring of various events in web service interactions

Some of the best practices for handling web service exceptions are:

- Use predefined exceptions classes in most of the web service implementations, such as:
 - SOAPFaultException for SOAP-based services; it is wrapper for a SOAP 1.1 or 1.2 fault.
 - HTTPException for REST-style services; it carries a simple HTTP status code.
- Use helper methods to avoid excessive code in exception handling as building a SOAPFaultException and its SOAPFault is tedious.
- Create custom web service exception classes that extend the JAX-WS exception WebServiceException.

Chapter 15

Best Practices and Design Patterns for JAX-RS

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

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

The screenshot shows a presentation slide with a red header bar containing the text 'Objectives'. Below the header, there is a section titled 'Objectives' with the following text: 'On completion of this module, you should be able to: • recognize and apply best practices associated with implementing web services using JAX-RS.' At the bottom right of the slide, there is a small footer with icons for navigation and a copyright notice: '2014-2015 APAC - Developing Web Services with Java™ Technology' and 'Copyright © 2014, Oracle and/or its affiliates.'

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

RESTful Best Practices

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

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

- 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

Table 15.1 shows examples of several resource ids in the Traveller application, and what the intent behind issuing specific HTTP methods with those resource ids might be.

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

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

Table 15.1: HTTP Action Semantics – Examples



This example taken from:

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

- 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

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?

Java REST API - Designing Web Services with Java™ Technology

- 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

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...

Java REST API - Designing Web Services with Java™ Technology

- 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.

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:

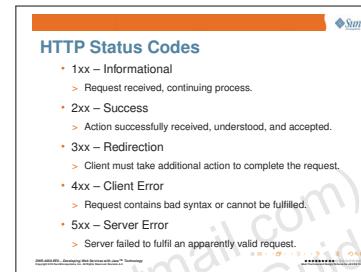
`PUT /airports/clientGeneratedId`
 where `clientGeneratedId` can be used to disambiguate requests.

Java REST API - Designing Web Services with Java™ Technology

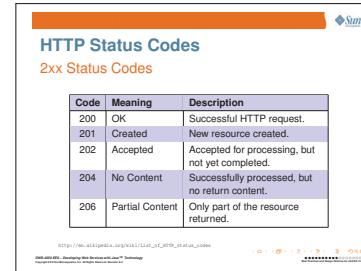
Code	Meaning	Description
200	OK	Successful HTTP request.
201	Created	New resource created.
202	Accepted	Accepted for processing, but not yet completed.
204	No Content	Successfully processed, but no return content.
206	Partial Content	Only part of the resource returned.

Table 15.2: HTTP Status Codes – 2xx

- 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 fulfill an apparently valid request.



Status codes in the 200-range indicate success. The body section, if present, is the object returned by the request. It is a MIME-encoded object, and may only be in `text/plain`, `text/html` or one of the formats specified as acceptable in the request.



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.

Table 15.3: HTTP Status Codes – 3xx

Status codes in the 300-range indicate action to be taken (normally automatically) by the client in order to fulfill the request.

The screenshot shows a table titled "HTTP Status Codes" under the heading "3xx Status Codes". The table has three columns: "Code", "Meaning", and "Description". The rows correspond to the four status codes listed in the table above: 300 (Multiple Choices), 301 (Moved Permanently), 303 (See Other), and 304 (Not Modified). Each row contains a brief description of the code's meaning and how it should be handled by a client.

The list of status codes is drawn from Wikipedia, at:

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

The W3C organization maintains a list, too, at:

<http://www.w3.org/Protocols/HTTP/HTRESP.html>.



Status codes in the 400-range are intended for cases in which the client seems to have erred, and the codes in the 500-range for the cases in which the server is aware that the server has erred. It is impossible to distinguish these cases in general, so the difference is only informational.

The screenshot shows a table titled "HTTP Status Codes" under the heading "4xx Status Codes". The table has three columns: "Code", "Meaning", and "Description". The rows correspond to the four status codes listed in the table above: 400 (Bad Request), 401 (Unauthorized), 403 (Forbidden), and 404 (Not Found). Each row contains a brief description of the code's meaning and how it should be handled by a client.

The body section may contain a document describing the error, typically in human readable form. The document is in MIME format, and may only be in text/plain, text/html or one of the formats specified as acceptable in the request.

- Use nouns for resources, rather than verbs.
 - Resources are “things”, not “actions”.
- Minimize the use of query strings.

The screenshot shows a section titled "URI Best Practices" under the heading "URI". It contains a bulleted list of 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”.

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.
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.
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".

Table 15.4: HTTP Status Codes – 4xx

- Slashes – / – in URIs denote parent/child or whole/part relationships.
- Aim for “progressive disclosure”.



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

- **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.

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.

[Navigation icons]



Taken from:

<http://www.xml.com/pub/a/2004/08/11/rest.html>

- 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

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
 - client initiates request
 - server returns a list of available representations
 - client selects representation and sends a second request to the server.
- URI-specified representation

[Navigation icons]



Taken from: <http://www.xml.com/pub/a/2004/08/11/rest.html>



- 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.

The screenshot shows a presentation slide with a blue header bar containing the Sun logo. The main title is "Request Validation". Below it is a bulleted list:

- 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.

At the bottom of the slide, there is a URL: <http://blog.feedly.com/2009/05/06/best-practices-for-building-json-rest-web-services/>. The slide has a standard presentation footer with navigation icons.

Taken from:

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

Roy T. Fielding, the author of the original dissertation introducing the REST architecture, proposed that architects consider the following rules, before calling their creation a REST API:

- A REST API should not be dependent on any single communication protocol, though its successful mapping to a given protocol may be dependent on the availability of metadata, choice of methods, etc. In general, any protocol element that uses a URI for identification must allow any URI scheme to be used for the sake of that identification.
Failure here implies that identification is not separated from interaction.
- A REST API should not contain any changes to the communication protocols aside from filling-out or fixing the details of underspecified bits of standard protocols, such as HTTP's PATCH method or Link header fields. Workarounds for broken implementations (such as those browsers stupid enough to believe that HTML defines HTTP's method set) should be defined separately, or at least in appendices, with an expectation that the workaround will eventually be obsolete.
Failure here implies that the resource interfaces are object-specific, not generic.
- A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types. Any effort spent describing what methods to use on what URIs of interest should be entirely defined within the scope of the processing rules for a media type (and, in most cases, already defined by existing media types).
Failure here implies that out-of-band information is driving interaction instead of hypertext.
- A REST API must not define fixed resource names or hierarchies (an obvious coupling of client and server). Servers must have the freedom to control their own namespace. Instead, allow servers to instruct clients on how to construct appropriate URIs, such as is done in HTML forms and URI templates, by defining those instructions within media types and link relations.
Failure here implies that clients are assuming a resource structure due to out-of-band information, such as a domain-specific standard, which is the data-oriented equivalent to RPC's functional coupling.
- A REST API should never have typed resources that are significant to the client. Specification authors may use resource types for describing server implementation behind the interface, but those types must be irrelevant and invisible to the client. The only types that are significant to a client are the current representations media type and standardized relation names.

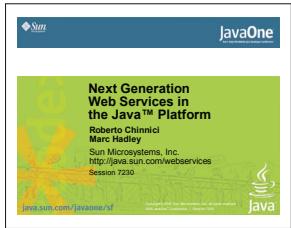
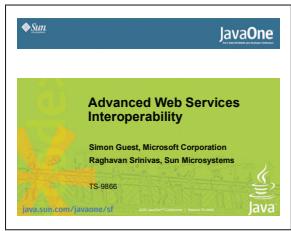
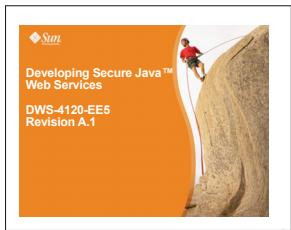
-
- A REST API should be entered with no prior knowledge beyond the initial URI (bookmark) and set of standardized media types that are appropriate for the intended audience (i.e., expected to be understood by any client that might use the API). From that point on, all application state transitions must be driven by client selection of server-provided choices that are present in the received representations or implied by the users manipulation of those representations. The transitions may be determined (or limited by) the clients knowledge of media types and resource communication mechanisms, both of which may be improved on-the-fly (e.g., code-on-demand). Failure here implies that out-of-band information is driving interaction instead of hypertext.

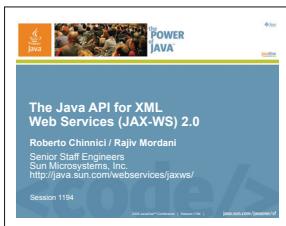


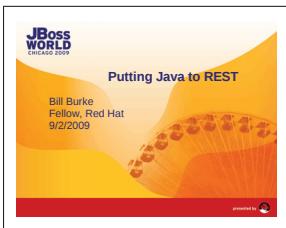
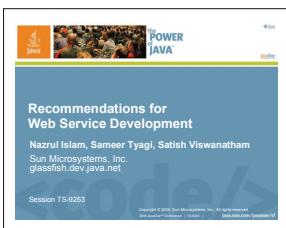
This section is taken from Roy Fielding's blog, at:

<http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>.

References







Oscar Hernando Nivia Aragon (onivia@hotmail.com) has a
non-transferable license to use this Student Guide.

Appendix A

XML Schema

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 advances features in XML Schemas
 - Use mixed content
 - Use empty elements
 - Use annotations
 - Define namespaces
 - Include other XML schema documents

The screenshot shows a presentation slide with a blue header bar containing the Sun logo. The main title is 'Objectives'. Below it, a bulleted list details what should be learned: 'On completion of this module, you should be able to:' followed by a list of XML Schema features. At the bottom right, there are navigation icons for a presentation slide.

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 advances features in XML Schemas
 - > Use mixed content
 - > Use empty elements
 - > Use annotations
 - > Define namespaces
 - > Include other XML schema documents

Additional Resources

Additional Resources

The following references provide additional information on the topics described in this module:

- XML Schema Part 0: Primer, at
<http://www.w3.org/TR/xmlschema-0/>, accessed 16 March 2002.
- van der Vlist, Eric, Using W3C XML Schema, on O'ReillyXML.com, at
<http://www.xml.com/pub/a/2000/11/29/schemas/part1.html>, accessed 17 October 2001.
- Walmsley, Priscilla, Definitive XML Schema. Saddle River: Prentice Hall, 2002.
- Merriam-Webster Dictionary Online, [<http://www.m-w.com/>], accessed 16 March 2002.

Creating a Basic Schema

The W3C XML Schema Definition Language, or schemas, is an XML language for describing and defining the content of XML documents. This standard was released by the W3C on May 2, 2001. The current specification can be found at the <http://www.w3.org/XML/Schema> Web site.

Websters Dictionary defines schema as a structured framework or plan. Essentially, that is what schemas are for, defining the framework of an XML document. The schema standard is much more powerful than the DTD language you have been using in this class. Schemas have a number of advantages over DTDs. They include:

- Schemas are defined in XML. You do not have to learn a new language to use them.
- Schemas include real data types, such as integers or floating point numbers. With a schema, you can actually check the data in your elements and attributes.
- Schemas work well with namespaces and were, in fact, designed from the ground up to work with them.
- Schemas take a more object-oriented approach to element and attribute definitions. Reuse of definitions is an easier task with schemas.

- 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

The screenshot shows a presentation slide with the title "Building a Schema" in bold. Below the title is a bulleted list of eight items. At the bottom right of the slide, there is a small navigation bar with icons for back, forward, and search.

- 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

Schemas use XML to define the structure of a document. Therefore, the file must be well-formed, just as any other XML document should. A number of XML tags that have been defined for schemas allow you to define the structure of document.

Figure A.1 shows a sample XML schema. The elements in that schema can be described as follows:

1. The `schema` element is the root element for a schema document. It defines the schema namespace and prefix. The prefix is used in all schema documents to identify elements that belong to the schema application.
2. This `element` tag declares the `people` element. This is the root element for this XML document. You can define subelements within an `element` tag.
3. The `complexType` element defines the structure and order of the `people` element. This `complexType` element indicates that the `people` element contains one or more elements. The structure of the data is complex in that it does not contain a simple type, such as `string`. In this example, the `complexType` includes one additional element definition.

```

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

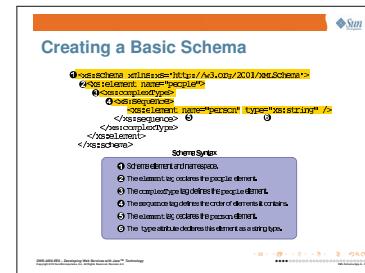


Figure A.1: Creating a Basic Schema

4. The sequence element defines the order of the elements it contains. This tag is called a *compositor* because it groups elements together. As the element name implies, elements within this tag are defined in a sequential order.
5. This element tag declares the person element. This element is considered a *simple* type because it is composed of the string schema data type.
6. The string data type is one of a number of data types that can be used with XML schema.

Embedding the XML schema for a document within the document proper can quickly turn unwieldy. One can link an XML document to its schema, instead, by first declaring an XML Schema Instance namespace, which is identified in this example by the prefix xsi. Then, you use the noNamespaceSchemaLocation attribute in that namespace to define the location of the schema file. The XML parser recognizes these attributes and validates an XML document based on the values set here.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- Pet club people file -->
①<people xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    ② 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.

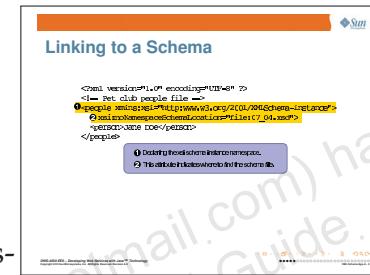
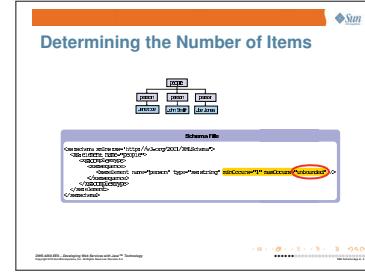


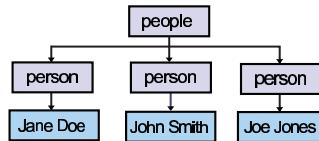
Figure A.2: Linking to a Schema

The number of times an element occurs in an XML document is determined by the minOccurs and maxOccurs attributes. In the example in Figure A.3, the person element must occur once, but has no upper limit on the number of times it can show up in the document.

The default value for both attributes is 1. Therefore, if no value is specified, only one element may appear in a document.



Creating a Basic Schema



Schema File

```

<xs:schema xmlns:xs='http://www.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>
  
```

Figure A.3: Determining the Number of Items

The schema document shown in Figure A.4 defines the name and pet elements. To do this, the person element must be converted from a simple type to a complex type. First, apply both the complexType and sequence elements to the person element in the same way they were applied to the element. Then, define the name and pet elements as simple types.

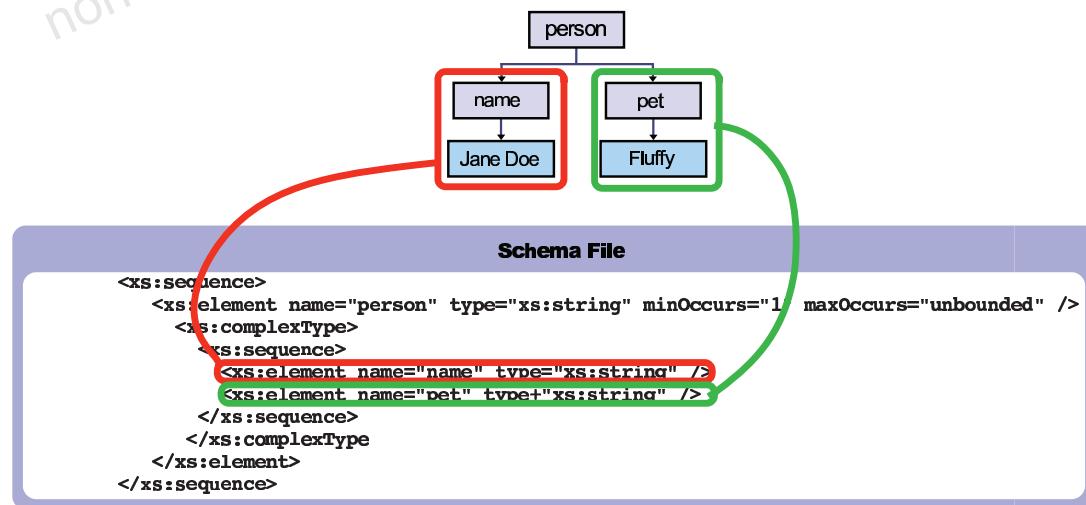
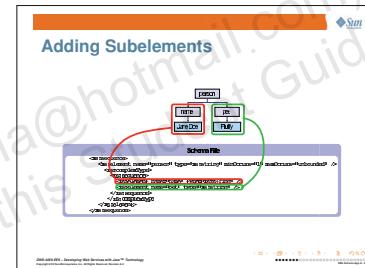


Figure A.4: Adding Subelements

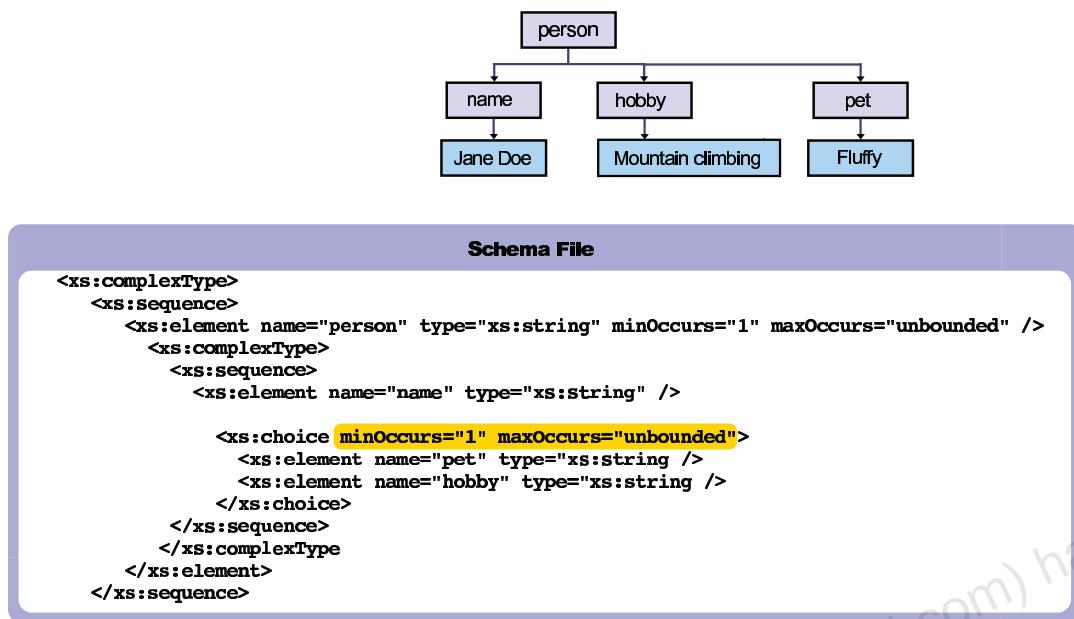
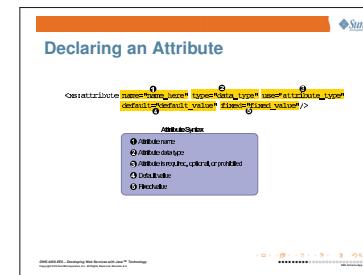
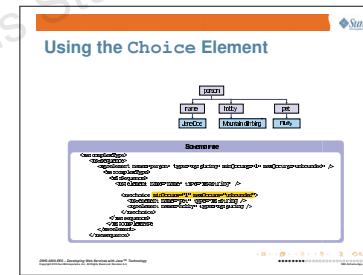


Figure A.5: Using the Choice Element

To this point, the schema has only defined elements in a sequential order. You can use the choice element to define elements in any order. The choice element functions much like the choice operator in DTDs. Setting the maxOccurs attribute to unbounded essentially allows any number of elements in any order.

Declaring an attribute element in a schema file is straightforward. Use the attribute element and its attributes, shown in Figure A.6, to determine the attributes name, content type, use, and default value:

1. The name of the attribute.
2. The data type of the attribute. This can be any schema type. In addition, any of the attribute types presented in DTDs may also be used in this field, except CDATA.
3. Much like the attribute **default** in DTDs, the use attribute determines if an attribute is required or optional. The prohibited value is only used when you are restricting complex types.



Creating a Basic Schema

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

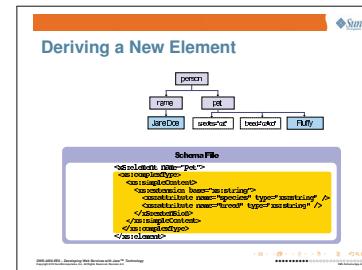
Attribute Syntax

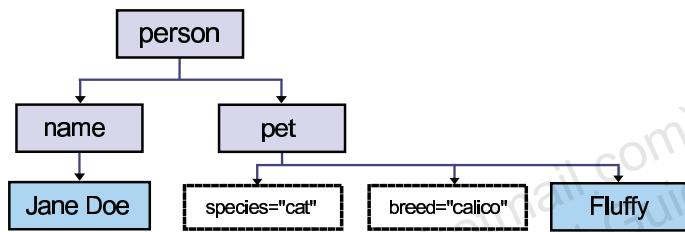
- ❶ Attribute name
- ❷ Attribute data type
- ❸ Attribute is required, optional, or prohibited
- ❹ Default value
- ❺ Fixed value

Figure A.6: Declaring an Attribute

4. Defines a default value for the attribute. This attribute can be used only with a use value of optional. The **default** and **fixed** attributes are mutually exclusive.
5. Defines a fixed value for the attribute. The attribute can be used only with the required value. The **default** and **fixed** attributes are mutually exclusive.

Although using the attribute element is straightforward, adding attributes to the pet element in this example is more difficult than one might think. By definition, a simple type cannot contain attributes. Therefore, the simple type for pet must be converted into a complex type. When an element contains only attributes and text and no child elements, you use the simpleContent element with the extension element to define the data that appears in the element. The element attributes are specified inside the extension 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>
```

Figure A.7: Deriving a New Element

Structuring XML Schemas

Using named types is the most flexible way to structure schemas. Essentially, each element or attribute definition is defined as a component within the schema document. These components are then fitted together to form a schema. Because each type is independent, types can be reused within a schema. This method of structuring schemas takes a more object-oriented approach to defining the relationships between elements and attributes.

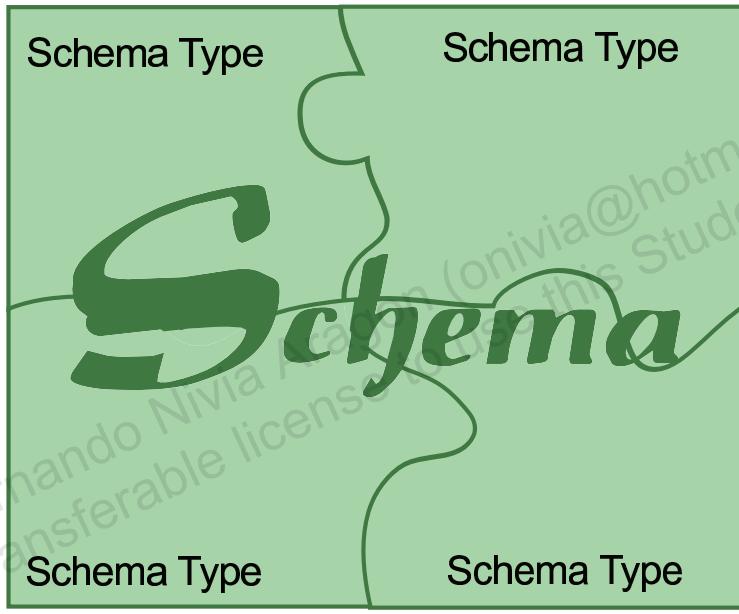
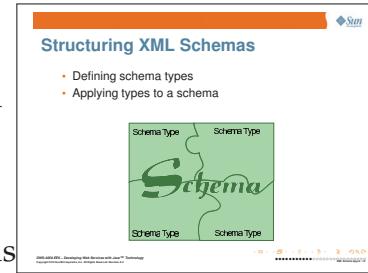
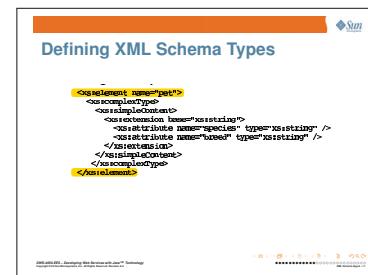


Figure A.8: Structuring XML Schemas

You must follow a couple of steps to convert from a flat schema model to a named type schema model. First, remove the element tags from the definition. Then, give the complex type a name. In this example, the pet element definition is converted into a petType definition. By convention, a schema type normally ends in the word Type.



When the pet element is declared in the personType, the element uses the type attribute to specify petType as the definition for this element. If needed, the petType type could be reused any number of times in the schema.

```

<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>

```

Figure A.9: Defining XML Schema Types

The effect of applying named types to a schema is to create building blocks. First, you assemble the individual parts, and then you snap together the individual parts to create an overall schema. This approach creates a highly extendable and flexible module for creating schemas.

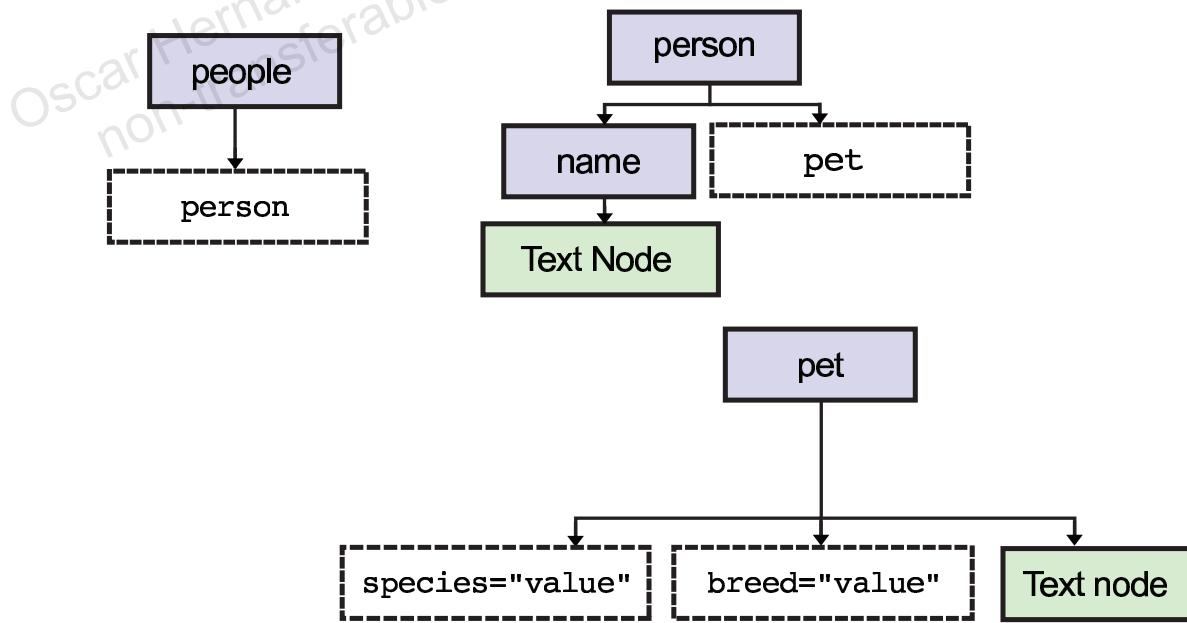
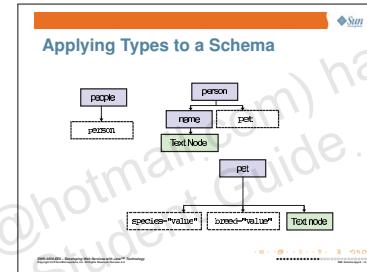


Figure A.10: Applying Types to a Schema

Using Data Types in an XML Schema

One of the more powerful features of schemas is the ability to place restrictions on data types. Restrictions are placed on data by using a *facet*. Facets set upper and lower limits on data or make data match a specific pattern. To use facets, you must redefine the simple types used so far.

So that you can examine the new simple types, the pet club member list has been updated to include some more varied data. A new schema has been created that goes with the updated file. All simple elements have been defined using the built-in string type, to begin with – but you will see how you can derive your own simple types by using facets to restricting an integer, using patterns, using enumerations, using dates, and restricting strings.

To set constraints on a simple type, define the new type in a manner similar to the way you would define a complex type. There are three important parts to each definition:

- The type definition that starts with the `simpleType` element. This defines the name and is similar a complex type definition.
- The `restriction` element identifies the data type that is constrained or restricted. In the example in Figure A.11, the `int` type is restricted.
- The child element inside of the `restriction` element determines the type of restrictions placed on the data.

Figure A.12 uses the two attributes `minInclusive` and `maxInclusive` elements to restrict the range of the `int` value. These elements can be used with any numeric type. In this example, any `member_since` must be between 1990 and 2100.

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

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

❶ Define the name of this simpleType.
❷ The restriction element sets the simpleType that will be constrained.
❸ The constraints to be applied to the simpleType.

Restricting an Integer

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

member_since
1991

```

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`.

Figure A.11: Defining Simple XML Schema Types

```

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

```

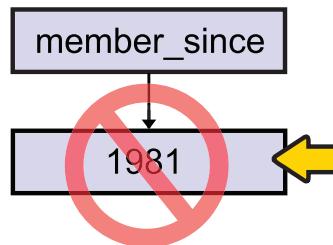


Figure A.12: Restricting an Integer

Using Data Types in an XML Schema

```

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

```

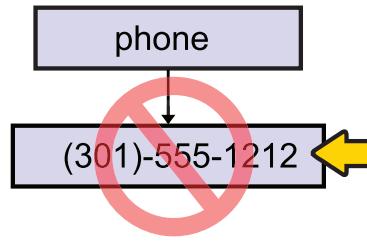


Figure A.13: Using Patterns

The pattern element uses a regular expression to define the structure of a particular string. In the example in Figure A.13, the string must contain three numbers, a dash, three numbers, a dash, and four numbers. This is the basic format for a phone number in the U.S.

Figure A.14 shows an example of a named type that is used with the `species` attribute. Notice that the basic structure of the definition has not changed. This definition restricts the string type to the three values specified by the `enumeration` elements. Although the syntax is different, the result is the same as a DTD enumeration definition.

```

<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>

```

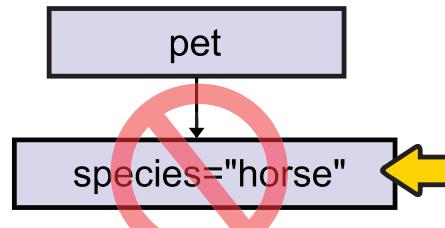


Figure A.14: Using Enumerations

```

<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>

```

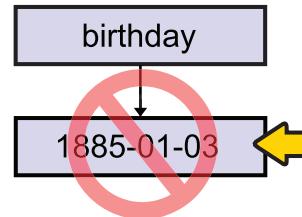


Figure A.15: Using Dates

Figure A.15 shows a type that sets up constraints for the date type. The two elements `minInclusive` and `maxInclusive` are used with the date type to set a range, just as they do with numeric types.

Using Data Types in an XML Schema

```

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

```

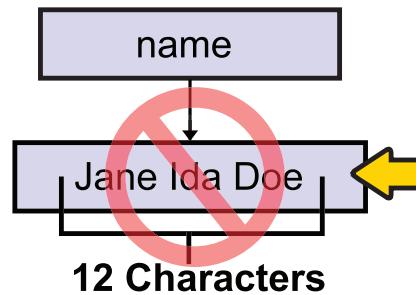
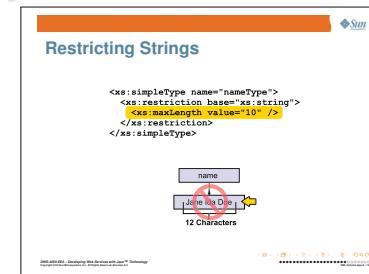


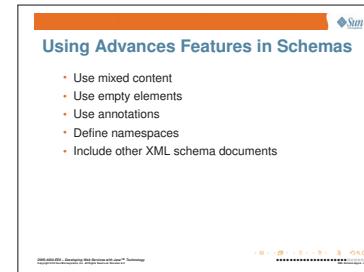
Figure A.16: Restricting Strings

Another restriction you can set is the length of a string. The example in Figure A.16 restricts the length of names to 10 characters. The `maxLength` element specifies the length through the `value` attribute.

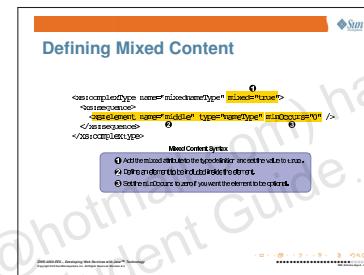


Using Advances Features in Schemas

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



Mixed content elements contain both text and other elements. The example if Figure A.17 shows to define an element as mixed content in a schema. Basically, when you define a complex type, you set the `mixed` attribute to `true`. Some of the key steps for setting up a mixed content element in this example include:



1. Set the `mixed` attribute to `true`, and mixed content is enabled for an element.
2. The `middle` element and text for a personss name is included for this type of element.
3. To make the element optional, do not forget to set `minOccurs` to 0. Otherwise, you are required to include at least one element.

```

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

Figure A.17: Defining Mixed Content

Using Advances Features in Schemas

```

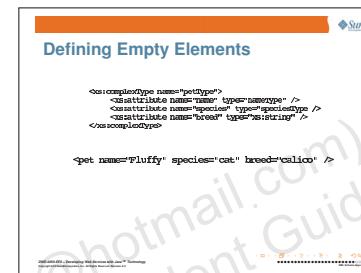
<xs:complexType name="petType">
    <xs:attribute name="name" type="nameType" />
    <xs:attribute name="species" type="speciesType" />
    <xs:attribute name="breed" type="xs:string" />
</xs:complexType>

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

```

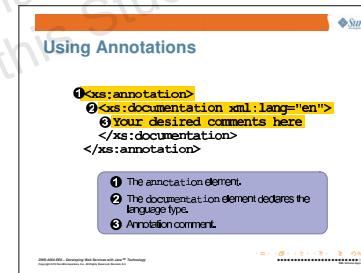
Figure A.18: Defining Empty Elements

To define an empty element, define a complex type that contains only attributes. The definition shown in Figure A.18 changes the pet element. With this change, the pets name is an attribute because there is no content for the element.



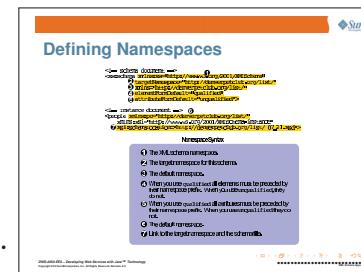
Annotations are a way to add comments to a schema document. The parts of the annotation element are:

- The annotation element is the container for comments.
- The documentation element declares the language type that the included comments are written in.
- The text comments are included between the documentation tags.



To this point, a namespace has not yet been defined for the example schema. Figure A.20 shows how to define a namespace in both the schema and the instance document:

1. This statement defines the schema namespace. This attribute is included in all of the example schema documents.
2. This statement defines the target namespace for your schema. In this example, the <http://denverpetclub.org/list/> URI uniquely identifies this schema. It is not actually checked by the parser.



```
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.

Figure A.19: Using Annotations

3. This attribute sets the default namespace to the target namespace. By setting the default, you do not have to include prefixes in your schema definitions.
4. This attribute determines whether a prefix is required for your instance document elements. If the value is qualified, each element must be preceded by a prefix. If the value is unqualified, the prefix may be omitted.
5. This attribute determines whether a prefix is required for your instance document attributes. If the value is qualified, each attribute must be preceded by a prefix. If the value is unqualified, the prefix may be omitted.
6. This statement defines the prefix for your namespace. The prefix should match your `targetNameSpace` attribute. In this example, the prefix `pc` is specified. Figure A.21 shows how this prefix definition affects the XML document.
7. This statement links the document to the target namespace and the namespace file.

Using Advances Features in Schemas

```
<!-- schema document -->
<xss:schema xmlns:xss="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

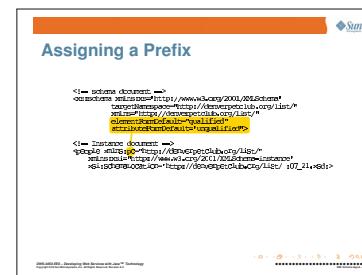
<!-- Instance document --> ⑥
<people xmlns:pc="http://denverpetclub.org/list/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://denverpetclub.org/list/ 07_21.xsd"> ⑦
```

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.

Figure A.20: Defining Namespaces

The two attributes `elementFormDefault` and `attributeFormDefault` determine if you must include a prefix before each element or attribute. In Figure A.21, the `elementFormDefault` value is set to `qualified`. Now all elements from the pet club namespace must be preceded by the `pc` prefix. Because `attributeFormDefault` is set to `unqualified`, attributes do not require a prefix.



```

<!-- schema document -->
<xsd:schema xmlns:xsd='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">

```

Figure A.21: Assigning a Prefix

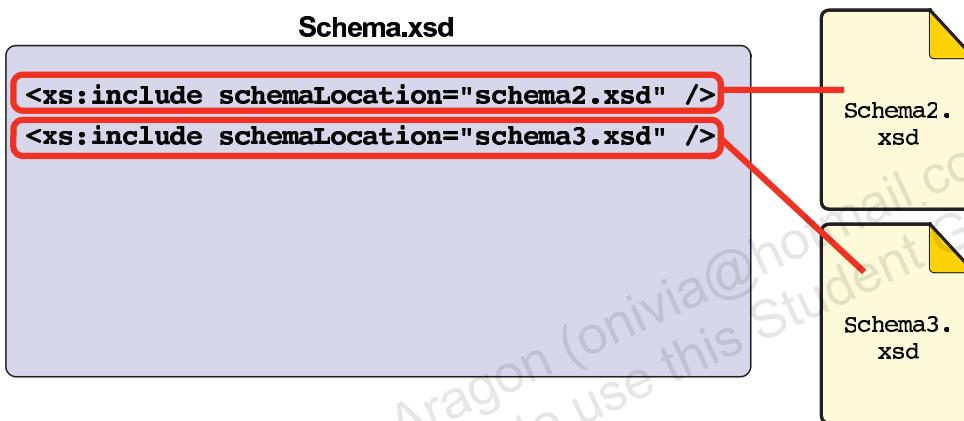
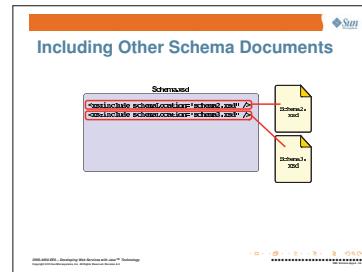


Figure A.22: Including Other Schema Documents

A large design for a schema might require that the schema be broken up into multiple documents. To include other schema documents in your schema file, add the `include` element, as shown in Figure A.22. The `include` element must come before any type definitions in your schema file. Included schema documents must be defined with the same target namespace as the document that includes them.



Using Advances Features in Schemas

Oscar Hernando Nivia Aragon (onivia@hotmail.com) has a
non-transferable license to use this Student Guide.

Appendix B

JAXB: the Java XML Binding API

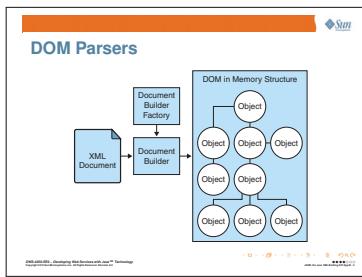
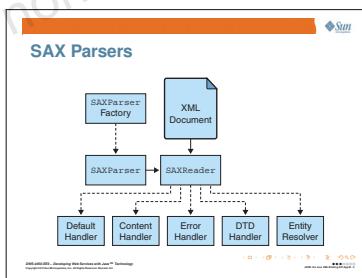
On completion of this module, you should be able to:

- 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.

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.



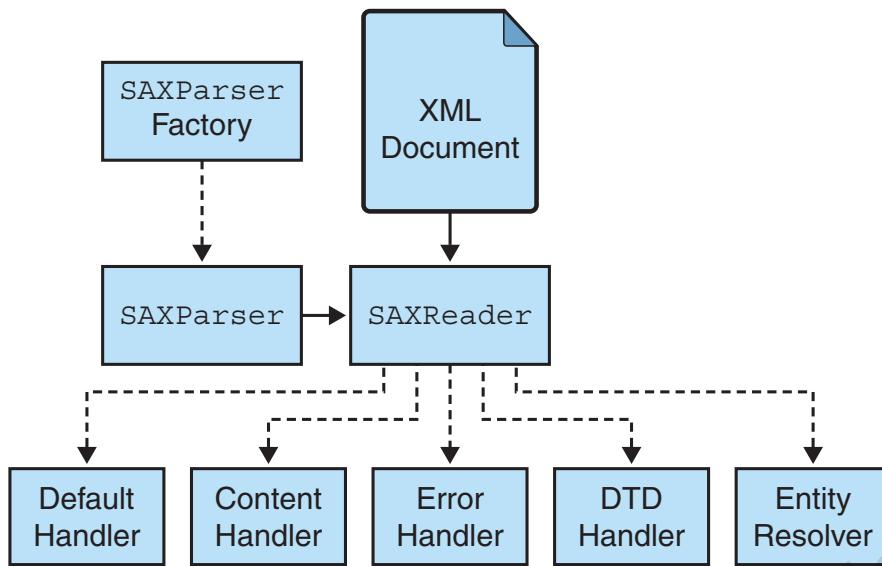


Figure B.1: SAX Parsers

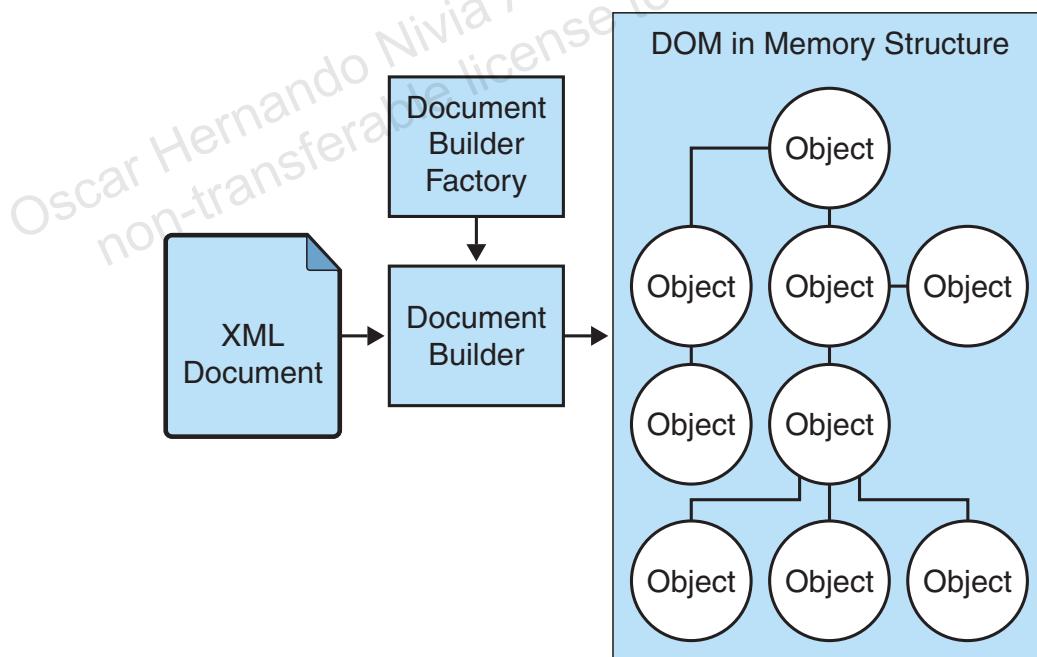


Figure B.2: DOM Parsers

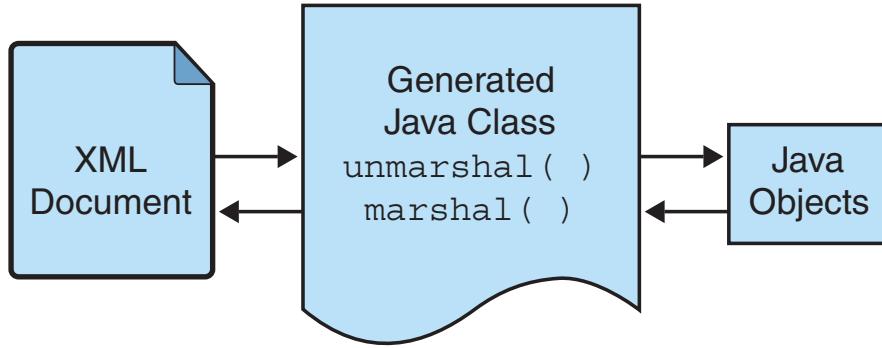


Figure B.3: JAXB Application

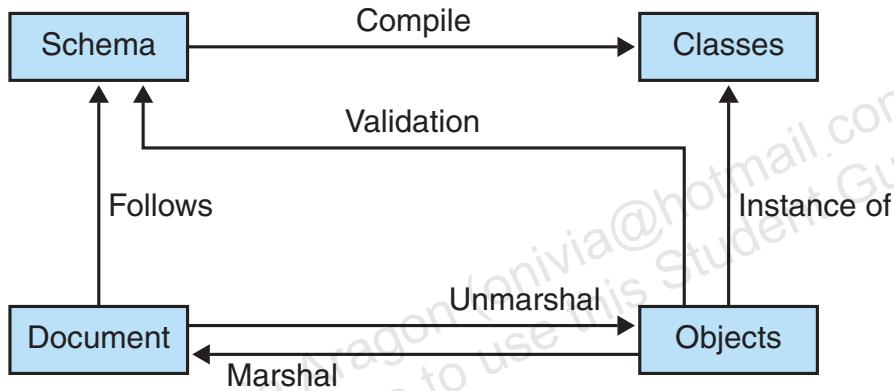
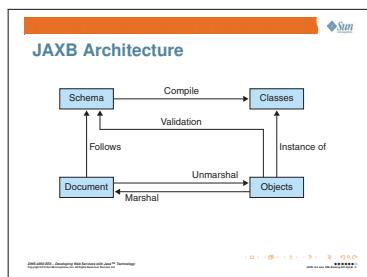
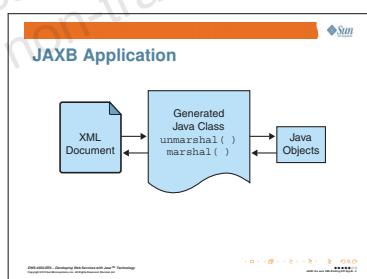


Figure B.4: JAXB Architecture



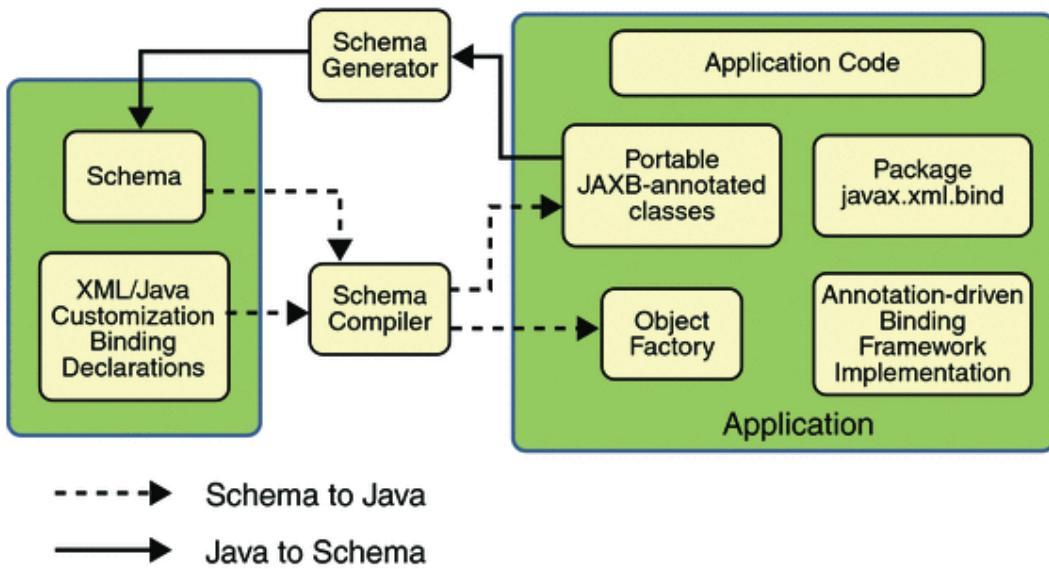
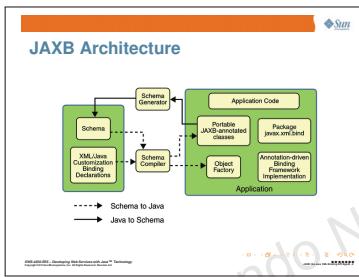


Figure B.5: JAXB Architecture



Schema Validation

Validation is the process of verifying that an XML document meets all the constraints expressed in the schema. JAXB 1.0 provided validation at unmarshal time and also enabled on-demand validation on a JAXB content tree. JAXB 2.0 only allows validation at unmarshal and marshal time. A web service processing model is to be lax in reading in data and strict on writing it out. To meet that model, validation was added to marshal time so one could confirm that they did not invalidate the XML document when modifying the document in JAXB form.

Schema Validation

Oscar Hernando Nivia Aragon (onivia@hotmail.com) has a
non-transferable license to use this Student Guide.

Appendix C

JAXP and SAAJ

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

- Obtain the SOAP envelope for a web service request from the JAX-WS runtime.
- Manipulate SOAP structures directly using the SOAP with Attachments API for Java (the SAAJ API).
 - Extract information from a SOAP envelope
 - Construct a SOAP envelope
- Issue a web service call directly through SAAJ

The screenshot shows a presentation slide with the following content:

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

At the bottom right of the slide are standard presentation navigation icons.

Additional Resources

Additional Resources

The following references provide additional information on the topics described in this module:

- SR-000005 XML Parsing (JAXP) Specification Final Release. [Online]. Available at:
<http://java.sun.com/aboutJava/communityprocess/final/jsr005/>.
- “Working with XML: The JavaTM API for XML Parsing (JAXP) Tutorial”. [Online]. Available at:
http://java.sun.com/xml/tutorial_intro.html.
- Java™ API for XML Parsing Release 1.1 (JAXP documentation and APIs). [Online]. Available at:
<http://java.sun.com/xml/download.html>.
- SAX 2.0/Java final release standard. [Online]. Available at:
<http://www.megginson.com/SAX/Java/index.html>.
- Document Object Model (DOM) standard. [Online]. Available at:
<http://www.w3.org/DOM>.

JAXP

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

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

SAX is an interface for XML parsers based on event handling. When an XML parser analyzes XML documents and encounters a start tag, a corresponding method is called. This corresponding method is defined in a SAX interface, and the method is registered in the event handler of the applications that parse XML documents. SAX provides many methods that the application programs can use to receive the contents of various XML documents. SAX lets you handle XML documents by obtaining data from the documents sequentially. Figure 2-1 demonstrates SAX processing of XML documents.

SAX Overview

XML document

User applications

SAX interface

Implementation

Event handler

Implements SAX interfaces and overrides necessary methods

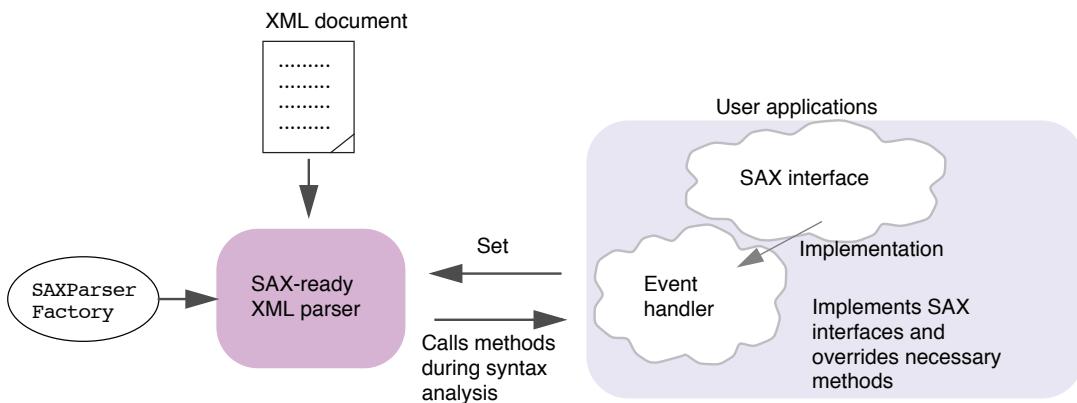
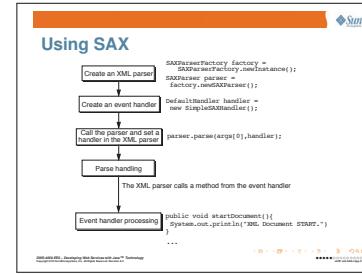


Figure C.1: SAX Overview

XML documents are parsed using the parse method of the `javax.xml.parsers.SAXParser` class to analyze the documents syntax. The input sources can be input streams, files, URIs, and instances of SAX InputSource. The following is a list of the parse methods:



- **void** `parse(File f, DefaultHandler dh)`
 - Parses the content of the file specified as XML using the specified `org.xml.sax.DefaultHandler` class.
- **void** `parse(InputSource is, DefaultHandler dh)`
 - Parses the content of the given `org.xml.sax.InputSource` instance using the specified `org.xml.sax.DefaultHandler` class.
- **void** `parse(InputStream is, DefaultHandler dh)`
 - Parses the content of the given `java.io.InputStream` instance as XML using the specified `org.xml.sax.DefaultHandler` class.
- **void** `parse(String uri, DefaultHandler dh)`
 - Parses the content described by the given URI as XML using the specified `org.xml.sax.DefaultHandler` class.

The features of the parse method are:

- Calls event handler methods Event handler methods are called from XML parsers during the parse method execution.
- Non-reentrant While a parser object of the XML parser is performing parse handling with the parse method, the same parser object should not call the parse method for other XML documents. You should have one instance of the SAXParser per thread because the parse method is not thread safe.
- Reusable After a parser object of the XML parser performs parse handling with the parse method, the same parser object can call the parse method for other XML documents.

The `org.xml.sax` package contains interfaces (`ContentHandler`, `DTDHandler`, `ErrorHandler`, and `EntityResolver`) related to various types of event handlers. The `org.xml.sax.helpers` package contains an adapter class (`DefaultHandler`) that provides default implementations of these four interfaces.

These handler interfaces handle the following types of events:

- `ContentHandler` - The interface for handling general document events. This is the main interface that most SAX applications implement. If the

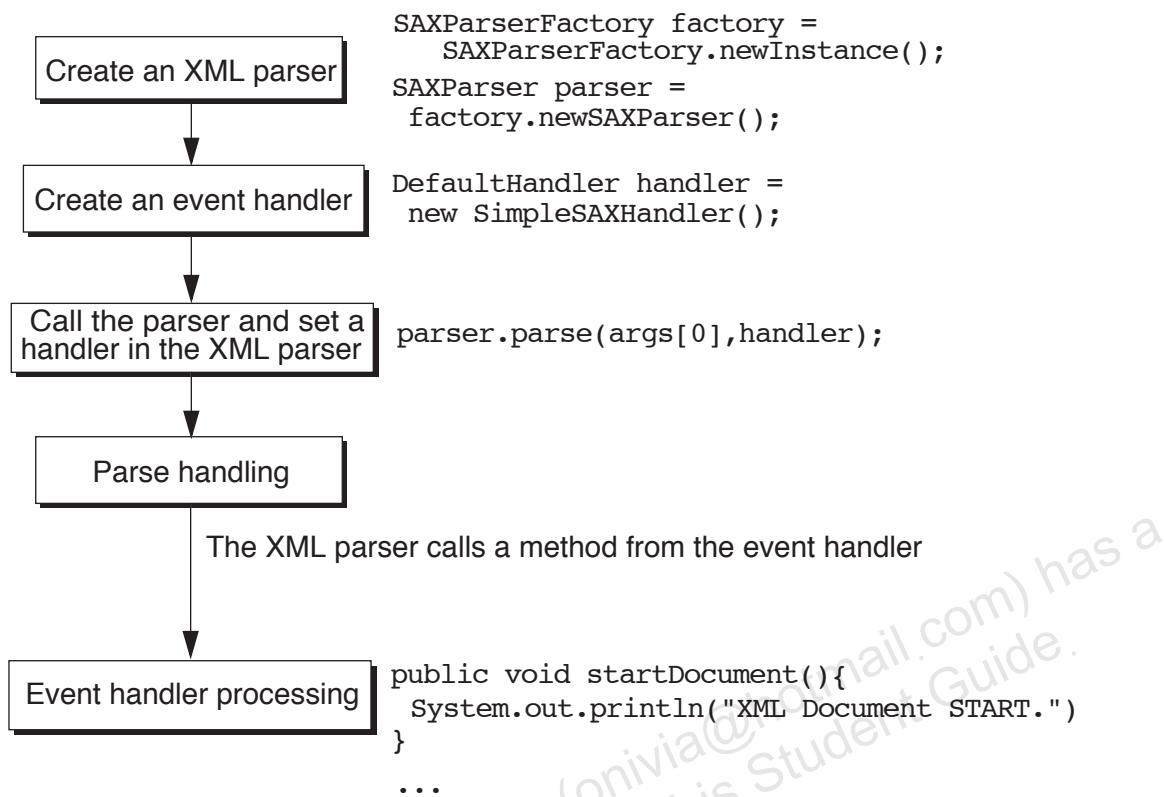


Figure C.2: Using SAX

application must be informed of basic parsing events, it implements this interface and registers an instance with the SAX parser.

- **DTDHandler** - The interface for handling DTD-related events. If a SAX application needs information about notations and unparsed entities, then the application must implement this interface and register an instance of the handler with the SAX parser.
- **ErrorHandler** - The basic interface for SAX error handlers. If a SAX application must implement customized error handling, it must implement this interface and register an instance of the handler with the SAX parser.
- **EntityResolver** - The basic interface for resolving entities. If a SAX application must implement customized handling for external entities, it must implement this interface and register an instance with the SAX parser.

Because the `DefaultHandler` class provided in `org.xml.sax.helpers` implements each of these handler interfaces, most applications only extend the `DefaultHandler` class and override methods as appropriate.

Most of the work associated with parsing an XML document with SAX is delegated then to these handlers. In particular, the `ContentHandler` interface in-

cludes the following methods:

- **void** startDocument()

Receives notification of the start of XML documents. This method is called only once when parsing starts.

- **void** endDocument()

Receives notification of the end of XML documents. This method is called only once when parsing ends.

- **void** startElement(String namespaceURI, String localName,
 String qName, Attributes atts)

Receives notification of the start of an element. The name of the element and attribute specification information are given. This method is paired with the endElement() method.

- **void** endElement(String namespaceURI, String localName,
 String qName)

Receives notification of the end of an element. The name of the element is given.

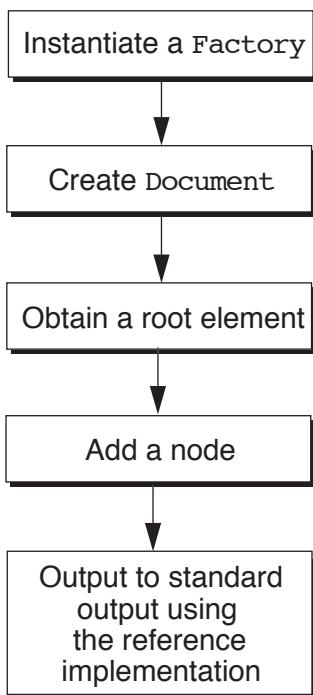
- **void** characters(char ch[], int start, int length)

Receives notification of character data. Character data arrays and their length, as well as the starting position in the XML document, are given.

DOM

DOM is a language-independent interface used for applications. DOM allows applications and scripts to access and update the contents of documents dynamically.

DOM uses a tree structure to represent the tag structure of XML documents, where in each node of the tree contains one of the components from an XML structure. The two most common types of nodes are element nodes and text nodes. Using DOM functions, you can create nodes, remove nodes, change their contents, and traverse the node hierarchy. Java technology applications that use DOM can manipulate data from XML documents as objects in this tree structure.



```

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));
  
```

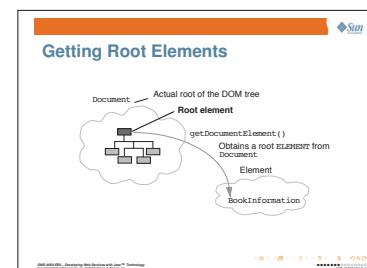
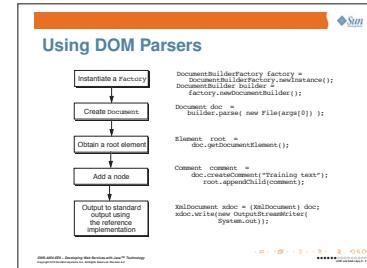
Figure C.3: Using DOM Parsers

The JAXP provides two classes for creating an `org.w3c.dom.Document` object:

- `DocumentBuilderFactory` Defines a factory API that enables applications to obtain a parser that produces DOM object trees from XML documents.
- `DocumentBuilder` Defines the API to obtain DOM Document instances from an XML document. Using this class, you can obtain an instance of `org.w3c.dom.Document` from the XML content.

After you have your `DocumentBuilder` object, you can call its `parse` method (which takes the same variety of input types as the `SAXParser parse` method). This method returns a new DOM `Document` object.

XML document roots are obtained using the `getDocumentElement()` method of the `Document` interface. Once the application has a reference to this root element, it can navigate its way around the DOM tree for the complete document.



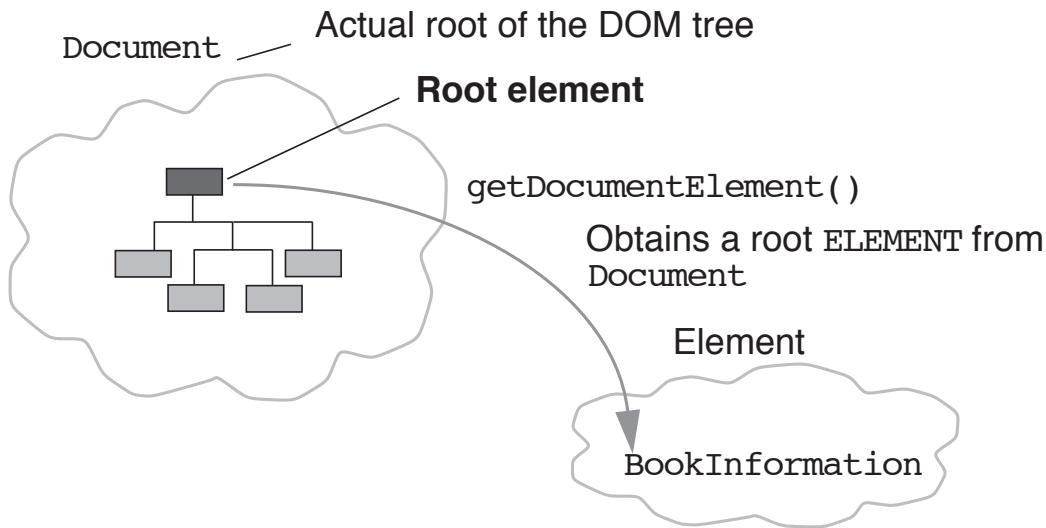


Figure C.4: Getting Root Elements

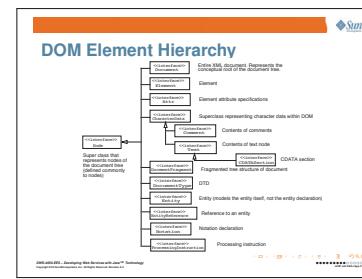
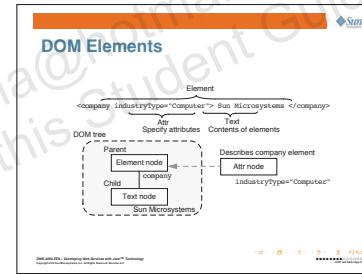
Element nodes, representing elements, are one of the most common nodes within document trees (text nodes and comment nodes are also very common). Root elements are element nodes.

An element tag and its corresponding contents have a parent-child relationship. An element node (with element tag name) is the parent, and a text node (with the element contents) is the child.

However, although the specification of an attribute within an element tag results in an attribute node, these node types are not part of the DOM tree. Therefore, values for `parentNode`, `previousSibling`, and `nextSibling` methods for attribute nodes are `null`.

Figure C.5 illustrates the DOM nodes that correspond to a sample XML element.

Various node types within a document tree are represented by a specific interface, with the `org.w3c.dom.Node` interface as the superclass, as illustrated in Figure C.6. Nodes are used as various types of information within XML documents, and this information becomes objects implemented in the interface.



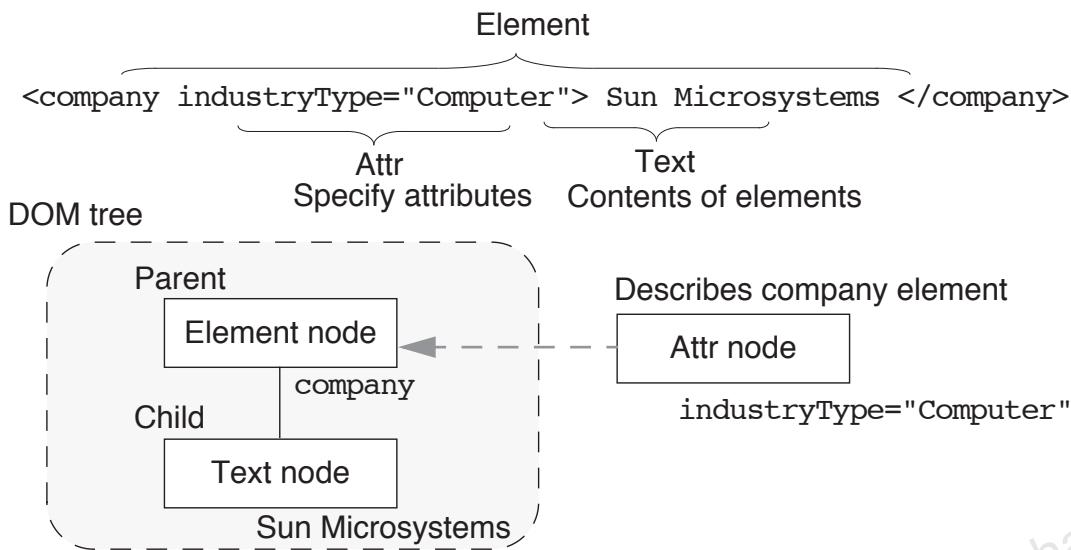


Figure C.5: DOM Elements

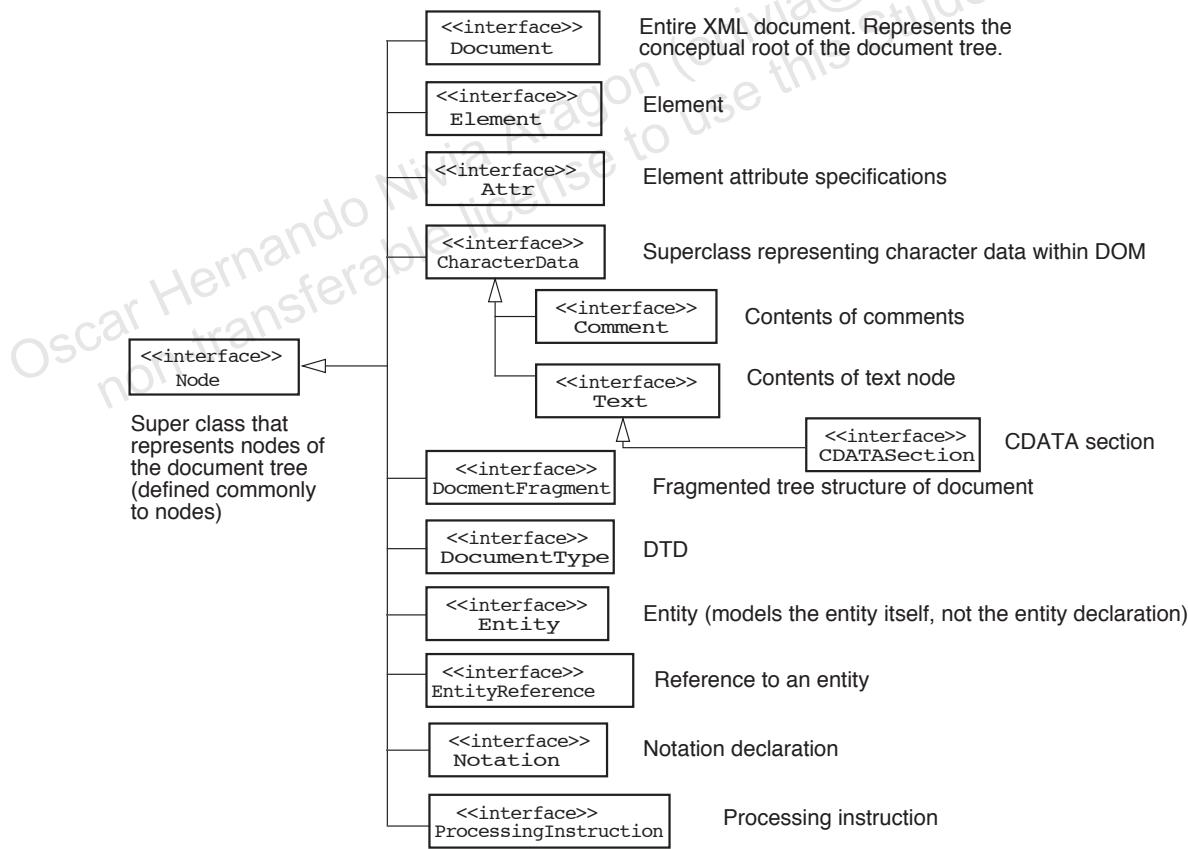


Figure C.6: DOM Element Hierarchy

SAAJ

You can use the SAAJ API to create, read, modify, send, and receive SOAP messages. SAAJ contains all of the functionality necessary to process SOAP messages used in a web service exchange. The SAAJ API supports two message types: XML documents and SOAP messages with MIME attachments.

The SAAJ API allows you to create XML messages that conform to the SOAP 1.1 or 1.2 specification and to the WS-I Basic Profile 1.1 specification by making Java API calls. SAAJ has classes and interfaces mapping to the SOAP elements, such as Envelope, Header, Body, and Fault. You can represent XML Namespaces, elements, attributes, and text nodes of a SOAP message using SAAJ. The SAAJ API types, and their relationship to SOAP structure, is illustrated in Figure C.7.

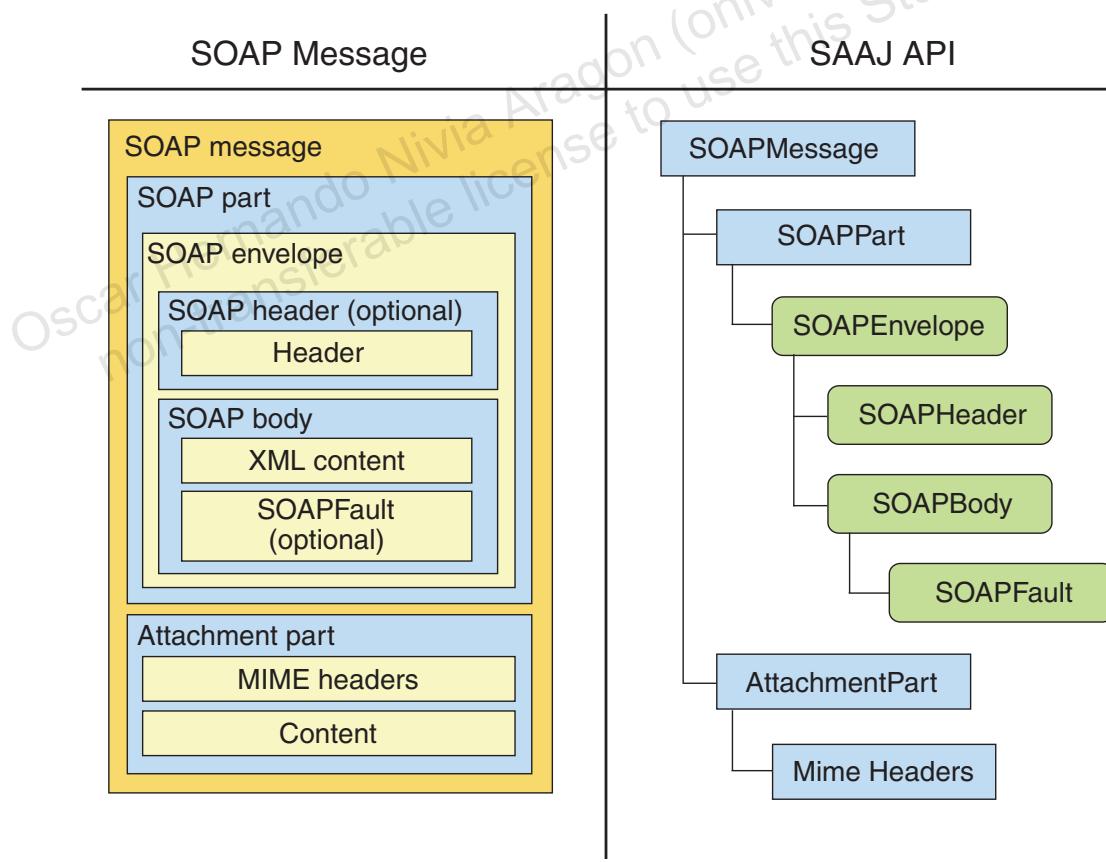
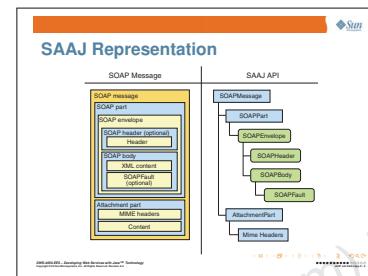


Figure C.7: SAAJ Representation

Java components can use SAAJ to abstract the details of processing SOAP messages:

- Java code works to the API, and therefore works with references to the SOAP envelope, the header, the body, and to individual XML elements within the header and body.
- SAAJ code handles the details of message construction and parsing.

Figure C.8 illustrates the high-level view of the SAAJ API. All classes and interfaces are in the package `javax.xml.soap`. SAAJ follows the Abstract Factory Pattern. The abstract factory class is `MessageFactory`, which creates an instance of itself. You can use this factory instance to create a `SOAPMessage`. The `SOAPMessage` contains `SOAPPart`, which represents the SOAP document and `AttachmentPart`, which represents MIME attachments. `SOAPPart` contains a group of objects to represent the structure of a SOAP document, namely `SOAPEnvelope`, which contains `SOAPHeader` and `SOAPBody`. `SOAPBody` contains `SOAPFault`, which represents the error or exception content.

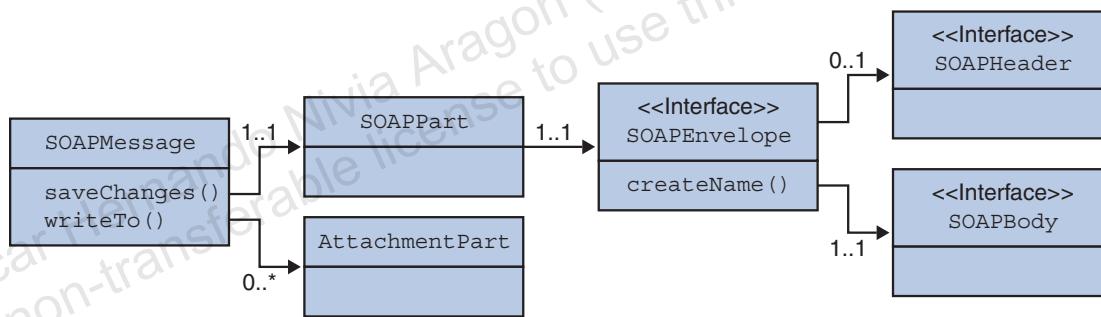
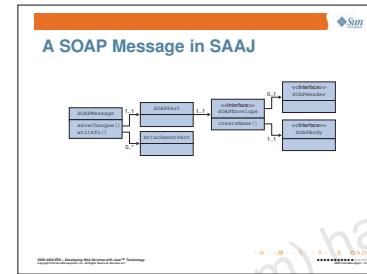


Figure C.8: A SOAP Message in SAAJ

The `SOAPHeader` interface provides a representation of the SOAP header element. A SOAP header element consists of XML data that affects the way the application-specific content is processed by the message provider. For example, you can specify transaction semantics, authentication information, and so on, as the content of a `SOAPHeader` object.

The `SOAPBody` interface is an object that represents the contents of the SOAP body element in a SOAP message. A SOAP body element consists of XML data that affects the way the application-specific content is processed.

A `SOAPBody` object contains `SOAPBodyElement` objects, which represent the content for the SOAP body. A `SOAPFault` object, which carries status or error information, is an example of a `SOAPBodyElement` object.

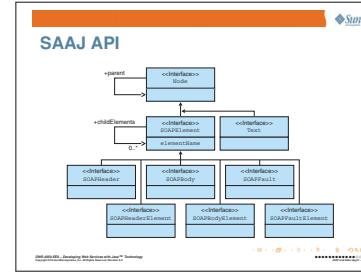
The SOAPBody interface is almost entirely a composition of body elements. The SOAP body holds a group of message-specific XML elements and attributes. From this point in a SOAP message, the message composition is driven by the needs of the application.

The Node interface is the head of an inheritance tree that includes many other types, each of which maps to a certain kind of XML element. The Node interface is not, however, the real centerpiece of the system:

- The Node interface only captures some of the most common semantics, including navigation up the hierarchy to a parent node.
- The SOAPElement interface provides the remaining functionality. The SOAPElement interface, rather than the Node interface, allows you to navigate down through child nodes.
- The Node interface also offers the `getValue()` method, which allows you to retrieve the text content of an element. As a convenience, this method is defined to return the value of a Text node directly, but also, when called on a SOAPElement, to return the value of any Text node that is the single, direct child of the element.

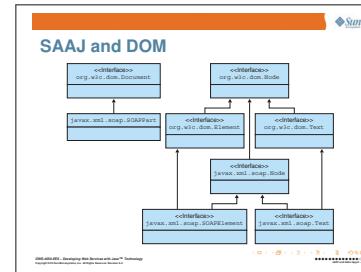


The Text interface is a slight specialization of the Node interface and so it cannot have children. This is similar to the DOM Text node type because it does not map to an XML element or attribute, but to the text-node content of an element in the XML infoset.



Because SAAJ nodes and elements implement the DOM Node and Element interfaces, you have many options for adding or changing message content:

- Use only DOM APIs.
- Use only SAAJ APIs.
- Use SAAJ APIs and then switch to using DOM APIs.
- Use DOM APIs and then switch to using SAAJ APIs.



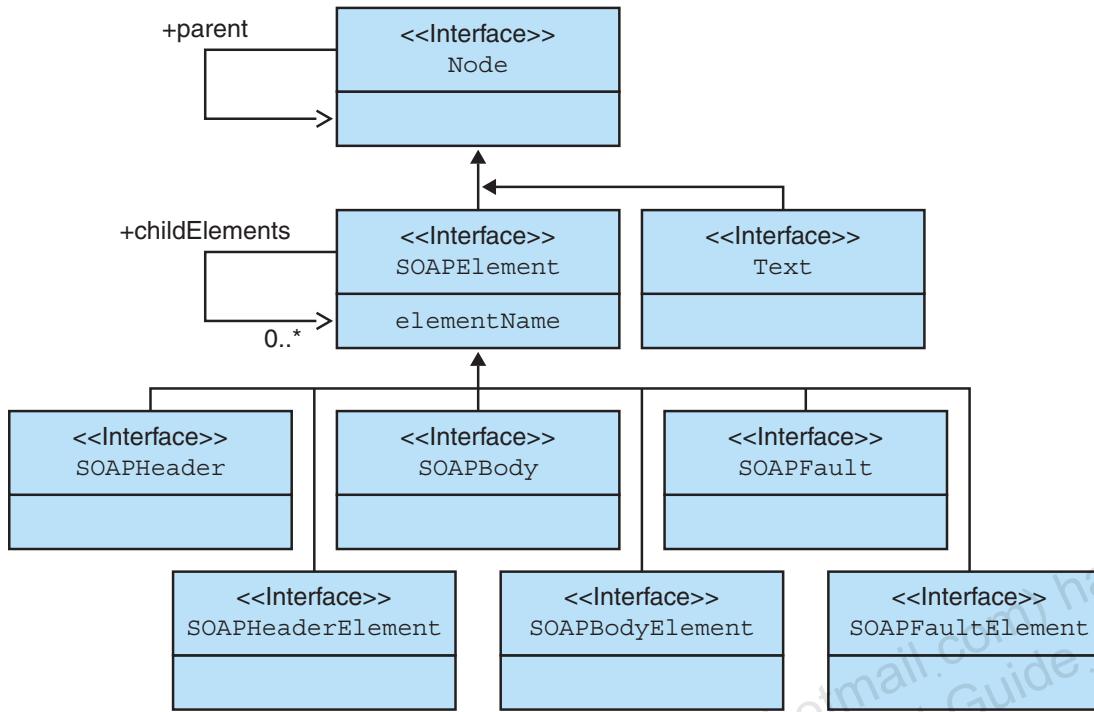


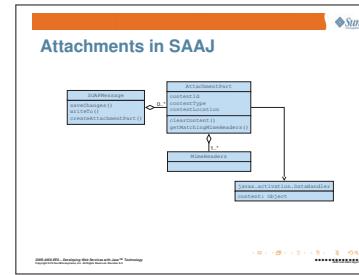
Figure C.9: SAAJ API

The first three cause no problems. After you have created a message, regardless of whether you have imported its content from another document, you can start adding or changing nodes using either SAAJ or DOM APIs.

But if you use DOM APIs and then switch to using SAAJ APIs to manipulate the document, any references to objects within the tree that were obtained using DOM APIs are no longer valid. If you must use SAAJ APIs after using DOM APIs, you should set all your DOM typed references to null, because they can become invalid.

The SAAJ object model provides full support for SOAP attachments as part of its object model. As with SOAP in general, there is a tradeoff of control over content for simplicity of coding. As Figure C.11 shows, the `SOAPMessage` class is the entry point for working with SOAP attachments. You can include multiple attachments directly under the root message object. The interface for managing attachment content is more complex than that for the SOAP part, which is mandatory and singular, and has its own encapsulation.

An `AttachmentPart` object can contain any type of content, including XML. Because the SOAP part can contain only XML content, an `AttachmentPart` object must be used for any content that is not in XML format. This `AttachmentPart`



SAAJ

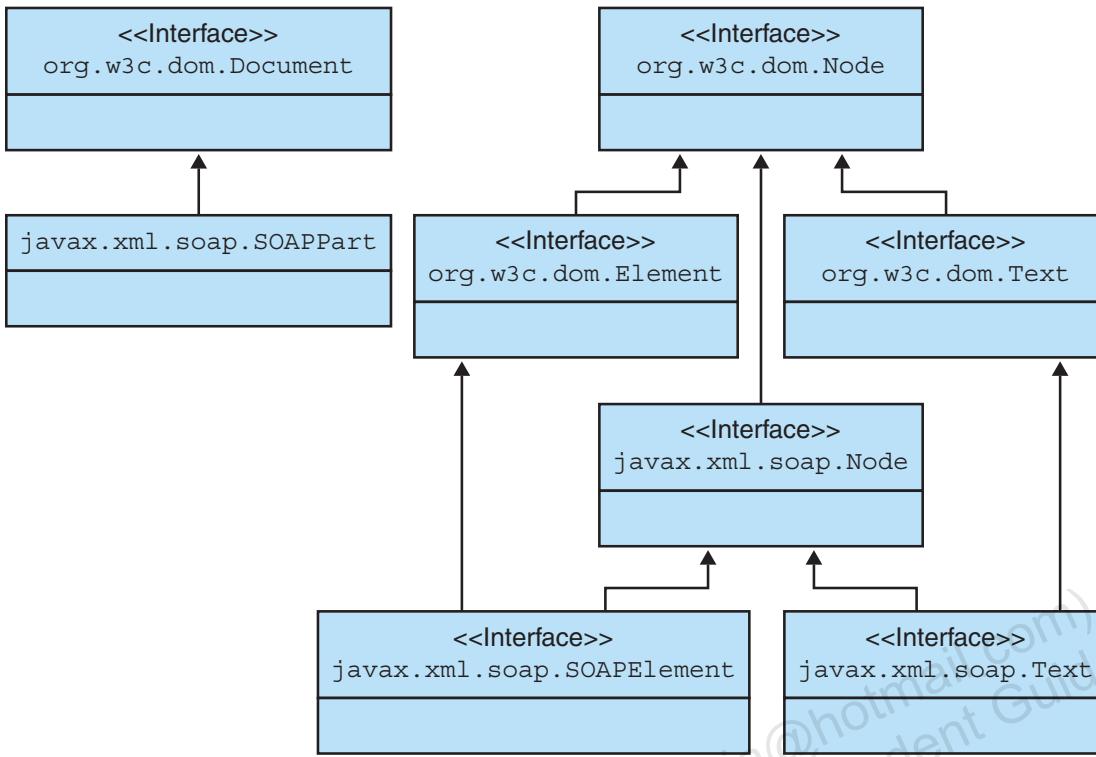


Figure C.10: SAAJ and DOM

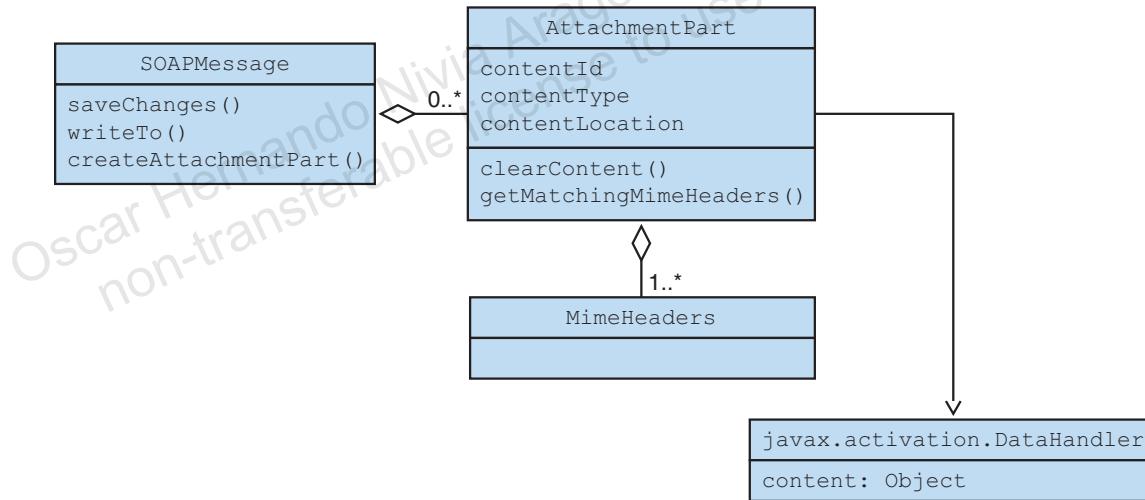
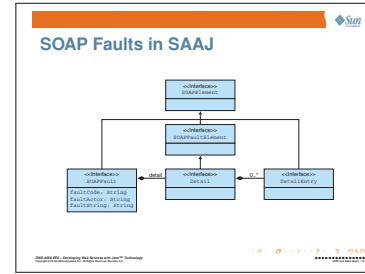


Figure C.11: Attachments in SAAJ

class encapsulates a single attachment. It duplicates some of the `SOAPMessage` functionality in dealing with `DataHandlers`, `MimeHeaders`, and so forth. The class also provides formal accessor and mutator methods for some important standard MIME headers, such as `Content-Type`, `Content-ID`, and `Location`.

If you send a message that was not successful, you might receive a response containing a SOAP fault element, which gives you status information, error information, or both. There can be only one SOAP fault element in a message, and it must be an entry in the SOAP body. Furthermore, if there is a SOAP fault element in the SOAP body, there can be no other elements in the SOAP body. This means that when you add a SOAP fault element, you have effectively completed the construction of the SOAP body.

SOAP uses SOAP faults as a mechanism for error reporting. SAAJ maps SOAP faults to the `SOAPFault` interface.



- `SOAPFault` extends `SOAPElement`, so the typical error-handling code creates a message with a `SOAPFault` as the only child of the `SOAPBody`.
- `SOAPBody` has a special factory method for this purpose.

Figure C.12 shows how a `SOAPFault` object breaks down the four standard parts of a fault message into its own properties.

A `SOAPFault` object, which is the representation of a SOAP fault element in the SAAJ API, is similar to an `Exception` object in that it conveys information about a problem. However, a `SOAPFault` object is different in that it is an element in a messages `SOAPBody` object rather than part of the try/catch mechanism used for `Exception` objects. Also, as part of the `SOAPBody` object, which provides a simple means for sending mandatory information intended for the ultimate recipient, a `SOAPFault` object only reports status or error information. It does not halt the execution of an application, as an `Exception` object can.

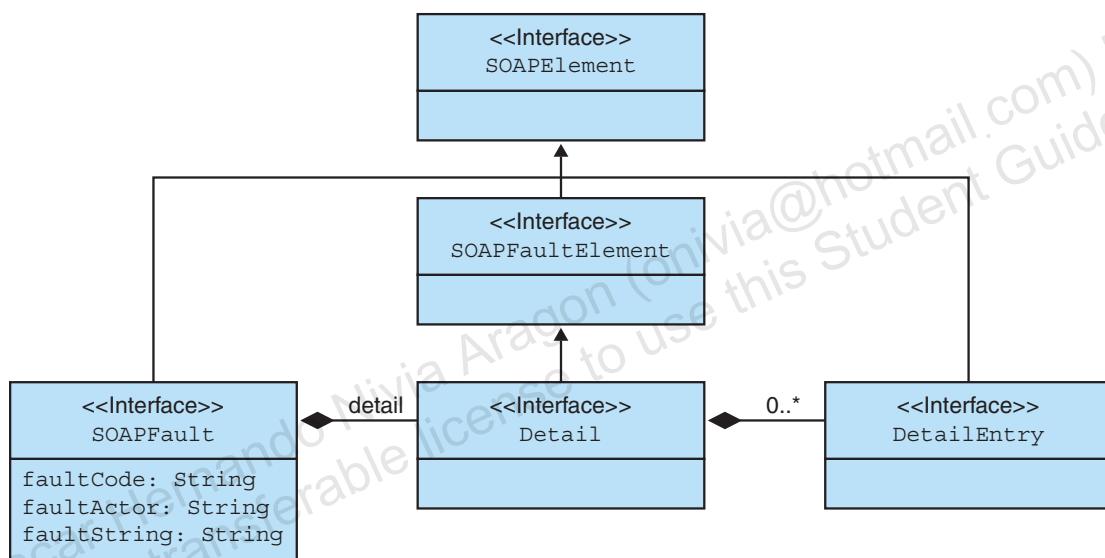


Figure C.12: SOAP Faults in SAAJ

Appendix D

JAX-WS Handlers

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

The screenshot shows a slide from a presentation. At the top right is the Sun logo. The title 'Objectives' is in bold. Below it, a bullet point states: 'On completion of this module, you should be able to:'. There are three main points listed:

- 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

Incorporating Request Metadata

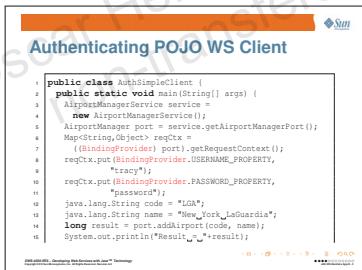
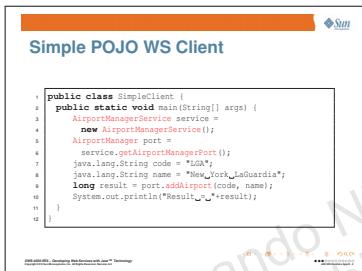
```

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 }
```

E-74

Code D.1: Simple POJO WS Client

Incorporating Request Metadata



```
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);
```

E-139



Code D.2: Authenticating POJO WS Client

Using JAX-WS Handlers

Handlers are message interceptors that can be easily plugged in to the JAX-WS runtime to do additional processing of the inbound and outbound messages. Figure D.1 illustrates the overall runtime architecture of JAX-WS, indicating where these handlers plug into the JAX-WS runtime.

JAX-WS defines two types of handlers, *logical* handlers and *protocol* handlers. Protocol handlers are specific to a protocol and may access or change the protocol specific aspects of a message. Logical handlers are protocol-agnostic and cannot change any protocol-specific parts (like headers) of a message. Logical handlers act only on the payload of the message. Figure D.2 illustrates the targets for the two types of handlers

Logical handlers can coexist with SOAP handlers in a handler chain. During runtime, the handler chain is re-ordered such that logical handlers are executed before the SOAP handlers on an outbound message and SOAP handlers are executed before logical handlers on an inbound message. Figure D.3 shows how logical and SOAP

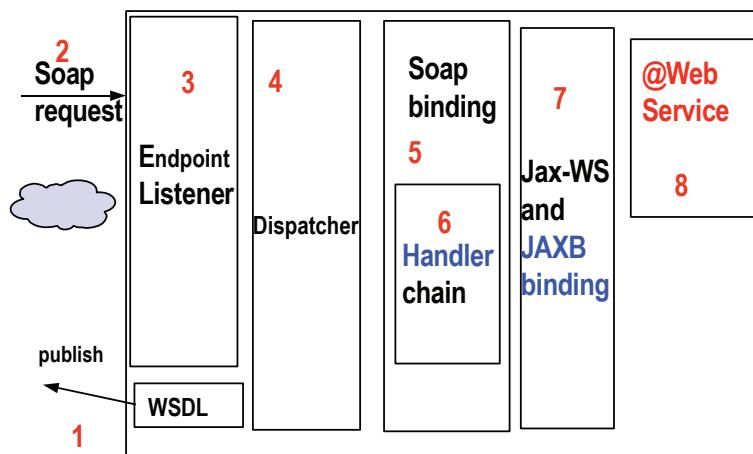
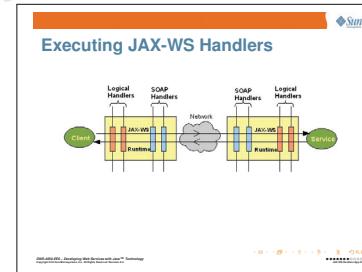
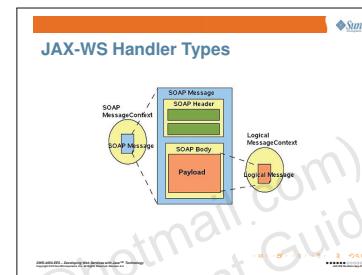
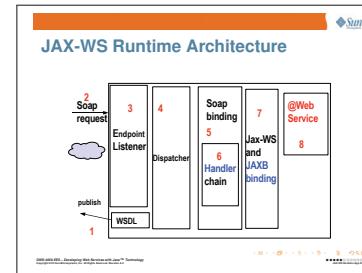


Figure D.1: JAX-WS Runtime Architecture

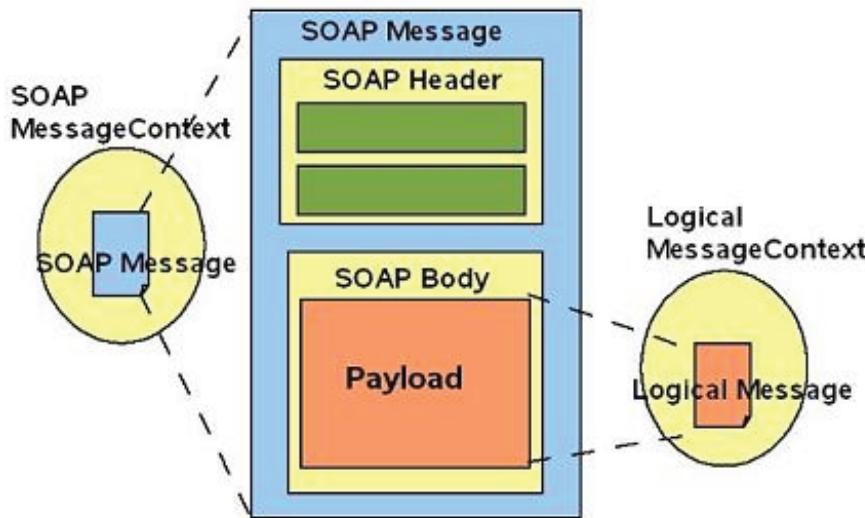


Figure D.2: JAX-WS Handler Types

handlers are invoked during a request and response.

Writing a handler in JAX-WS is easy. A basic handler should implement the three methods in `javax.xml.ws.handler.Handler`, illustrated in Figure D.3:

- `handleMessage()` –
This is called for inbound and outbound messages.
- `handleFault()` –
This is called instead of `handleMessage()` when the message contains a

Writing a JAX-WS Handler

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

```

public interface
Handler<? extends MessageContext> {
    boolean handleMessage(C ctx);
    boolean handleFault(C ctx);
    void close(MessageContext ctx);
}

```

Java code Java - Developing Web Services with Java™ Technology
Copyright 2010 Sun Microsystems, Inc. All rights reserved. DWS-4050-EE6 Rev. A.0

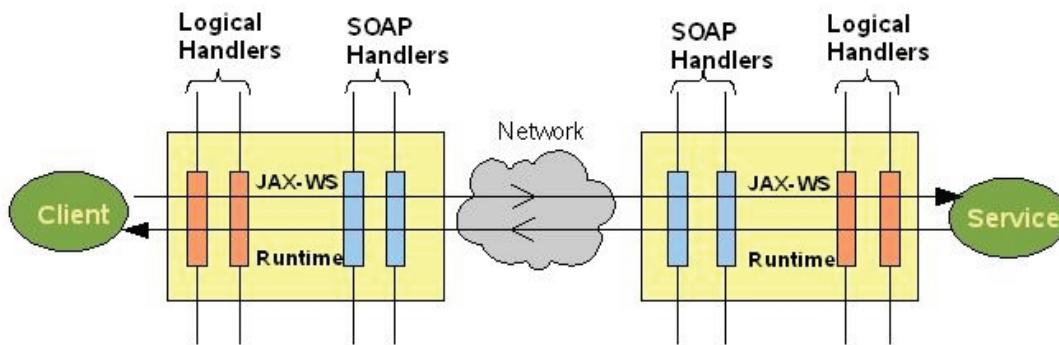


Figure D.3: Executing JAX-WS Handlers

Using JAX-WS Handlers

```

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

Code D.3: Interface javax.xml.ws.handler.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 }
```

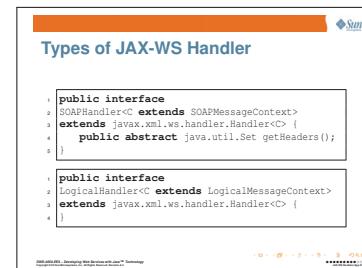
Code D.4: Types of JAX-WS Handler

protocol fault.

- `close()` –

This is called after the completion of message processing by all handlers for each web service invocation (after completion of MEP). This can be useful to clean up any resources used during processing the message.

SOAP handlers should extend `javax.xml.ws.handler.soap.SOAPHandler`, which is defined for SOAP binding by JAX-WS specification. SOAP handlers are invoked with `SOAPMessageContext`, which provides methods to access `SOAPMessage`. One can use the SAAJ API to manipulate the SOAP Message.



Logical handlers extend `javax.xml.ws.handler.LogicalHandler` and provide access to message context and message payload. If you are using SOAP over HTTP, the content of the SOAP body forms the payload. If you are using XML over HTTP, the XML content of the primary part of the message becomes the payload. Logical handlers are invoked with a `LogicalMessageContext`. `LogicalMessageContext.getMessage()` returns a `LogicalMessage`.

The `LogicalMessage` represents a protocol neutral XML message and contains methods that provide access to the payload of the message.

Handlers are invoked with a message context that provides methods to access and modify inbound and outbound messages and to manage a set of properties. These message context properties can be used to communicate information between handlers and client and service implementations.

```

1  public class AuthenticationHandler {
2      public boolean handleInbound(SOAPMessageContext smc) {
3          SOAPMessage msg = smc.getMessage();
4          String user = smc.getSOAPContext().getProperties("user");
5          String password = smc.getSOAPContext().getProperties("password");
6          if (user != null & password != null) {
7              try {
8                  embedAuthenticationData(msg);
9                  smc.setException(ex);
10             } catch (Exception ex) {
11                 smc.setException(ex);
12             }
13         }
14     }
15     return true;
16 }

```

```

19     private void embedAuthenticationData(SOAPMessage msg)
20         throws SOAPException {
21     SOAPEnvelope envelope = msg.getSOAPPart().getEnvelope();
22     Name username = envelope.createName("user");
23     Name passwordName = envelope.createName("password");
24     SOAPHeaderElement newHeader = envelope.addHeader();
25     newHeader.addAttribute(username, "iracy");
26     newHeader.addAttribute(passwordName, "password");
27 }

```

```

31     private void validateAuthenticationData(SOAPMessage msg)
32         throws SOAPException {
33     SOAPEnvelope envelope = msg.getSOAPPart().getEnvelope();
34     Name username = envelope.createName("user");
35     Name passwordName = envelope.createName("password");
36     SOAPHeader header = msg.getSOAPHeader();
37     for (Iterator<SOAPHeaderElement> it = header.getAllElements(); it.hasNext();)
38         autBlocker.hasNext();
39     String userHeaderName = header.getAttribute("user");
40     String passwordHeaderName = header.getAttribute("password");
41     validateUser(userHeaderName);
42     validatePassword(passwordHeaderName);
43 }

```

A more detailed discussion of how to set up JAX-WS handlers can be found here:

http://blogs.sun.com/sdimilla/entry/implementing_handlers_using_jaxws_2.



Using JAX-WS Handlers

```

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     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     }
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 =
43                 (SOAPHeaderElement) authNodes.next();
44             String user = authHeader.getAttributeValue(userName);
45             String password =
46                 authHeader.getAttributeValue(passwordName);
47             validate(user, password);

```

E-140



Code D.5: An Authentication Handler

Appendix E

Code Listings

Project: examples/jaxws/Managers

Path: src/com/example/standalone/AirportManager.java

```
1
2 package com.example.standalone;
3
4 import com.example.traveller.dao.pojo.AirportDAO;
5
6 /**
7 *
8 * @author education@oracle.com
9 */
10 public class AirportManager {
11     public long
12         addAirport(String code, String name) {
13             return dao.add(null, code, name).getId();
14         }
15     private AirportDAO dao = new AirportDAO();
16 }
```

Project: examples/jaxws/Managers
Path: src/com/example/jaxws/server/AirportManager.java

```
1
2 package com.example.jaxws.server;
3
4 import com.example.traveller.dao.pojo.AirportDAO;
5 import javax.jws.WebService;
6 import javax.xml.ws.Endpoint;
7
8 /**
9  * 
10 * @author education@oracle.com
11 */
12 @WebService
13 public class AirportManager {
14     public long
15         addAirport(String code, String name) {
16             return dao.add(null, code, name).getId();
17         }
18     private AirportDAO dao = new AirportDAO();
19     // ...
20     static public void main(String[] args) {
21         System.setProperty( "com.sun.xml.ws.transport.http.HttpAdapter.dump",
22                             "true" );
23         String url =
24             "http://localhost:8080/airportManager";
25         if (args.length > 0)
26             url = args[1];
27         AirportManager manager = new AirportManager();
28         Endpoint endpoint =
29             Endpoint.publish(url, manager);
30     }
31 }
```

Project: examples/jaxws/Managers**Path: src/com/example/jaxws/server/NamedAirportManager.java**

```

1
2 package com.example.jaxws.server;
3
4 import com.example.traveller.dao.pojo.AirportDAO;
5 import com.example.traveller.dao.pojo.GenericDAO;
6 import javax.jws.WebMethod;
7 import javax.jws.WebParam;
8 import javax.jws.WebService;
9 import javax.persistence.EntityManager;
10 import javax.persistence.EntityTransaction;
11 import javax.xml.ws.Endpoint;
12 import javax.xml.ws.RequestWrapper;
13 import javax.xml.ws.ResponseWrapper;
14
15 /**
16 *
17 * @author education@oracle.com
18 */
19 @WebService(portName="AirportMgr",
20             serviceName="Managers")
21 public class NamedAirportManager {
22     @WebMethod(operationName="add")
23     @RequestWrapper(className="generated.NAAddAirportRequest")
24     @ResponseWrapper(className="generated.NAAddAirportResponse")
25     public long
26         addAirport(@WebParam(name="code") String code,
27                    @WebParam(name="name") String name) {
28         return dao.add(null, code, name).getId();
29     }
30     private AirportDAO dao = new AirportDAO();
31     // ...
32     static public void
33     main(String[] args) {
34         String url =
35             "http://localhost:8080/namedManager";
36         if (args.length > 0)
37             url = args[1];
38         NamedAirportManager manager = new NamedAirportManager();
39         Endpoint endpoint =
40             Endpoint.publish(url, manager);
41     }
42 }
```

Project: examples/jaxws/Managers
Path: src/com/example/jaxws/server/BetterAirportManager.java

```

1
2 package com.example.jaxws.server;
3
4 import com.example.traveller.dao.pojo.AirportDAO;
5 import com.example.traveller.dao.pojo.GenericDAO;
6 import javax.jws.WebMethod;
7 import javax.jws.WebService;
8 import javax.persistence.EntityManager;
9 import javax.persistence.EntityTransaction;
10 import javax.xml.ws.Endpoint;
11 import javax.xml.ws.RequestWrapper;
12 import javax.xml.ws.ResponseWrapper;
13
14 /**
15 * 
16 * @author education@oracle.com
17 */
18 @WebService
19 public class BetterAirportManager {
20     @WebMethod
21     @RequestWrapper(className="generated.BAMAddAirportRequest")
22     @ResponseWrapper(className="generated.BAMAddAirportResponse")
23     public long
24         addAirport(String code, String name) {
25         return dao.add(null, code, name).getId();
26     }
27     @WebMethod(operationName="removeById")
28     @RequestWrapper(className="generated.BAMRemoveByIdRequest")
29     @ResponseWrapper(className="generated.BAMRemoveByIdResponse")
30     public void removeAirport(long id) {
31         dao.remove(null, id);
32     }
33     @WebMethod(operationName="removeByCode")
34     @RequestWrapper(className="generated.BAMRemoveByCodeRequest")
35     @ResponseWrapper(className="generated.BAMRemoveByCodeResponse")
36     public void removeAirport(String code) {
37         EntityManager em =
38             GenericDAO.getEMF().createEntityManager();
39         EntityTransaction tx = em.getTransaction();
40         tx.begin();
41         dao.remove(em, dao.findByCode(em, code));
42         tx.commit();
43         em.close();
44     }

```

```
45 private AirportDAO dao = new AirportDAO();
46 // ...
47 static public void
48 main(String[] args) {
49     String url =
50         "http://localhost:8080/betterManager";
51     if (args.length > 0)
52         url = args[1];
53     BetterAirportManager manager = new BetterAirportManager();
54     Endpoint endpoint =
55         Endpoint.publish(url, manager);
56 }
57 }
```

Project: examples/jaxws/Managers**Path: src/com/example/jaxws/server/NamespacedAirportManager.java**

```
1
2 package com.example.jaxws.server;
3
4 import com.example.traveller.dao.pojo.AirportDAO;
5 import javax.jws.WebMethod;
6 import javax.jws.WebService;
7 import javax.xml.ws.Endpoint;
8 import javax.xml.ws.RequestWrapper;
9 import javax.xml.ws.ResponseWrapper;
10
11 /**
12 * 
13 * @author education@oracle.com
14 */
15 @WebService(
16     targetNamespace="urn://com.example.managerNS")
17 public class NamespacedAirportManager {
18     @WebMethod
19     @RequestWrapper(className="generated.NAMAddAirportRequest")
20     @ResponseWrapper(className="generated.NAMAddAirportResponse")
21     public long
22         addAirport(String code, String name) {
23         return dao.add(null, code, name).getId();
24     }
25     private AirportDAO dao = new AirportDAO();
26     // ...
27     static public void
28     main(String[] args) {
29         String url =
30             "http://localhost:8080/namespacedManager";
31         if (args.length > 0)
32             url = args[1];
33         NamespacedAirportManager manager = new NamespacedAirportManager();
34         Endpoint endpoint =
35             Endpoint.publish(url, manager);
36     }
37 }
```

Project: examples/jaxws/Managers
Path: src/xml/PassengerManagerPort.wsdl

```

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
10 <types>
11     <xsd:schema>
12         <xsd:import namespace="urn://Traveller/">
13             schemaLocation="PassengerManagerSchema.xsd"/>
14     </xsd:schema>
15 </types>
16
17 <message name="addPassengerRequest">
18     <part name="params" element="tns:addPassenger"/>
19 </message>
20 <message name="addPassengerResp">
21     <part name="params" element="tns:addPassengerResponse"/>
22 </message>
23 <portType name="PassengerManager">
24     <operation name="addPassenger">
25         <input name="in1" message="tns:addPassengerRequest"/>
26         <output name="out1" message="tns:addPassengerResp"/>
27     </operation>
28 </portType>
29 </definitions>
```

Project: examples/jaxws/Managers
Path: src/xml/PassengerManagerSchema.xsd

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsd:schema id="PassengerManagerSchema.xsd"
3     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4     xmlns:tns="urn://Traveller/"
5     elementFormDefault="qualified"
6     targetNamespace="urn://Traveller/">
7
8     <xsd:element name="addPassenger">
9         <xsd:complexType>
10            <xsd:sequence>
11                <xsd:element name="firstName" type="xsd:string"/>
12                <xsd:element name="lastName" type="xsd:string"/>
13            </xsd:sequence>
14        </xsd:complexType>
15    </xsd:element>
16
17    <xsd:element name="addPassengerResponse">
18        <xsd:complexType>
19            <xsd:sequence>
20                <xsd:element name="result" type="xsd:long"/>
21            </xsd:sequence>
22        </xsd:complexType>
23    </xsd:element>
24 </xsd:schema>
```

Project: examples/jaxws/Managers
Path: src/xml/PassengerManagerService.wsdl

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <definitions name="UserDirectoryService.wsdl"
3     targetNamespace="urn://Traveller/"
4     xmlns="http://schemas.xmlsoap.org/wsdl/"
5     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
6     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
7     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
8     xmlns:tns="urn://Traveller/">
9
10    <wsdl:import namespace="urn://Traveller/"
11        location="PassengerManagerPort.wsdl"/>
12
13    <binding name="binding" type="tns:PassengerManager">
14        <soap:binding style="document"
15            transport="http://schemas.xmlsoap.org/soap/http"/>
16        <operation name="addPassenger">
17            <soap:operation/>
18            <input><soap:body use="literal"/></input>
19            <output><soap:body use="literal"/></output>
20        </operation>
21    </binding>
22    <service name="PassengerManagerService">
23        <port name="PassengerManager" binding="tns:binding">
24            <soap:address
25                location="http://localhost:8080/passengerManager"/>
26        </port>
27    </service>
28 </definitions>
```

Project: examples/jaxws/Managers
Path: generated/slides/PassengerManager.java

```

1
2 package com.example.generated;
3
4 import javax.jws.WebMethod;
5 import javax.jws.WebParam;
6 import javax.jws.WebResult;
7 import javax.jws.WebService;
8 import javax.xml.bind.annotation.XmlSeeAlso;
9 import javax.xml.ws.RequestWrapper;
10 import javax.xml.ws.ResponseWrapper;
11
12
13 /**
14  * This class was generated by the JAX-WS RI.
15  * JAX-WS RI 2.2-b02-rc1
16  * Generated source version: 2.2
17  *
18 */
19 @WebService(name = "PassengerManager",
20             targetNamespace = "urn://Traveller/")
21 @XmlSeeAlso({
22     ObjectFactory.class
23 })
24 public interface PassengerManager {
25
26     /**
27      *
28      * @param lastName
29      * @param firstName
30      * @return
31      *     returns long
32      */
33     @WebMethod
34     @WebResult(name = "result", targetNamespace = "urn://Traveller/")
35     @RequestWrapper(localName = "addPassenger",
36                     targetNamespace = "urn://Traveller/",
37                     className = "generated.AddPassenger")
38     @ResponseWrapper(localName = "addPassengerResponse",
39                      targetNamespace = "urn://Traveller/",
40                      className = "generated.AddPassengerResponse")
41     public long addPassenger(
42         @WebParam(name = "firstName",
43                   targetNamespace = "urn://Traveller/")
44         String firstName,

```

```
45     @WebParam(name = "lastName",
46                 targetNamespace = "urn://Traveller/")
47     String lastName);
48
49 }
```

Project: examples/jaxws/Managers**Path: src/com/example/jaxws/server/FancyServer.java**

```
1
2 package com.example.jaxws.server;
3
4 import com.sun.net.httpserver.HttpContext;
5 import com.sun.net.httpserver.HttpServer;
6 import java.net.InetSocketAddress;
7 import java.util.concurrent.Executor;
8 import java.util.concurrent.Executors;
9 import javax.xml.ws.Endpoint;
10
11 /**
12 * 
13 * @author education@oracle.com
14 */
15 public class FancyServer {
16     public static
17     void main( String[] args ) throws Exception {
18         HttpServer server =
19             HttpServer.create( new InetSocketAddress( 8080 ), 10 );
20         Executor executor = Executors.newFixedThreadPool( 10 );
21         server.setExecutor( executor );
22         HttpContext context =
23             server.createContext( "/fancyServer" );
24         AirportManager manager = new AirportManager();
25         Endpoint endpoint = Endpoint.create( manager );
26         endpoint.publish( context );
27         server.start();
28     }
29 }
```

Project: examples/jaxws/Managers
Path: src/com/example/jaxws/server/PassengerManager.java

```

1  /*
2   */
3
4  package com.example.jaxws.server;
5
6  import com.example.traveller.dao.pojo.PassengerDAO;
7  import com.example.traveller.model.Passenger;
8  import com.sun.xml.ws.developer.SchemaValidation;
9  import javax.jws.WebService;
10 import javax.xml.ws.Endpoint;
11
12 /**
13 *
14 * @author education@oracle.com
15 */
16 @SchemaValidation
17 @WebService(
18     endpointInterface=
19     "com.example.generated.PassengerManager")
20 public class PassengerManager
21 implements com.example.generated.PassengerManager {
22     public long
23     addPassenger(String firstName, String lastName) {
24         Passenger newPassenger =
25             new Passenger(firstName, lastName, null, null);
26         return dao.add(null, newPassenger).getId();
27     }
28     private PassengerDAO dao = new PassengerDAO();
29 // ...
30     static public void main(String[] args) {
31         String dumpPropertyName =
32             "com.sun.xml.ws.transport.http.HttpAdapter.dump";
33         System.setProperty(dumpPropertyName, true);
34         String url = "http://localhost:8081/passengerManager";
35         if (args.length > 0)
36             url = args[1];
37         PassengerManager manager = new PassengerManager();
38         Endpoint endpoint =
39             Endpoint.publish(url, manager);
40     }
41 }
```

Project: examples/jaxws/Managers
Path: src/xml/CustomPassengerManagerService.wsdl

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <definitions name="CustomPassengerManagerService.wsdl"
3     targetNamespace="urn://Traveller"
4     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
5     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
6     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
7     xmlns:tns="urn://Traveller"
8     xmlns="http://schemas.xmlsoap.org/wsdl/">
9
10    <wsdl:import namespace="urn://Traveller"
11        location="CustomPassengerManagerPort.wsdl"/>
12    <!-- ... -->
13    <jaxws:bindings
14        xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
15        <jaxws:package name="com.example.custom"/>
16    </jaxws:bindings>
17
18    <binding name="CustomPassengerManagerBinding"
19        type="tns:CustomPassengerManager">
20        <soap:binding style="document"
21            transport="http://schemas.xmlsoap.org/soap/http"/>
22        <operation name="addPassenger">
23            <soap:operation/>
24            <input name="input1">
25                <soap:body use="literal"/>
26            </input>
27            <output name="out1">
28                <soap:body use="literal"/>
29            </output>
30        </operation>
31    </binding>
32    <service name="CustomPassengerManagerService">
33        <port name="CustomPassengerManager"
34            binding="tns:CustomPassengerManagerBinding">
35            <soap:address
36                location="http://localhost:8080/customManager"/>
37        </port>
38    </service>
39 </definitions>
```

Project: examples/jaxws/Managers
Path: src/xml/CustomPassengerManagerPort.wsdl

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <definitions name="CustomPassengerManagerPort.wsdl"
3      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
5      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
6      xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
7      xmlns:tns="urn://Traveller"
8      targetNamespace="urn://Traveller"
9      xmlns="http://schemas.xmlsoap.org/wsdl/">
10     <!-- types -->
11
12     <types>
13         <xsd:schema>
14             <xsd:import namespace="urn://Traveller"
15                 schemaLocation="CustomPassengerManagerSchema.xsd"/>
16         </xsd:schema>
17     </types>
18     <message name="addPassengerRequest">
19         <part name="message" element="tns:addPassenger"/>
20     </message>
21     <message name="addPassengerResponse">
22         <part name="message" element="tns:addPassengerResponse"/>
23     </message>
24     <portType name="CustomPassengerManager">
25         <jaxws:bindings>
26             <jaxws:class name="CustomManager"/>
27         </jaxws:bindings>
28         <!-- operations ... -->
29         <operation name="addPassenger">
30             <jaxws:bindings>
31                 <jaxws:method name="add"/>
32             </jaxws:bindings>
33             <!-- messages ... -->
34             <input name="input1" message="tns:addPassengerRequest"/>
35             <output name="out1" message="tns:addPassengerResponse"/>
36         </operation>
37     </portType>
38 </definitions>
```

Project: examples/jaxws/Managers
Path: generated/slides/CustomManager.java

```

1
2 package com.example.custom;
3
4 import javax.jws.WebMethod;
5 import javax.jws.WebParam;
6 import javax.jws.WebResult;
7 import javax.jws.WebService;
8 import javax.xml.bind.annotation.XmlSeeAlso;
9 import javax.xml.ws.RequestWrapper;
10 import javax.xml.ws.ResponseWrapper;
11
12
13 /**
14  * This class was generated by the JAX-WS RI.
15  * JAX-WS RI 2.2-b02-rc1
16  * Generated source version: 2.2
17  *
18 */
19 @WebService(name = "CustomPassengerManager",
20             targetNamespace = "urn://Traveller")
21 @XmlSeeAlso({
22     ObjectFactory.class
23 })
24 public interface CustomManager {
25
26     /**
27      *
28      * @param lastName
29      * @param firstName
30      * @return
31      *         returns long
32      */
33     @WebMethod(operationName = "addPassenger")
34     @WebResult(name = "newid", targetNamespace = "urn://Traveller")
35     @RequestWrapper(localName = "addPassenger",
36                     targetNamespace = "urn://Traveller",
37                     className = "generated.CAddPassenger")
38     @ResponseWrapper(localName = "addPassengerResponse",
39                      targetNamespace = "urn://Traveller",
40                      className = "generated.CAddPassengerResponse")
41     public long add(
42         @WebParam(name = "firstName",
43                   targetNamespace = "urn://Traveller")
44         String firstName,

```

```
45     @WebParam(name = "lastName",
46                 targetNamespace = "urn://Traveller")
47     String lastName);
48
49 }
```

Project: examples/jaxws/Managers
Path: generated/slides/rpcLiteralManager.wsdl

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <definitions name="Managers"
3      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
4      xmlns:tns="http://server.jaxws.example.com/"
5      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
6      xmlns="http://schemas.xmlsoap.org/wsdl/"
7      targetNamespace="http://server.jaxws.example.com/">
8  <!-- ... -->
9  <types>
10     <xsd:schema>
11         <xsd:import
12             namespace="http://jaxb.dev.java.net/array"
13             schemaLocation="http://localhost:8081/rpcLiteralManager?xsd=1"/>
14     </xsd:schema>
15 </types>
16 <!-- ... -->
17 <message name="addAirport">
18     <part name="code" type="xsd:string"/>
19     <part name="name" type="xsd:string"/>
20 </message>
21 <message name="addAirportResponse">
22     <part name="return" type="xsd:long"/>
23 </message>
24 <message name="findNeighbors">
25     <part name="code" type="xsd:string"/>
26 </message>
27 <message name="findNeighborsResponse">
28     <part xmlns:ns1="http://jaxb.dev.java.net/array"
29         name="return" type="ns1:stringArray"/>
30 </message>
31 <!-- ... -->
32 <portType name="RPCLiteralAirportManager">
33     <operation name="addAirport"
34         parameterOrder="code_name">
35         <input message="tns:addAirport"/>
36         <output message="tns:addAirportResponse"/>
37     </operation>
38     <operation name="findNeighbors">
39         <input message="tns:findNeighbors"/>
40         <output message="tns:findNeighborsResponse"/>
41     </operation>
42 </portType>
43 <!-- ... -->
44 <binding name="AirportMgrBinding">
```

```

45      type="tns:RPCLiteralAirportManager">
46      <soap:binding style="rpc"
47          transport="http://schemas.xmlsoap.org/soap/http"/>
48      <operation name="addAirport">
49          <soap:operation soapAction=""></soap:operation>
50          <input>
51              <soap:body use="literal"
52                  namespace="http://server.jaxws.example.com/">
53          </input>
54          <output>
55              <soap:body use="literal"
56                  namespace="http://server.jaxws.example.com/">
57          </output>
58      </operation>
59      <operation name="findNeighbors">
60          <soap:operation soapAction=""></soap:operation>
61          <input>
62              <soap:body use="literal"
63                  namespace="http://server.jaxws.example.com/">
64          </input>
65          <output>
66              <soap:body use="literal"
67                  namespace="http://server.jaxws.example.com/">
68          </output>
69      </operation>
70  </binding>
71  <!-- ... -->
72  <service name="Managers">
73      <port name="AirportMgr" binding="tns:AirportMgrBinding">
74          <soap:address
75              location="http://localhost:8081/rpcLiteralManager"/>
76      </port>
77  </service>
78 </definitions>

```

Project: examples/jaxws/Managers
Path: generated/slides/rpcLiteralManager.xsd

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema version="1.0"
3     xmlns:xs="http://www.w3.org/2001/XMLSchema"
4     targetNamespace="http://jaxb.dev.java.net/array">
5     <xs:complexType name="stringArray" final="#all">
6         <xs:sequence>
7             <xs:element name="item"
8                 minOccurs="0" maxOccurs="unbounded"
9                 nillable="true" type="xs:string" />
10            </xs:sequence>
11        </xs:complexType>
12    </xs:schema>
```

Project: examples/jaxws/Managers**Path: src/com/example/jaxws/server/RPCLiteralAirportManager.java**

```

1
2 package com.example.jaxws.server;
3
4 import com.example.traveller.dao.pojo.AirportDAO;
5 import com.example.traveller.model.Airport;
6 import java.util.ArrayList;
7 import java.util.List;
8 import javax.jws.WebMethod;
9 import javax.jws.WebParam;
10 import javax.jws.WebService;
11 import javax.jws.soap.SOAPBinding;
12 import javax.jws.soap.SOAPBinding.ParameterStyle;
13 import javax.jws.soap.SOAPBinding.Style;
14 import javax.jws.soap.SOAPBinding.Use;
15 import javax.xml.ws.Endpoint;
16 import javax.xml.ws.RequestWrapper;
17 import javax.xml.ws.ResponseWrapper;
18
19 /**
20 *
21 * @author education@oracle.com
22 */
23 @WebService(portName="AirportMgr",
24             serviceName="ManagersRPCLit")
25 @SOAPBinding(style=Style.RPC,
26             parameterStyle=ParameterStyle.WRAPPED,
27             use=Use.LITERAL)
28 public class RPCLiteralAirportManager {
29     @WebMethod
30     @RequestWrapper(className="generated.RPCLAMAddAirportRequest")
31     @ResponseWrapper(className="generated.RPCLAMAddAirportResponse")
32     public long
33         addAirport(@WebParam(name="code") String code,
34                    @WebParam(name="name") String name) {
35         return dao.add(null, code, name).getId();
36     }
37     @WebMethod
38     public String[]
39         findNeighbors(
40             @WebParam(name="code") String code ) {
41         Airport[] neighbors =
42             dao.findNeighbors( null, code );
43         if (neighbors.length == 0)
44             return new String[]{ "JFK", "SFO" };

```

```
45     List<String> result = new ArrayList<String>();
46     for ( Airport neighbor : neighbors )
47         result.add( neighbor.getCode() );
48     return
49         result.toArray( new String[0] );
50     }
51 private AirportDAO dao = new AirportDAO();
52 // ...
53 static public void
54 main(String[] args) {
55     String url =
56         "http://localhost:8080/rpcLiteralManager";
57     if (args.length > 0)
58         url = args[1];
59     RPCLiteralAirportManager manager =
60         new RPCLiteralAirportManager();
61     Endpoint endpoint =
62         Endpoint.publish(url, manager);
63 }
64 }
```

Project: examples/jaxws/Managers
Path: generated/slides/docLiteralManager.wsdl

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <definitions name="Managers"
3      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
4      xmlns:tns="http://server.jaxws.example.com/"
5      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
6      xmlns="http://schemas.xmlsoap.org/wsdl/"
7      targetNamespace="http://server.jaxws.example.com/" >
8  <types>
9      <xsd:schema>
10         <xsd:import namespace="http://server.jaxws.example.com/" schemaLocation=
11             "http://localhost:8081/docLiteralManager?xsd=1"/>
12     </xsd:schema>
13     <xsd:schema>
14         <xsd:import namespace="http://jaxb.dev.java.net/array" schemaLocation=
15             "http://localhost:8081/docLiteralManager?xsd=2"/>
16     </xsd:schema>
17 </types>
18 <!-- ... -->
19 <message name="addAirport">
20     <part name="airport" element="tns:airport"/>
21 </message>
22 <message name="addAirportResponse">
23     <part name="addAirportResponse" element="tns:addAirportResponse"/>
24 </message>
25 <message name="findNeighbors">
26     <part name="code" element="tns:code"/>
27 </message>
28 <message name="findNeighborsResponse">
29     <part name="findNeighborsResponse" element="tns:findNeighborsResponse"/>
30 </message>
31 <!-- ... -->
32 <portType name="DocumentLiteralAirportManager">
33     <operation name="addAirport">
34         <input message="tns:addAirport"/>
35         <output message="tns:addAirportResponse"/>
36     </operation>
37     <operation name="findNeighbors">
38         <input message="tns:findNeighbors"/>
39         <output message="tns:findNeighborsResponse"/>
40     </operation>
41 
```

```
45 </portType>
46 <!-- ... -->
47 <binding name="AirportMgrBinding"
48     type="tns:DocumentLiteralAirportManager">
49     <soap:binding style="document"
50         transport="http://schemas.xmlsoap.org/soap/http"/>
51     <operation name="addAirport">
52         <soap:operation soapAction="" />
53         <input><soap:body use="literal" /></input>
54         <output><soap:body use="literal" /></output>
55     </operation>
56     <operation name="findNeighbors">
57         <soap:operation soapAction="" />
58         <input><soap:body use="literal" /></input>
59         <output><soap:body use="literal" /></output>
60     </operation>
61 </binding>
62 <!-- ... -->
63 <service name="Managers">
64     <port name="AirportMgr"
65         binding="tns:AirportMgrBinding">
66         <soap:address
67             location="http://localhost:8081/docLiteralManager" />
68     </port>
69 </service>
70 </definitions>
```

Project: examples/jaxws/Managers
Path: generated/slides/docLiteralManager.xsd

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema version="1.0"
3   xmlns:tns="http://server.jaxws.example.com/"
4   xmlns:ns1="http://jaxb.dev.java.net/array"
5   xmlns:xs="http://www.w3.org/2001/XMLSchema"
6   targetNamespace="http://server.jaxws.example.com/">
7
8 <xs:import namespace="http://jaxb.dev.java.net/array"
9   schemaLocation=
10  "http://localhost:8081/docLiteralManager?xsd=2"/>
11
12 <xs:element name="addAirportResponse" type="xs:long"/>
13 <xs:element name="airport" nillable="true"
14   type="tns:airport"/>
15 <xs:element name="code" nillable="true" type="xs:string"/>
16 <xs:element name="findNeighborsResponse" nillable="true"
17   type="ns1:stringArray"/>
18 <xs:complexType name="airport">
19   <xs:complexContent>
20     <xs:extension base="tns:domainEntity">
21       <xs:sequence>
22         <xs:element name="code" type="xs:string" minOccurs="0"/>
23         <xs:element name="name" type="xs:string" minOccurs="0"/>
24       </xs:sequence>
25     </xs:extension>
26   </xs:complexContent>
27 </xs:complexType>
28
29 <xs:complexType name="domainEntity">
30   <xs:sequence>
31     <xs:element name="id" type="xs:long"/>
32     <xs:element name="version" type="xs:int" minOccurs="0"/>
33   </xs:sequence>
34 </xs:complexType>
35 </xs:schema>
```

Project: examples/jaxws/Managers**Path: src/com/example/jaxws/server/DocumentLiteralAirportManager.java**

```

1
2 package com.example.jaxws.server;
3
4 import com.example.traveller.dao.pojo.AirportDAO;
5 import com.example.traveller.model.Airport;
6 import java.util.ArrayList;
7 import java.util.List;
8 import javax.jws.WebMethod;
9 import javax.jws.WebParam;
10 import javax.jws.WebService;
11 import javax.jws.soap.SOAPBinding;
12 import javax.jws.soap.SOAPBinding.ParameterStyle;
13 import javax.jws.soap.SOAPBinding.Style;
14 import javax.jws.soap.SOAPBinding.Use;
15 import javax.xml.ws.Endpoint;
16 import javax.xml.ws.RequestWrapper;
17 import javax.xml.ws.ResponseWrapper;
18
19 /**
20  * 
21  * @author education@oracle.com
22  */
23 @WebService(portName="DocLiteralAirportMgr",
24             serviceName="ManagersDoc")
25 @SOAPBinding(style=Style.DOCUMENT,
26             parameterStyle=ParameterStyle.BARE,
27             use=Use.LITERAL)
28 public class DocumentLiteralAirportManager {
29     @WebMethod
30     @RequestWrapper(className="generated.DlAMAddAirportRequest")
31     @ResponseWrapper(className="generated.DlAMAddAirportResponse")
32     public long
33         addAirport(@WebParam(name="airport")
34                   Airport airport) {
35         return dao.add(null, airport.getCode(), airport.getName()).getId();
36     }
37     @WebMethod
38     public String[]
39         findNeighbors(
40             @WebParam(name="code") String code) {
41         Airport[] neighbors =
42             dao.findNeighbors(null, code);
43         if (neighbors.length == 0)
44             return new String[]{ "JFK", "SFO" };

```

```
45     List<String> result = new ArrayList<String>();
46     for ( Airport neighbor : neighbors )
47         result.add( neighbor.getCode() );
48     return
49         result.toArray( new String[0] );
50     }
51 private AirportDAO dao = new AirportDAO();
52 // ...
53 static public void
54 main(String[] args) {
55     String url =
56         "http://localhost:8080/docLiteralManager";
57     if (args.length > 0)
58         url = args[1];
59     DocumentLiteralAirportManager manager =
60         new DocumentLiteralAirportManager();
61     Endpoint endpoint =
62         Endpoint.publish(url, manager);
63 }
64 }
```

Project: examples/jaxws/Managers
Path: generated/slides/wrappedAirportManager.wsdl

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <definitions name="AirportManagerService"
3      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
4      xmlns:tns="http://server.jaxws.example.com/"
5      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
6      xmlns="http://schemas.xmlsoap.org/wsdl/"
7      targetNamespace="http://server.jaxws.example.com/" >
8  <types>
9      <xsd:schema>
10         <xsd:import
11             namespace="http://server.jaxws.example.com/"
12             schemaLocation="wrappedAirportManager.xsd"/>
13     </xsd:schema>
14 </types>
15 <message name="addAirport">
16     <part name="parameters"
17         element="tns:addAirport"/>
18 </message>
19 <message name="addAirportResponse">
20     <part name="parameters"
21         element="tns:addAirportResponse"/>
22 </message>
23 <portType name="AirportManager">
24     <operation name="addAirport">
25         <input message="tns:addAirport"/>
26         <output message="tns:addAirportResponse"/>
27     </operation>
28 </portType>
29 <binding name="WrappedAirportManagerPortBinding"
30     type="tns:AirportManager">
31     <soap:binding style="document"
32         transport="http://schemas.xmlsoap.org/soap/http"/>
33     <jaxws:bindings
34         xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
35         <jaxws:enableWrapperStyle>
36             true
37         </jaxws:enableWrapperStyle>
38     </jaxws:bindings>
39     <operation name="addAirport">
40         <soap:operation soapAction="" />
41         <input><soap:body use="literal"/></input>
42         <output><soap:body use="literal"/></output>
43     </operation>
44 </binding>
```

```
45 <service name="AirportManagerService">
46   <port name="AirportManagerPort"
47     binding="tns:WrappedAirportManagerPortBinding">
48     <soap:address
49       location="http://localhost:8081/wrappedAirportManager"/>
50   </port>
51 </service>
52 </definitions>
```

Project: examples/jaxws/Managers
Path: generated/slides/wrappedAirportManager.xsd

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema version="1.0"
3     xmlns:tns="http://server.jaxws.example.com/"
4     xmlns:xss="http://www.w3.org/2001/XMLSchema"
5     targetNamespace="http://server.jaxws.example.com/">
6
7 <xs:element name="addAirport" type="tns:addAirport"/>
8 <xs:element name="addAirportResponse"
9     type="tns:addAirportResponse"/>
10 <xs:complexType name="addAirport">
11     <xs:sequence>
12         <xs:element name="code" type="xs:string"
13             minOccurs="0"/>
14         <xs:element name="name" type="xs:string"
15             minOccurs="0"/>
16     </xs:sequence>
17 </xs:complexType>
18 <xs:complexType name="addAirportResponse">
19     <xs:sequence>
20         <xs:element name="return" type="xs:long"/>
21     </xs:sequence>
22 </xs:complexType>
23
24 </xs:schema>
```

Oscar Hernando Nivia Aragon (onvia@hotmail.com) has a
non-transferable license to use this Student Guide.

Project: examples/jaxws/JavaEEJAXWS
Path: src/java/com/example/jaxws/server/AirportManager.java

```
1
2 package com.example.jaxws.server;
3
4 import com.example.traveller.dao.pojo.AirportDAO;
5 import javax.jws.WebService;
6 import javax.xml.ws.Endpoint;
7
8 /**
9  * AirportManager Web Service, deployed to a web container.
10 * @author education@oracle.com
11 */
12 @WebService(serviceName="AirportManagerWS")
13 public class AirportManager {
14     public long
15         addAirport(String code, String name) {
16             return dao.add(null, code, name).getId();
17         }
18     private AirportDAO dao = new AirportDAO();
19 }
```

Project: examples/jaxws/JavaEEJAXWS**Path: src/java/com/example/jaxws/server/SecureAirportManager.java**

```
1
2 package com.example.jaxws.server;
3
4 import com.example.traveller.dao.pojo.AirportDAO;
5 import javax.annotation.Resource;
6 import javax.annotation.security.DeclareRoles;
7 import javax.annotation.security.PermitAll;
8 import javax.annotation.security.RolesAllowed;
9 import javax.jws.WebMethod;
10 import javax.jws.WebService;
11 import javax.servlet.ServletContext;
12 import javax.xml.ws.WebServiceContext;
13 import javax.xml.ws.handler.MessageContext;
14
15 /**
16  * AirportManager Web Service, deployed to a web container.
17  * @author education@oracle.com
18  */
19 @WebService(serviceName="SecureManagerWS")
20 @DeclareRoles({"client","administrator"})
21 public class SecureAirportManager {
22     @WebMethod @RolesAllowed("administrator")
23     public long addAirport(String code, String name) {
24         String user = context.getUserPrincipal().getName();
25         MessageContext msgContext = context.getMessageContext();
26         ServletContext webContext = (ServletContext)
27             msgContext.get(MessageContext.SERVLET_CONTEXT);
28         webContext.log("add_requested_by:_ " + user);
29         return dao.add(null, code, name).getId();
30     }
31     @WebMethod @PermitAll
32     public String getNameByCode(String code) {
33         return dao.findByCode(null, code).getName();
34     }
35     // ...
36     private AirportDAO dao = new AirportDAO();
37     @Resource WebServiceContext context;
38 }
```

Project: examples/jaxws/JavaEEJAXWS
Path: web/WEB-INF/web.xml

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app version="3.0"
3      xmlns="http://java.sun.com/xml/ns/javaee"
4      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5      xsi:schemaLocation=
6          "http://java.sun.com/xml/ns/javaee
7          http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
8      <session-config>
9          <session-timeout>
10             30
11         </session-timeout>
12     </session-config>
13     <security-constraint>
14         <display-name>SecureManagerWS</display-name>
15         <web-resource-collection>
16             <web-resource-name>
17                 SecureManagerWS
18             </web-resource-name>
19             <url-pattern>/SecureManagerWS</url-pattern>
20             <http-method>POST</http-method>
21         </web-resource-collection>
22         <auth-constraint>
23             <role-name>administrator</role-name>
24             <role-name>client</role-name>
25         </auth-constraint>
26     </security-constraint>
27     <security-constraint>
28         <display-name>SecureServlet</display-name>
29         <web-resource-collection>
30             <web-resource-name>
31                 SecureServlet
32             </web-resource-name>
33             <url-pattern>/SecureServlet</url-pattern>
34             <http-method>GET</http-method>
35         </web-resource-collection>
36         <auth-constraint>
37             <role-name>administrator</role-name>
38             <role-name>client</role-name>
39         </auth-constraint>
40     </security-constraint>
41     <login-config>
42         <auth-method>BASIC</auth-method>
43         <realm-name>file</realm-name>
44     </login-config>
```

```
45    <security-role>
46        <role-name>client</role-name>
47    </security-role>
48    <security-role>
49        <role-name>administrator</role-name>
50    </security-role>
51 </web-app>
```

Project: examples/jaxws/JavaEEJAXWS
Path: web/WEB-INF/sun-web.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <sun-web-app error-url="">
3   <context-root>/JavaEEJAXWS</context-root>
4   <security-role-mapping>
5     <role-name>client</role-name>
6     <principal-name>tracy</principal-name>
7   </security-role-mapping>
8   <security-role-mapping>
9     <role-name>administrator</role-name>
10    <principal-name>kelly</principal-name>
11  </security-role-mapping>
12  <class-loader delegate="true"/>
13  <jsp-config>
14    <property name="keepgenerated" value="true"/>
15  </jsp-config>
16 </sun-web-app>
```

Project: examples/jaxws/JavaEEJAXWS
Path: src/java/com/example/servlets/SecureServlet.java

```

1  /*
2   */
3
4  package com.example.servlets;
5
6  import java.io.IOException;
7  import java.io.PrintWriter;
8  import java.security.Principal;
9  import javax.servlet.ServletException;
10 import javax.servlet.annotation.WebServlet;
11 import javax.servlet.http.HttpServlet;
12 import javax.servlet.http.HttpServletRequest;
13 import javax.servlet.http.HttpServletResponse;
14
15 /**
16 *
17 * @author education@oracle.com
18 */
19 @WebServlet(name="SecureServlet",
20             urlPatterns={"/SecureServlet"})
21 public class SecureServlet extends HttpServlet {
22
23     /**
24      * Processes requests for HTTP POST and GET methods.
25      * @param request servlet request
26      * @param response servlet response
27      * @throws ServletException if a servlet-specific error occurs
28      * @throws IOException if an I/O error occurs
29     */
30     protected void
31     processRequest(HttpServletRequest request,
32                     HttpServletResponse response)
33     throws ServletException, IOException {
34         response.setContentType("text/html; charset=UTF-8");
35         PrintWriter out = response.getWriter();
36         String user = request.getRemoteUser();
37         Principal principal = request.getUserPrincipal();
38         try {
39             out.println("<html>");
40             out.println("<body>");
41             out.println("<h1>Servlet_SecureServlet_at_"
42                         + request.getContextPath () + "</h1>");
43             out.println("User:_" + user);
44             out.println("Principal:_" + principal);

```

```

45         out.println("</body>");
46         out.println("</html>");
47     } finally {
48         out.close();
49     }
50 }
51 /**
52 * Handles the HTTP <code>GET</code> method.
53 * @param request servlet request
54 * @param response servlet response
55 * @throws ServletException if a servlet-specific error occurs
56 * @throws IOException if an I/O error occurs
57 */
58
59 @Override
60 protected void doGet(HttpServletRequest req,
61                         HttpServletResponse resp)
62 throws ServletException, IOException {
63     processRequest(req, resp);
64 }
65 /**
66 * Handles the HTTP <code>POST</code> method.
67 * @param request servlet request
68 * @param response servlet response
69 * @throws ServletException if a servlet-specific error occurs
70 * @throws IOException if an I/O error occurs
71 */
72
73 @Override
74 protected void doPost(HttpServletRequest req,
75                         HttpServletResponse resp)
76 throws ServletException, IOException {
77     processRequest(req, resp);
78 }
79 /**
80 * Returns a short description of the servlet.
81 * @return a String containing servlet description
82 */
83
84 @Override
85 public String getServletInfo() {
86     return "Short_description";
87 } // </editor-fold>
88
89 }

```

Project: examples/jaxws/JavaEEJAXWS
Path: src/java/com/example/jaxws/ejb/AirportManager.java

```
1
2 package com.example.jaxws.ejb;
3
4 import com.example.traveller.dao.ejb.AirportDAO;
5 import javax.ejb.EJB;
6 import javax.ejb.Stateless;
7 import javax.jws.WebService;
8
9 /**
10 * AirportManager Web Service, deployed to a web container.
11 * @author education@oracle.com
12 */
13 @WebService(serviceName="AirportManagerEJBWS")
14 @Stateless
15 public class AirportManager {
16     public long
17         addAirport(String code, String name) {
18             return dao.add(code, name).getId();
19         }
20     @EJB private AirportDAO dao;
21 }
```

Project: examples/jaxws/JavaEEJAXWS
Path: src/java/com/example/jaxws/ejb/BetterAirportManager.java

```

1
2 package com.example.jaxws.ejb;
3
4 import com.example.traveller.dao.ejb.AirportDAO;
5 import com.example.traveller.dao.ejb.GenericDAO;
6 import javax.ejb.EJB;
7 import javax.ejb.Stateless;
8 import javax.jws.WebMethod;
9 import javax.jws.WebService;
10 import javax.persistence.EntityManager;
11 import javax.persistence.EntityTransaction;
12 import javax.xml.ws.RequestWrapper;
13 import javax.xml.ws.ResponseWrapper;
14
15 /**
16 *
17 * @author education@oracle.com
18 */
19 @WebService(serviceName="BetterManagerEJBWS")
20 @Stateless
21 public class BetterAirportManager {
22     @WebMethod
23     @RequestWrapper(className="generated.BAMAddAirportRequest")
24     @ResponseWrapper(className="generated.BAMAddAirportResponse")
25     public long
26         addAirport(String code, String name) {
27             return dao.add(code, name).getId();
28         }
29     @WebMethod(operationName="removeById")
30     @RequestWrapper(className="generated.BAMRemoveByIdRequest")
31     @ResponseWrapper(className="generated.BAMRemoveByIdResponse")
32     public void removeAirport(long id) {
33         dao.remove(id);
34     }
35     @WebMethod(operationName="removeByCode")
36     @RequestWrapper(className="generated.BAMRemoveByCodeRequest")
37     @ResponseWrapper(className="generated.BAMRemoveByCodeResponse")
38     public void removeAirport(String code) {
39         dao.remove(dao.findByCode(code));
40     }
41     @EJB private AirportDAO dao;
42     // ...
43 }
```

Project: examples/jaxws/JavaEEJAXWS**Path: src/java/com/example/jaxws/ejb/SingletonAirportManager.java**

```

1
2 package com.example.jaxws.ejb;
3
4 import com.example.traveller.dao.ejb.AirportDAO;
5 import javax.ejbConcurrencyManagement;
6 import javax.ejbConcurrencyManagementType;
7 import javax.ejb.EJB;
8 import javax.ejb.Lock;
9 import javax.ejb.LockType;
10 import javax.ejb.Singleton;
11 import javax.jws.WebMethod;
12 import javax.jws.WebService;
13 import javax.xml.ws.RequestWrapper;
14 import javax.xml.ws.ResponseWrapper;
15
16 /**
17 *
18 * @author education@oracle.com
19 */
20 @WebService(serviceName="SingletonManagerEJBWS")
21 @Singleton
22 @Lock(LockType.WRITE)
23 public class SingletonAirportManager {
24     @WebMethod
25     @RequestWrapper(className="generated.SAMAddAirportRequest")
26     @ResponseWrapper(className="generated.SAMAddAirportResponse")
27     public long addAirport(String code, String name) {
28         return dao.add(code, name).getId();
29     }
30     @WebMethod(operationName="removeById")
31     @RequestWrapper(className="generated.SAMRemoveByIdRequest")
32     @ResponseWrapper(className="generated.SAMRemoveByIdResponse")
33     public void removeAirport(long id) {
34         dao.remove(id);
35     }
36     @WebMethod(operationName="removeByCode")
37     @RequestWrapper(className="generated.SAMRemoveByCodeRequest")
38     @ResponseWrapper(className="generated.SAMRemoveByCodeResponse")
39     public void removeAirport(String code) {
40         dao.remove(dao.findByCode(code));
41     }
42     @WebMethod
43     @Lock(LockType.READ)
44     public String getNameByCode(String code) {

```

```
45     return dao.findByName(code).getName();  
46 }  
47 @EJB private AirportDAO dao;  
48 // ...  
49 }
```

Project: projects/Traveller/projects/TravellerEntities
Path: src/com/example/traveller/dao/pojo/AirportDAO.java

```

1  /*
2   */
3
4  package com.example.traveller.dao.pojo;
5
6  import com.example.traveller.model.Airport;
7  import java.util.List;
8  import javax.persistence.EntityManager;
9  import javax.persistence.EntityTransaction;
10 import javax.persistence.TypedQuery;
11
12 /**
13  * Data Access Object implementation for persistent
14  * class {@see com.example.traveller.model.Airport}.
15  * @author education@oracle.com
16  */
17 public class AirportDAO extends GenericDAO<Airport> {
18
19 /**
20  * Adds a new Airport to the persistent store.
21  * If <code>em</code> is not <code>null</code>, the function
22  * assumes that the caller has already begun a transaction,
23  * and that the caller will commit/rollback as appropriate.
24  * @param em EntityManager to use
25  * @param code airport code for new Airport
26  * @param name airport name for new Airport
27  * @return newly created persistent Airport.
28  * If the function had to create its own persistence context,
29  * the return instance will be detached, as its persistence
30  * context will have disappeared by the time the function returns.
31  */
32 public
33     Airport add(EntityManager em, String code, String name) {
34         return add( em, new Airport( code, name ) );
35     }
36
37 /**
38  * Find a persistent Airport instance by its airport code.
39  * If <code>em</code> is not <code>null</code>, the function
40  * assumes that the caller has already begun a transaction,
41  * and that the caller will commit/rollback as appropriate.
42  * @param em EntityManager to use
43  * @param code airport code for Airport to be found
44  * @return persistent Airport by given code.

```

```

45     * If the function had to create its own persistence context,
46     * the return instance will be detached, as its persistence
47     * context will be gone by the time the function returns.
48     */
49     public Airport findByCode(EntityManager em, String code) {
50         EntityTransaction tx = null;
51         if (em == null) {
52             em = getEMF().createEntityManager();
53             tx = em.getTransaction();
54         }
55         if (tx != null) tx.begin();
56         String sqlQuery =
57             "SELECT_a_from_Airport_a WHERE_a.code_a=:code";
58         TypedQuery<Airport> query =
59             em.createQuery(sqlQuery, Airport.class);
60         query.setParameter( "code", code );
61         List<Airport> results = query.getResultList();
62         if (tx != null) {
63             tx.commit();
64             em.close();
65         }
66         if (results.size() != 1)
67             throw new QueryException();
68         return results.get(0);
69     }
70
71     public Airport[] findByName(EntityManager em, String name) {
72         EntityTransaction tx = null;
73         if (em == null) {
74             em = getEMF().createEntityManager();
75             tx = em.getTransaction();
76         }
77         if (tx != null) tx.begin();
78         String sqlQuery =
79             "SELECT_a_FROM_Airport_a WHERE_a.name_a LIKE_a:_name";
80         TypedQuery<Airport> query =
81             em.createQuery( sqlQuery, Airport.class );
82         query.setParameter( "name", "%" + name + "%" );
83         List<Airport> results = query.getResultList();
84         if (tx != null) {
85             tx.commit();
86             em.close();
87         }
88         return results.toArray( new Airport[0] );
89     }
90
91

```

```

92  /**
93   * Removes the persistent Airport instance by the given code.
94   * If <code>em</code> is not <code>null</code>, the function
95   * assumes that the caller has already begun a transaction,
96   * and that the caller will commit/rollback as appropriate.
97   * @param em EntityManager to use
98   * @param code airport code for persistent instance to remove
99   */
100  public void removeByCode(EntityManager em, String code) {
101      EntityTransaction tx = null;
102      if (em == null) {
103          em = getEMF().createEntityManager();
104          tx = em.getTransaction();
105      }
106      if (tx != null) tx.begin();
107      remove(em, findByCode(em, code));
108      if (tx != null) {
109          tx.commit();
110          em.close();
111      }
112  }
113
114  public Airport[] findNeighbors(EntityManager em, String code) {
115      EntityTransaction tx = null;
116      if (em == null) {
117          em = getEMF().createEntityManager();
118          tx = em.getTransaction();
119      }
120      String sqlQuery =
121          "SELECT DISTINCT f.arrives FROM Flight f"
122          + " WHERE f.departs.code = :source";
123      TypedQuery<Airport> query =
124          em.createQuery(sqlQuery, returnedClass());
125      query.setParameter("source", code);
126      if (tx != null) tx.begin();
127      List<Airport> neighbors = query.getResultList();
128      if (tx != null) {
129          tx.commit();
130          em.close();
131      }
132      return neighbors.toArray(new Airport[0]);
133  }
134
135  /**
136   * Retrieves all airport codes known.
137   * @param em - EntityManager to use for operation, if known
138   * @return list of airport codes known, as Strings

```

```

139     */
140     public List<String> getAllCodes( EntityManager em ) {
141         EntityTransaction tx = null;
142         if (em == null) {
143             em = getEMF().createEntityManager();
144             tx = em.getTransaction();
145         }
146         String sqlQuery =
147             "SELECT DISTINCT a.code FROM Airport a";
148         TypedQuery<String> query =
149             em.createQuery( sqlQuery, String.class );
150         if (tx != null) tx.begin();
151         List<String> codes = query.getResultList();
152         if (tx != null) {
153             tx.commit();
154             em.close();
155         }
156         return codes;
157     }
158
159 /**
160 * Retrieves all airports known.
161 * @param em - EntityManager to use for operation, if known
162 * @return list of airports known.
163 */
164 public List<Airport> list( EntityManager em ) {
165     EntityTransaction tx = null;
166     if (em == null) {
167         em = getEMF().createEntityManager();
168         tx = em.getTransaction();
169     }
170     String sqlQuery = "SELECT a FROM Airport a";
171     TypedQuery<Airport> query =
172         em.createQuery( sqlQuery, Airport.class );
173     if (tx != null) tx.begin();
174     List<Airport> as = query.getResultList();
175     if (tx != null) {
176         tx.commit();
177         em.close();
178     }
179     return as;
180 }
181 }

```

Project: projects/Traveller/projects/TravellerEntities
Path: src/com/example/traveller/dao/pojo/GenericDAO.java

```

1  /*
2   */
3
4  package com.example.traveller.dao.pojo;
5
6  import com.example.traveller.model.DomainEntity;
7  import java.lang.reflect.ParameterizedType;
8  import java.util.List;
9  import javax.persistence.EntityManager;
10 import javax.persistence.EntityManagerFactory;
11 import javax.persistence.EntityTransaction;
12 import javax.persistence.Persistence;
13 import javax.persistence.Query;
14 import javax.persistence.TypedQuery;
15
16 /**
17  * Generic DAO implementation which supports basic operations for
18  * classes derived from com.example.traveller.model.DomainEntity.
19  * The implementation used only supports classes that are direct
20  * children of this class (see {@see #returnedClass}).
21  * @param <PersistentClass> type of DomainEntity to be supported
22  * in a given child class.
23  * @author education@oracle.com
24 */
25 public abstract
26 class GenericDAO<PersistentClass extends DomainEntity> {
27
28     private static final EntityManagerFactory emf =
29         Persistence.createEntityManagerFactory( "TravellerEntitiesPU" );
30
31 /**
32  * Returns global singleton EntityManagerFactory for persistent
33  * unit named "TravellerEntitiesPU".
34  * @return singleton EntityManagerFactory
35 */
36     public static EntityManagerFactory getEMF() {
37         return emf;
38     }
39
40 /**
41  * Inserts instance into persistent store.
42  * If <code>em</code> is <code>null</code>, the call will create
43  * its own local <code>EntityManager</code> and persistent context,
44  * and will use a local transaction to perform the insert.

```

```

45     * If <code>em</code> is not <code>null</code>, the function
46     * assumes that the caller has already begun a transaction,
47     * and that the caller will commit/rollback as appropriate.
48     * @param em EntityManager to use
49     * @param newObj instance to insert into persistent store
50     * @return the instance inserted into persistent store
51     */
52     public PersistentClass add(EntityManager em,
53                               PersistentClass newObj) {
54         EntityTransaction tx = null;
55         if (em == null) {
56             em = getEMF().createEntityManager();
57             tx = em.getTransaction();
58         }
59         if (tx != null) tx.begin();
60         em.persist( newObj );
61         if (tx != null) {
62             tx.commit();
63             em.close();
64         }
65         return newObj;
66     }
67
68     /**
69      * Updates the persistent store representation of the instance
70      * provided. If <code>em</code> is <code>null</code>, the call
71      * will create its own local <code>EntityManager</code> and
72      * persistent context, and will use a local transaction to
73      * perform the insert. The instance to update must be one of:
74      * <ul>
75      *   <li> a persistent instance in the current persistence context
76      *       (in which case the call is redundant)
77      *   <li> a detached instance, with a valid id and updated values
78      *   <li> a transient instance, with a valid id and updated values
79      * </ul>
80      * If the instance is persistent in the current persistence
81      * context, it is returned. Otherwise, a new persistent instance
82      * with updated values is returned.
83      * If <code>em</code> is not <code>null</code>, the function
84      * assumes that the caller has already begun a transaction, and
85      * that the caller will commit/rollback as appropriate.
86      * If the function had to create its own persistence context,
87      * the return instance will be detached, as its persistence
88      * context will have disappeared by the time the function returns.
89      * @param em EntityManager to use
90      * @param instance instance to update.
91      * @return an updated persistent instance

```

```

92     */
93     public PersistentClass update(EntityManager em,
94                                     PersistentClass instance) {
95         EntityTransaction tx = null;
96         if (em == null) {
97             em = getEMF().createEntityManager();
98             tx = em.getTransaction();
99         }
100        if (tx != null) tx.begin();
101        PersistentClass result = em.merge( instance );
102        if (tx != null) {
103            tx.commit();
104            em.close();
105        }
106        return result;
107    }
108
109 /**
110 * Finds a persistent instance, given its unique id.
111 * If <code>em</code> is <code>null</code>, the call will create
112 * its own local <code>EntityManager</code> and persistent context,
113 * and will use a local transaction to perform the insert.
114 * If <code>em</code> is not <code>null</code>, the function
115 * assumes that the caller has already begun a transaction, and
116 * that the caller will commit/rollback as appropriate.
117 * If the function had to create its own persistence context, the
118 * return instance will be detached, as its persistence context
119 * will have disappeared by the time the function returns.
120 * @param em EntityManager to use
121 * @param id unique id for instance to return
122 * @return persistent instance identified by id provided
123 */
124     public PersistentClass find( EntityManager em, long id ) {
125         EntityTransaction tx = null;
126         if (em == null) {
127             em = getEMF().createEntityManager();
128             tx = em.getTransaction();
129         }
130         if (tx != null) tx.begin();
131         PersistentClass result = em.find( returnedClass(), id );
132         if (tx != null) {
133             tx.commit();
134             em.close();
135         }
136         return result;
137     }
138

```

```

139 /**
140 * Finds all persistent instances that match the query provided.
141 * If <code>em</code> is <code>null</code>, the call will create
142 * its own local <code>EntityManager</code> and persistent context,
143 * and will use a local transaction to perform the insert.
144 * If <code>em</code> is not <code>null</code>, the function
145 * assumes that the caller has already begun a transaction, and
146 * that the caller will commit/rollback as appropriate.
147 * If the function had to create its own persistence context, the
148 * return value will consist of detached instances, as their
149 * persistence context will be gone when the function returns.
150 * @param em EntityManager to use
151 * @param jpaQL JPA QL query to execute
152 * @return list of persistent instances that match the query
153 */
154 public List<PersistentClass> query(EntityManager em, String jpaQL) {
155     EntityTransaction tx = null;
156     if (em == null) {
157         em = getEMF().createEntityManager();
158         tx = em.getTransaction();
159     }
160     if (tx != null) tx.begin();
161     TypedQuery<PersistentClass> query =
162         em.createQuery(jpaQL, returnedClass());
163     List<PersistentClass> result = query.getResultList();
164     if (tx != null) {
165         tx.commit();
166         em.close();
167     }
168     return result;
169 }
170 /**
171 * Removes an obj from the persistent store, given its unique id.
172 * If <code>em</code> is <code>null</code>, the call will create
173 * its own local <code>EntityManager</code> and persistent context,
174 * and will use a local transaction to perform the insert.
175 * If <code>em</code> is not <code>null</code>, the function
176 * assumes that the caller has already begun a transaction, and
177 * that the caller will commit/rollback as appropriate.
178 * @param em EntityManager to use
179 * @param id unique id of persistent instance to remove
180 */
181 public void remove( EntityManager em, long id ) {
182     EntityTransaction tx = null;
183     if (em == null) {
184         em = getEMF().createEntityManager();

```

```

186         tx = em.getTransaction();
187     }
188     if (tx != null) tx.begin();
189     em.remove( em.getReference( returnedClass(), id ) );
190     if (tx != null) {
191         tx.commit();
192         em.close();
193     }
194 }
195 /**
196 * Removes the instance provided from the persistent store.
197 * If <code>em</code> is <code>null</code>, the call will create
198 * its own local <code>EntityManager</code> and persistent context,
199 * and will use a local transaction to perform the insert.
200 * If <code>em</code> is not <code>null</code>, the function
201 * assumes that the caller has already begun a transaction, and
202 * that the caller will commit/rollback as appropriate.
203 * @param em EntityManager to use
204 * @param instance to remove
205 */
206 public void remove(EntityManager em, PersistentClass instance) {
207     EntityTransaction tx = null;
208     if (em == null) {
209         em = getEMF().createEntityManager();
210         tx = em.getTransaction();
211     }
212     if (tx != null) tx.begin();
213     em.remove( instance );
214     if (tx != null) {
215         tx.commit();
216         em.close();
217     }
218 }
219 /**
220 * Returns the parameterized type used when extending directly
221 * from this generic class, as an instance of class Class. This
222 * return value can be used when such an instance is needed by
223 * Java methods, such as those in javax.persistence.EntityManager.
224 * The trick used here will only work when the child class directly
225 * extends this one. A complete discussion of this trick can be
226 * found here:
227 * http://www.artima.com/weblogs/viewpost.jsp?thread=208860
228 * @return parameterized type used to instance class, as an
229 * instance of class Class.
230 */
231
232

```

```
233     @SuppressWarnings("unchecked")
234     protected Class<PersistentClass> returnedClass() {
235         ParameterizedType parameterizedType =
236             (ParameterizedType) getClass().getGenericSuperclass();
237         return (Class<PersistentClass>)
238             (parameterizedType.getActualTypeArguments()[0]);
239     }
240
241     /**
242      *
243      */
244     public static class QueryException extends RuntimeException {
245         private static final long serialVersionUID = 0L;
246     }
247
248 }
```

Project: projects/Traveller/projects/TravellerEJBs
Path: src/java/com/example/traveller/dao/ejb/GenericDAO.java

```

1  /*
2   */
3
4  package com.example.traveller.dao.ejb;
5
6  import com.example.traveller.model.DomainEntity;
7  import java.lang.reflect.ParameterizedType;
8  import java.util.List;
9  import javax.ejb.Stateless;
10 import javax.ejb.TransactionAttribute;
11 import javax.ejb.TransactionAttributeType;
12 import javax.persistence.EntityManager;
13 import javax.persistence.PersistenceContext;
14 import javax.persistence.TypedQuery;
15
16 /**
17  * Generic DAO implementation which supports basic operations for
18  * classes derived from com.example.traveller.model.DomainEntity.
19  * The implementation used only supports classes that are direct
20  * children of this class (see {@see #returnedClass}).
21  * @param <PersistentClass> type of DomainEntity to be supported
22  * in a given child class.
23  * @author education@oracle.com
24 */
25
26 @TransactionAttribute(TransactionAttributeType.REQUIRED)
27 public abstract
28 class GenericDAO<PersistentClass extends DomainEntity> {
29
30     @PersistenceContext(unitName="TravellerEJBsPU")
31     protected EntityManager em;
32
33 /**
34  * Inserts instance into persistent store.
35  * @param newInstance instance to insert into persistent store
36  * @return the instance inserted into persistent store
37 */
38 public PersistentClass add(PersistentClass newInstance) {
39     em.persist(newInstance);
40     return newInstance;
41 }
42
43 /**
44  * Updates the persistent store representation of the instance

```

```

45     * provided. The instance to update must be one of:
46     * <ul>
47     *   <li> a persistent instance in the current persistence
48     *   context (in which case the call is redundant)
49     *   <li> a detached instance, updated, with a valid id
50     *   <li> a transient instance, updated, with a valid id
51     * </ul>
52     * If the instance provided is persistent in the current
53     * persistence context, it is returned. Otherwise, a new
54     * persistent instance with updated values is returned.
55     * @param instance instance to update.
56     * @return an updated persistent instance
57   */
58   public PersistentClass update(PersistentClass instance) {
59     PersistentClass result = em.merge(instance);
60     return result;
61   }
62
63 /**
64  * Finds a persistent instance, given its unique id.
65  * @param id unique id for instance to return
66  * @return persistent instance identified by id provided
67 */
68   public PersistentClass find( long id ) {
69     PersistentClass result = em.find( returnedClass(), id );
70     return result;
71   }
72
73 /**
74  * Finds all persistent instances that match the query provided.
75  * @param jpaQL JPA QL query to execute
76  * @return list of persistent instances that match the query
77 */
78   public List<PersistentClass> query( String jpaQL ) {
79     TypedQuery<PersistentClass> query =
80       em.createQuery( jpaQL, returnedClass() );
81     List<PersistentClass> result = query.getResultList();
82     return result;
83   }
84
85 /**
86  * Removes an instance from the persistent store, given its id.
87  * @param id unique id of persistent instance to remove
88 */
89   public void remove( long id ) {
90     em.remove( em.getReference( returnedClass(), id ) );
91   }

```

```

92
93     /**
94      * Removes the instance provided from the persistent store.
95      * @param instance to remove
96      */
97     public void remove( PersistentClass instance ) {
98         em.remove( instance );
99     }
100
101    /**
102     * Returns the parameterized type used when extending directly
103     * from this generic class, as an instance of class Class. This
104     * return value can be used when such an instance is needed by
105     * Java methods, such as those in javax.persistence.EntityManager.
106     * The trick used here will only work when the child class directly
107     * extends this one. A complete discussion of this trick can be
108     * found here:
109     * http://www.artima.com/weblogs/viewpost.jsp?thread=208860
110     * @return parameterized type used to instance class, as an
111     * instance of class Class.
112     */
113     @SuppressWarnings("unchecked")
114     protected Class<PersistentClass> returnedClass() {
115         ParameterizedType parameterizedType =
116             (ParameterizedType) getClass().getGenericSuperclass();
117         return (Class<PersistentClass>)
118             (parameterizedType.getActualTypeArguments()[0]);
119     }
120
121    /**
122     *
123     */
124     public static class QueryException extends RuntimeException {
125         private static final long serialVersionUID = 0L;
126     }
127
128 }
```

Project: projects/Traveller/projects/TravellerEJBs
Path: src/java/com/example/traveller/dao/ejb/AirportDAO.java

```

1  /*
2   */
3
4  package com.example.traveller.dao.ejb;
5
6  import com.example.traveller.model.Airport;
7  import java.util.List;
8  import javax.ejb.Stateless;
9  import javax.persistence.TypedQuery;
10
11 /**
12  * Data Access Object implementation for persistent
13  * class {@see com.example.traveller.model.Airport}.
14  * @author education@oracle.com
15  */
16 @Stateless
17 public class AirportDAO extends GenericDAO<Airport> {
18
19 /**
20  * Adds a new Airport to the persistent store.
21  * @param code airport code for new Airport
22  * @param name airport name for new Airport
23  * @return newly created persistent Airport.
24 */
25 public Airport add( String code, String name ) {
26     return add( new Airport( code, name ) );
27 }
28
29 /**
30  * Find a persistent Airport instance by its airport code.
31  * @param code airport code for Airport to be found
32  * @return persistent Airport by given code.
33 */
34 public Airport findByCode( String code ) {
35     String sqlQuery =
36         "SELECT_a_from_Airport_a_WHERE_a.code_=:_code";
37     TypedQuery<Airport> query =
38         em.createQuery(sqlQuery,Airport.class);
39     query.setParameter( "code", code );
40     List<Airport> results = query.getResultList();
41     if (results.size() != 1)
42         throw new QueryException();
43     return results.get(0);
44 }
```

```

45
46     public Airport[] findByName( String name ) {
47         String sqlQuery =
48             "SELECT_a_FROM_Airport_a_WHERE_a.name_LIKE_:name";
49         TypedQuery<Airport> query =
50             em.createQuery(sqlQuery, Airport.class);
51         query.setParameter( "name", "%" + name + "%" );
52         List<Airport> results = query.getResultList();
53         return results.toArray( new Airport[0] );
54     }
55
56 /**
57 * Removes the persistent Airport instance by the given code.
58 * @param code airport code for persistent instance to remove
59 */
60 public void removeByCode( String code ) {
61     Airport instance = findByCode( code );
62     remove( instance );
63 }
64
65 public Airport[] findNeighbors( String code ) {
66     String sqlQuery =
67         "SELECT_DISTINCT_f.arrives_FROM_Flight_f"
68         + "WHERE_f.departs.code_=:_source";
69     TypedQuery<Airport> query =
70         em.createQuery( sqlQuery, returnedClass() );
71     query.setParameter( "source", code );
72     List<Airport> neighbors = query.getResultList();
73     return neighbors.toArray( new Airport[0] );
74 }
75
76 /**
77 * Retrieves all airport codes known.
78 * @return list of airport codes known, as Strings
79 */
80 public List<String> getAllCodes() {
81     String sqlQuery =
82         "SELECT_DISTINCT_a.code_FROM_Airport_a";
83     TypedQuery<String> query =
84         em.createQuery( sqlQuery, String.class );
85     List<String> codes = query.getResultList();
86     return codes;
87 }
88
89 /**
90 * Retrieves all airports known.
91 * @param em - EntityManager to use for operation, if known

```

```
92     * @return list of airports known.  
93     */  
94     public List<Airport> list() {  
95         String sqlQuery = "SELECT _a FROM Airport _a";  
96         TypedQuery<Airport> query =  
97             em.createQuery( sqlQuery, Airport.class );  
98         List<Airport> as = query.getResultList();  
99         return as;  
100    }  
101 }
```

Project: examples/jaxws/JavaEEJAXWS
Path: src/java/com/example/jaxws/server/FlightManager.java

```

1
2 package com.example.jaxws.server;
3
4 import com.example.traveller.dao.ejb.AirportDAO;
5 import com.example.traveller.dao.ejb.FlightDAO;
6 import com.example.traveller.model.Flight;
7 import java.util.Date;
8 import javax.jws.WebMethod;
9 import javax.jws.WebService;
10 import javax.xml.ws.Endpoint;
11
12 /**
13 * 
14 * @author education@oracle.com
15 */
16 @WebService
17 public class FlightManager {
18     @WebMethod
19     public Flight
20         addFlight(String airline, String number,
21                 String departsPort, Date departsTime,
22                 String arrivesPort, Date arrivesTime) {
23         Flight flight = new
24             Flight(airline, number,
25                     airportDAO.findByName(departsPort),
26                     airportDAO.findByName(arrivesPort),
27                     departsTime, arrivesTime, 150);
28         return flightDao.add(flight);
29     }
30     @WebMethod(operationName="simpleAddFlight")
31     public Flight
32         addFlight(String airline, String number,
33                 String departsPort, String arrivesPort) {
34         Date now = new Date();
35         Flight flight = new
36             Flight(airline, number,
37                     airportDAO.findByName(departsPort),
38                     airportDAO.findByName(arrivesPort),
39                     now, now, 150);
40         return flightDao.add(flight);
41     }
42     @WebMethod
43     public Flight getFlight(String airline, long id) {
44         return flightDao.find(id);

```

```
45    }
46    private FlightDAO flightDao = new FlightDAO();
47    private AirportDAO airportDAO = new AirportDAO();
48    // ...
49    static public void main(String[] args) {
50        String dumpPropertyName =
51            "com.sun.xml.ws.transport.http.HttpAdapter.dump";
52        System.setProperty(dumpPropertyName, "true");
53        String url =
54            "http://localhost:8081/flightManager";
55        if (args.length > 0)
56            url = args[1];
57        FlightManager manager = new FlightManager();
58        Endpoint endpoint =
59            Endpoint.publish(url, manager);
60    }
61 }
```

Project: examples/jaxws/Managers
Path: generated/xml/FlightManager.xsd

```

1   <xs:schema version="1.0"
2       targetNamespace="http://server.jaxws.example.com/">
3
4       <xs:element name="addFlight" type="tns:addFlight"/>
5       <xs:element name="addFlightResponse"
6           type="tns:addFlightResponse"/>
7       <xs:element name="airport" type="tns:airport"/>
8       <xs:element name="getFlight" type="tns:getFlight"/>
9       <xs:element name="getFlightResponse"
10          type="tns:getFlightResponse"/>
11       <xs:element name="simpleAddFlight" type="tns:addFlight"/>
12       <xs:element name="simpleAddFlightResponse"
13          type="tns:addFlightResponse"/>
14
15       <xs:complexType name="getFlight">
16           <xs:sequence>
17               <xs:element name="arg0" type="xs:string" />
18               <xs:element name="arg1" type="xs:long" />
19           </xs:sequence>
20       </xs:complexType>
21       <xs:complexType name="getFlightResponse">
22           <xs:sequence>
23               <xs:element name="return" type="tns:flight" />
24           </xs:sequence>
25       </xs:complexType>
26
27       <xs:complexType name="flight">
28           <xs:complexContent>
29               <xs:extension base="tns:domainEntity">
30                   <xs:sequence>
31                       <xs:element name="departs" type="tns:airport" />
32                       <xs:element name="arrives" type="tns:airport" />
33                       <xs:element name="departure" type="xs:dateTime" />
34                       <xs:element name="arrival" type="xs:dateTime" />
35                       <xs:element name="airline" type="xs:string" />
36                       <xs:element name="number" type="xs:string" />
37                       <xs:element name="maxSeats" type="xs:int" />
38                       <xs:element name="tickets" type="tns:ticket"
39                           maxOccurs="unbounded" />
40                   </xs:sequence>
41               </xs:extension>
42           </xs:complexContent>
43       </xs:complexType>
44   </xs:complexType>
```

```

45
46    <xs:complexType name="domainEntity">
47        <xs:sequence/>
48            <xs:attribute name="id" type="xs:long" use="required"/>
49            <xs:attribute name="version" type="xs:int"/>
50    </xs:complexType>
51
52    <xs:complexType name="airport">
53        <xs:complexContent>
54            <xs:extension base="tns:domainEntity">
55                <xs:sequence>
56                    <xs:element name="code" type="xs:string" />
57                    <xs:element name="name" type="xs:string" />
58                </xs:sequence>
59            </xs:extension>
60        </xs:complexContent>
61    </xs:complexType>
62
63    <xs:complexType name="ticket">
64        <xs:complexContent>
65            <xs:extension base="tns:domainEntity">
66                <xs:sequence>
67                    <xs:element name="issueDate" type="xs:dateTime" />
68                    <xs:element name="confirmationCode" type="xs:string"/>
69                    <xs:element name="price" type="xs:double"/>
70                    <xs:element name="flights" type="tns:flight"
71                        minOccurs="0" maxOccurs="unbounded"/>
72                    <xs:element name="passenger" type="tns:passenger" />
73                        <xs:element name="confirmed" type="xs:boolean"/>
74                        <xs:element name="payment" type="tns:payment" />
75                </xs:sequence>
76            </xs:extension>
77        </xs:complexContent>
78    </xs:complexType>
79
80    <xs:complexType name="passenger">
81        <xs:complexContent>
82            <xs:extension base="tns:domainEntity">
83                <xs:sequence>
84                    <xs:element name="firstName" type="xs:string" />
85                    <xs:element name="lastName" type="xs:string" />
86                    <xs:element name="phoneNumber" type="xs:string" />
87                    <xs:element name="freqFlyerId" type="xs:string" />
88                    <xs:element name="tickets" type="tns:ticket"
89                        minOccurs="0" maxOccurs="unbounded"/>
90                </xs:sequence>
91            </xs:extension>

```

```
92      </xs:complexContent>
93  </xs:complexType>
94
95  <xs:complexType name="payment">
96    <xs:complexContent>
97      <xs:extension base="tns:domainEntity">
98        <xs:sequence>
99          <xs:element name="ticket" type="tns:ticket"/>
100         <xs:element name="creditCardNum" type="xs:string"/>
101         <xs:element name="bankName" type="xs:string"/>
102         <xs:element name="expirationDate" type="xs:dateTime"/>
103         <xs:element name="status" type="tns:status"/>
104       </xs:sequence>
105     </xs:extension>
106   </xs:complexContent>
107 </xs:complexType>
108
109 <xs:complexType name="addFlight">
110   <xs:sequence>
111     <xs:element name="arg0" type="xs:string" />
112     <xs:element name="arg1" type="xs:string" />
113     <xs:element name="arg2" type="xs:string" />
114     <xs:element name="arg3" type="xs:dateTime" />
115     <xs:element name="arg4" type="xs:string" />
116     <xs:element name="arg5" type="xs:dateTime" />
117   </xs:sequence>
118 </xs:complexType>
119
120 <xs:complexType name="addFlightResponse">
121   <xs:sequence>
122     <xs:element name="return" type="tns:flight" />
123   </xs:sequence>
124 </xs:complexType>
125
126 <xs:simpleType name="status">
127   <xs:restriction base="xs:string">
128     <xs:enumeration value="pending"/>
129     <xs:enumeration value="processing"/>
130     <xs:enumeration value="accepted"/>
131     <xs:enumeration value="rejected"/>
132   </xs:restriction>
133 </xs:simpleType>
134 </xs:schema>
```

Project: examples/jaxws/Managers
Path: src/xml/SaferPassengerManagerPort.wsdl

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <definitions name="SaferPassengerManagerPort.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
10 <types>
11     <xsd:schema>
12         <xsd:import namespace="urn://Traveller/"
13             schemaLocation="SaferPassengerManagerSchema.xsd"/>
14     </xsd:schema>
15 </types>
16
17 <message name="addPassengerRequest">
18     <part name="parameters" element="tns:addPassenger"/>
19 </message>
20 <message name="addPassengerResponse">
21     <part name="parameters" element="tns:addPassengerResponse"/>
22 </message>
23 <message name="addPassengerFault">
24     <part name="parameters" element="tns:addPassengerFault"/>
25 </message>
26 <portType name="SaferPassengerManager">
27     <operation name="addPassenger">
28         <input name="input1" message="tns:addPassengerRequest"/>
29         <output name="out1" message="tns:addPassengerResponse"/>
30         <fault name="fault1" message="tns:addPassengerFault"/>
31     </operation>
32 </portType>
33 </definitions>
```

Project: examples/jaxws/Managers
Path: src/xml/SaferPassengerManagerSchema.xsd

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsd:schema id="SaferPassengerManagerSchema.xsd"
3      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4      xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
5      jxb:version="1.0"
6      xmlns:tns="urn://Traveller/"
7      elementFormDefault="qualified"
8      targetNamespace="urn://Traveller/">
9
10 <xsd:element name="addPassenger">
11     <xsd:complexType>
12         <xsd:sequence>
13             <xsd:element name="firstName" type="xsd:string"/>
14             <xsd:element name="lastName" type="xsd:string"/>
15         </xsd:sequence>
16     </xsd:complexType>
17 </xsd:element>
18
19 <xsd:element name="addPassengerResponse">
20     <xsd:complexType>
21         <xsd:sequence>
22             <xsd:element name="result" type="xsd:long"/>
23         </xsd:sequence>
24     </xsd:complexType>
25 </xsd:element>
26
27 <xsd:element name="addPassengerFault">
28     <xsd:complexType>
29         <xsd:sequence>
30             <xsd:element name="duplicateCode" type="xsd:string"/>
31             <xsd:element name="duplicateName" type="xsd:string"/>
32         </xsd:sequence>
33     </xsd:complexType>
34 </xsd:element>
35
36 <xsd:annotation>
37     <xsd:appinfo>
38         <jxb:schemaBindings>
39             <jxb:package name="com.example.safe"/>
40         </jxb:schemaBindings>
41     </xsd:appinfo>
42 </xsd:annotation>
43 </xsd:schema>
```

Project: examples/jaxws/Managers
Path: generated/slides/AddPassengerFault.java

```

1
2 package traveller;
3
4 import javax.xml.bind.annotation.XmlAccessType;
5 import javax.xml.bind.annotation.XmlAccessorType;
6 import javax.xml.bind.annotation.XmlElement;
7 import javax.xml.bind.annotation.XmlRootElement;
8 import javax.xml.bind.annotation.XmlType;
9
10 /**
11  * <p>Java class for anonymous complex type.
12  *
13  * <p>The following schema fragment specifies the expected content
14  * contained within this class.
15  * <pre>
16  * <complexType>
17  *   <complexContent>
18  *     <restriction
19  *       base="{http://www.w3.org/2001/XMLSchema}anyType">
20  *       <sequence>
21  *         <element name="duplicateCode"
22  *           type="{http://www.w3.org/2001/XMLSchema}string"/>
23  *         <element name="duplicateName"
24  *           type="{http://www.w3.org/2001/XMLSchema}string"/>
25  *       </sequence>
26  *     </restriction>
27  *   </complexContent>
28  * </complexType>
29  * </pre>
30  */
31 @XmlAccessorType(XmlAccessType.FIELD)
32 @XmlType(name = "", propOrder = {
33     "duplicateCode",
34     "duplicateName"
35 })
36 @XmlRootElement(name = "addPassengerFault")
37 public class AddPassengerFault {
38
39     @XmlElement(required = true)
40     protected String duplicateCode;
41     @XmlElement(required = true)
42     protected String duplicateName;
43
44 /**

```

```
45     * Gets the value of the duplicateCode property.
46     *
47     * @return
48     *     possible object is
49     *     {@link String }
50     *
51     */
52 public String getDuplicateCode() {
53     return duplicateCode;
54 }
55
56 /**
57 * Sets the value of the duplicateCode property.
58 *
59 * @param value
60 *     allowed object is
61 *     {@link String }
62 *
63 */
64 public void setDuplicateCode(String value) {
65     this.duplicateCode = value;
66 }
67
68 /**
69 * Gets the value of the duplicateName property.
70 *
71 * @return
72 *     possible object is
73 *     {@link String }
74 */
75 public String getDuplicateName() {
76     return duplicateName;
77 }
78
79 /**
80 * Sets the value of the duplicateName property.
81 *
82 * @param value
83 *     allowed object is
84 *     {@link String }
85 */
86 public void setDuplicateName(String value) {
87     this.duplicateName = value;
88 }
89
90 }
```

Project: projects/Traveller/projects/TravellerEntities
Path: src/com/example/traveller/model/Airport.java

```

1  package com.example.traveller.model;
2
3  import java.io.Serializable;
4  import javax.persistence.Basic;
5  import javax.persistence.Column;
6  import javax.persistence.Entity;
7  import javax.xml.bind.annotation.XmlRootElement;
8  import javax.xml.bind.annotation.XmlType;
9
10 /**
11  * @(#) Airport.java
12  */
13
14 @Entity
15 @XmlRootElement
16 public class Airport
17     extends DomainEntity
18     implements Serializable {
19     @Basic(optional=false) @Column(updatable=false, unique=true)
20     private String code;
21     private String name;
22
23     public Airport() {}
24
25     public Airport( String code, String name ) {
26         this.code = code; this.name = name;
27     }
28
29     @Override
30     public boolean equals( Object obj ) {
31         if ((obj == null) || (getClass() != obj.getClass())) {
32             return false;
33         }
34         final Airport other = (Airport) obj;
35         if ((id != 0) && (other.id != 0) && (id != other.id)) {
36             return false;
37         }
38         if ((code == null) ?
39             (other.code != null) : !code.equals( other.code )) {
40             return false;
41         }
42         return true;
43     }
44

```

```

45     @Override
46     public int hashCode() {
47         int hash = 7;
48         hash = 53 * hash + (code != null ? code.hashCode() : 0);
49         return hash;
50     }
51
52     /**
53      * @return the code
54      */
55     public String getCode() {
56         return code;
57     }
58
59     /**
60      * @param code the code to set
61      */
62     public void setCode( String code ) {
63         this.code = code;
64     }
65
66     /**
67      * @return the name
68      */
69     public String getName() {
70         return name;
71     }
72
73     /**
74      * @param name the name to set
75      */
76     public void setName( String name ) {
77         this.name = name;
78     }
79
80     /**
81      * Construct string representation of instance.
82      * @return string representation of instance.
83      */
84     public String toString() {
85         return "[Airport_code:'" + code + "'_name:'" + name + "']";
86     }
87
88     private static final long serialVersionUID = 0L;
89 }
```

Project: projects/Traveller/projects/TravellerEntities
Path: src/com/example/traveller/model/DomainEntity.java

```

1  /*
2   */
3
4  package com.example.traveller.model;
5
6  import javax.persistence.GeneratedValue;
7  import javax.persistence.GenerationType;
8  import javax.persistence.Id;
9  import javax.persistence.MappedSuperclass;
10 import javax.persistence.Version;
11 import javax.xml.bind.annotation.XmlAccessType;
12 import javax.xml.bind.annotation.XmlAccessorType;
13 import javax.xml.bind.annotation.XmlAttribute;
14
15 /**
16 *
17 * @author education@oracle.com
18 */
19 @MappedSuperclass
20 @XmlAccessorType(XmlAccessType.FIELD)
21 public class DomainEntity {
22     @Id @GeneratedValue(strategy=GenerationType.AUTO)
23     @XmlAttribute
24     protected long id;
25     @Version
26     @XmlAttribute()
27     protected Integer version;
28
29 /**
30 * @return the id
31 */
32     public long getId() {
33         return id;
34     }
35
36 /**
37 * @param id the id to set
38 */
39     public void setId( long id ) {
40         this.id = id;
41     }
42
43 /**
44 * @return the version

```

```
45      */
46  public Integer getVersion() {
47      return version;
48  }
49
50 /**
51 * @param version the version to set
52 */
53 public void setVersion( Integer version ) {
54     this.version = version;
55 }
56 }
```

Project: projects/Traveller/projects/TravellerEntities
Path: src/com/example/traveller/model/Payment.java

```

1  package com.example.traveller.model;
2
3  import java.io.Serializable;
4  import java.util.Date;
5  import javax.persistence.Basic;
6  import javax.persistence.Entity;
7  import javax.persistence.OneToOne;
8  import javax.persistence.Temporal;
9  import javax.persistence.TemporalType;
10 import javax.xml.bind.annotation.XmlEnum;
11 import javax.xml.bind.annotation.XmlType;
12
13 /**
14  * @(#) Payment.java
15 */
16 @Entity
17 @XmlType
18 public class Payment extends DomainEntity
19     implements Serializable {
20
21     @XmlAttribute(String.class)
22     public static enum Status {
23         pending, processing, accepted, rejected
24     };
25
26     @OneToOne(mappedBy = "payment")
27     private Ticket ticket;
28     @Basic(optional = false)
29     private String creditCardNum;
30     @Basic(optional = false)
31     private String bankName;
32     @Temporal(TemporalType.DATE)
33     @Basic(optional = false)
34     private Date expirationDate;
35     private Status status = Status.pending;
36
37     public Payment() {
38     }
39
40     public Payment(String bankName, String ccNum, Date expDate) {
41         this.bankName = bankName;
42         creditCardNum = ccNum;
43         expirationDate = expDate;
44     }

```

```

45
46     /**
47      * @return the ticket
48      */
49  public Ticket getTicket() {
50      return ticket;
51  }
52
53 /**
54  * @param ticket the ticket to set
55  */
56  public void setTicket(Ticket ticket) {
57      this.ticket = ticket;
58  }
59
60 /**
61  * @return the creditCardNum
62  */
63  public String getCreditCardNum() {
64      return creditCardNum;
65  }
66
67 /**
68  * @param creditCardNum the creditCardNum to set
69  */
70  public void setCreditCardNum(String creditCardNum) {
71      this.creditCardNum = creditCardNum;
72  }
73
74 /**
75  * @return the bankName
76  */
77  public String getBankName() {
78      return bankName;
79  }
80
81 /**
82  * @param bankName the bankName to set
83  */
84  public void setBankName(String bankName) {
85      this.bankName = bankName;
86  }
87
88 /**
89  * @return the expirationDate
90  */
91  public Date getExpirationDate() {

```

```
92         return expirationDate;
93     }
94
95     /**
96      * @param expirationDate the expirationDate to set
97      */
98     public void setExpirationDate(Date expirationDate) {
99         this.expirationDate = expirationDate;
100    }
101    private static final long serialVersionUID = 0L;
102 }
```

Project: examples/jaxws/JAXWSClients
Path: src/clients/SimpleClient.java

```
1  /*
2   */
3
4  package clients;
5
6  /**
7   *
8   * @author education@oracle.com
9   */
10 public class SimpleClient {
11     public static void main(String[] args) {
12         AirportManagerService service =
13             new AirportManagerService();
14         AirportManager port =
15             service.getAirportManagerPort();
16         java.lang.String code = "LGA";
17         java.lang.String name = "New_York_LaGuardia";
18         long result = port.addAirport(code, name);
19         System.out.println("Result = " + result);
20     }
21 }
```

Project: examples/jaxws/JAXWSClients
Path: src/xml/stricterAirportManager.xsd

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema version="1.0"
3   xmlns:tns="http://server.jaxws.example.com/"
4   xmlns:xs="http://www.w3.org/2001/XMLSchema"
5   targetNamespace="http://server.jaxws.example.com/">
6
7 <xs:element name="addAirport" type="tns:addAirport"/>
8 <xs:element name="addAirportResponse"
9   type="tns:addAirportResponse"/>
10
11 <xs:complexType name="addAirport">
12   <xs:sequence>
13     <xs:element name="arg0" type="tns:code" minOccurs="0"/>
14     <xs:element name="arg1" type="xs:string" minOccurs="0"/>
15   </xs:sequence>
16 </xs:complexType>
17
18 <xs:simpleType name="code">
19   <xs:restriction base="xs:string">
20     <xs:pattern value="\w{3}"/>
21   </xs:restriction>
22 </xs:simpleType>
23
24 <xs:complexType name="addAirportResponse">
25   <xs:sequence>
26     <xs:element name="return" type="xs:long"/>
27   </xs:sequence>
28 </xs:complexType>
29
30 </xs:schema>
```

Project: examples/jaxws/JAXWSClients
Path: src/xml/stricterAirportManager.wsdl

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <definitions name="AirportManagerService"
3      xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
4      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
5      xmlns:tns="http://server.jaxws.example.com/"
6      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
7      xmlns="http://schemas.xmlsoap.org/wsdl/"
8      targetNamespace="http://server.jaxws.example.com/">
9  <types>
10     <xsd:schema>
11         <xsd:import
12             namespace="http://server.jaxws.example.com/"
13             schemaLocation="stricterAirportManager.xsd">
14         </xsd:import>
15     </xsd:schema>
16 </types>
17     <message name="addAirport">
18         <part name="parameters" element="tns:addAirport"/>
19     </message>
20     <message name="addAirportResponse">
21         <part name="parameters" element="tns:addAirportResponse"></part>
22     </message>
23     <portType name="AirportManager">
24         <operation name="addAirport">
25             <input message="tns:addAirport"
26                 wsam:Action=
27                 "http://server.jaxws.example.com/AirportManager/addAirportRequest"/>
28             <output message="tns:addAirportResponse"
29                 wsam:Action=
30                 "http://server.jaxws.example.com/AirportManager/addAirportResponse"/>
31         </operation>
32     </portType>
33     <binding name="AirportManagerPortBinding"
34             type="tns:AirportManager">
35         <soap:binding style="document"
36             transport="http://schemas.xmlsoap.org/soap/http"/>
37         <operation name="addAirport">
38             <soap:operation soapAction="" />
39             <input>
40                 <soap:body use="literal"/>
41             </input>
42             <output>
43                 <soap:body use="literal"/>
44             </output>
```

```
45    </operation>
46  </binding>
47 <service name="StricterAirportManagerService">
48   <port name="AirportManagerPort"
49     binding="tns:AirportManagerPortBinding">
50     <soap:address
51       location="http://localhost:8081/airportManager"/>
52   </port>
53 </service>
54 </definitions>
```

Project: examples/jaxws/JAXWSClients
Path: src/clients/StricterClient.java

```
1  /*
2   */
3
4  package clients;
5
6  import com.example.jaxws.server.StricterAirportManager;
7  import com.example.jaxws.server.StricterAirportManagerService;
8  import com.sun.xml.ws.developer.SchemaValidationFeature;
9  import javax.xml.ws.WebServiceFeature;
10
11 /**
12  *
13  * @author education@oracle.com
14  */
15 public class StricterClient {
16     public static void main(String[] args) {
17         StricterAirportManagerService service =
18             new StricterAirportManagerService();
19         WebServiceFeature validation =
20             new SchemaValidationFeature();
21         StricterAirportManager port =
22             service.getAirportManagerPort(validation);
23         java.lang.String code =
24             (args.length >= 1) ? args[0] : "NYLGA";
25         java.lang.String name =
26             (args.length >= 2) ? args[1] : "New_York_LaGuardia";
27         long result = port.addAirport(code, name);
28         System.out.println("Result_=_" + result);
29     }
30 }
```

Project: examples/mod02/UserDirectoryEJB
Path: src/java/com/example/userDirectory/ejb/UserDirectoryBean.java

```

1
2 package com.example.userDirectory.ejb;
3
4 import com.example.security.server.HelperService;
5 import javax.ejb.Stateless;
6 import javax.jws.WebMethod;
7 import javax.jws.WebService;
8 import javax.xml.ws.WebServiceRef;
9
10 /**
11 *
12 * @author education@oracle.com
13 */
14 @Stateless @WebService
15 public class UserDirectoryBean
16     implements UserDirectoryRemote {
17     @WebServiceRef
18     (wsdlLocation = "http://localhost/securityHelper?WSDL")
19     private HelperService service;
20
21     @WebMethod
22     public boolean addUser(String login,
23                           String name, String email) {
24         // add to registry, if not there...
25         return false;
26     }
27
28     public boolean testPassword(String value)
29     throws Exception {
30         com.example.security.server.Helper port =
31             service.getHelperPort();
32         double result = port.passwordStrength(value);
33         return result > 0.75;
34     }
35 }
```

Project: examples/jaxws/Managers**Path: src/com/example/jaxws/server/OneWayPassengerManager.java**

```

1  /*
2   */
3
4  package com.example.jaxws.server;
5
6  import com.example.traveller.dao.pojo.PassengerDAO;
7  import com.example.traveller.model.Passenger;
8  import javax.jws.Oneway;
9  import javax.jws.WebService;
10 import javax.xml.ws.Endpoint;
11 import javax.xml.ws.RequestWrapper;
12 import javax.xml.ws.ResponseWrapper;
13
14 /**
15 *
16 * @author education@oracle.com
17 */
18 @WebService
19 public class OneWayPassengerManager {
20     @RequestWrapper(className="generated.OWPMAddPassengerRequest")
21     @ResponseWrapper(className="generated.OWPMAddPassengerResponse")
22     public long
23         addPassenger(String firstName, String lastName) {
24         Passenger newPassenger =
25             new Passenger(firstName, lastName, null, null);
26         return dao.add(null, newPassenger).getId();
27     }
28     @Oneway
29     public void removePassenger(long id) {
30         dao.remove(null, id);
31     }
32     private PassengerDAO dao = new PassengerDAO();
33 // ...
34     static public void main(String[] args) {
35         String dumpPropertyName=
36             "com.sun.xml.ws.transport.http.HttpAdapter.dump";
37         System.setProperty(dumpPropertyName, "true");
38         String url = "http://localhost:8081/oneWayPassengerManager";
39         if (args.length > 0)
40             url = args[1];
41         OneWayPassengerManager manager = new OneWayPassengerManager();
42         Endpoint endpoint = Endpoint.publish(url, manager);
43     }
44 }
```

Project: examples/jaxws/JAXWSClients
Path: src/xml/asyncStrictAirportManager.wsdl

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <definitions name="AirportManagerService"
3      xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
4      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
5      xmlns:tns="http://server.jaxws.example.com/"
6      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
7      xmlns="http://schemas.xmlsoap.org/wsdl/"
8      targetNamespace="http://server.jaxws.example.com/">
9  <types>
10    <xsd:schema>
11      <xsd:import namespace="http://server.jaxws.example.com/" 
12          schemaLocation="stricterAirportManager.xsd"/>
13    </xsd:schema>
14  </types>
15  <message name="addAirport">
16    <part name="parameters" element="tns:addAirport"/>
17  </message>
18  <message name="addAirportResponse">
19    <part name="parameters" element="tns:addAirportResponse"/></part>
20  </message>
21  <portType name="AirportManager">
22    <jaxws:bindings
23        xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
24      <jaxws:enableAsyncMapping>
25        true
26      </jaxws:enableAsyncMapping>
27    </jaxws:bindings>
28    <operation name="addAirport">
29      <input message="tns:addAirport"
30          wsam:Action=
31          "http://server.jaxws.example.com/AirportManager/addAirportRequest"/>
32      <output message="tns:addAirportResponse"
33          wsam:Action=
34          "http://server.jaxws.example.com/AirportManager/addAirportResponse"/>
35    </operation>
36  </portType>
37  <binding name="AirportManagerPortBinding"
38      type="tns:AirportManager">
39    <soap:binding style="document"
40        transport="http://schemas.xmlsoap.org/soap/http"/>
41    <operation name="addAirport">
42      <soap:operation soapAction="" />
43      <input><soap:body use="literal"/></input>
44      <output><soap:body use="literal"/></output>
```

```
45    </operation>
46  </binding>
47 <service name="AsyncAirportManagerService">
48   <port name="AirportManagerPort"
49     binding="tns:AirportManagerPortBinding">
50     <soap:address
51       location="http://localhost:8081/airportManager"/>
52   </port>
53 </service>
54 </definitions>
```

Project: examples/jaxws/JAXWSClients
Path: slides/AsyncAirportManager.java

```

1
2 package clients.async;
3
4 import java.util.concurrent.Future;
5 import javax.jws.WebMethod;
6 import javax.jws.WebParam;
7 import javax.jws.WebResult;
8 import javax.jws.WebService;
9 import javax.xml.bind.annotation.XmlSeeAlso;
10 import javax.xml.ws.Action;
11 import javax.xml.ws.AsyncHandler;
12 import javax.xml.ws.RequestWrapper;
13 import javax.xml.ws.Response;
14 import javax.xml.ws.ResponseWrapper;
15
16
17 /**
18 * This class was generated by the JAX-WS RI.
19 * JAX-WS RI 2.2-b02-rc1
20 * Generated source version: 2.2
21 *
22 */
23 @WebService(name = "AirportManager",
24             targetNamespace = "http://server.jaxws.example.com/")
25 @XmlSeeAlso({
26     ObjectFactory.class
27 })
28 public interface AsyncAirportManager {
29
30     /**
31     *
32     * @param code
33     * @param name
34     * @return
35     *     javax.xml.ws.Response<clients.async.AddAirportResponse>
36     */
37     @WebMethod(operationName = "addAirport")
38     @RequestWrapper(localName = "addAirport",
39                     targetNamespace="http://server.jaxws.example.com/",
40                     className = "clients.async.AddAirport")
41     @ResponseWrapper(localName = "addAirportResponse",
42                      targetNamespace="http://server.jaxws.example.com/",
43                      className = "clients.async.AddAirportResponse")
44     public Response<AddAirportResponse>

```

```

45     addAirportAsync(String code, String name);
46
47     /**
48      *
49      * @param code
50      * @param name
51      * @param asyncHandler
52      * @return
53      *    java.util.concurrent.Future<? extends java.lang.Object>
54     */
55     @WebMethod(operationName = "addAirport")
56     @RequestWrapper(localName = "addAirport",
57                     targetNamespace="http://server.jaxws.example.com/",
58                     className = "clients.async.AddAirport")
59     @ResponseWrapper(localName = "addAirportResponse",
60                      targetNamespace="http://server.jaxws.example.com/",
61                      className = "clients.async.AddAirportResponse")
62     public Future<?>
63         addAirportAsync(String code, String name,
64                         @WebParam(name = "asyncHandler")
65                         AsyncHandler<AddAirportResponse> h);
66
67     /**
68      *
69      * @param arg1
70      * @param arg0
71      * @return
72      *    returns long
73     */
74     @WebMethod
75     @WebResult(targetNamespace = "")
76     @RequestWrapper(localName = "addAirport",
77                     targetNamespace="http://server.jaxws.example.com/",
78                     className = "clients.async.AddAirport")
79     @ResponseWrapper(localName = "addAirportResponse",
80                      targetNamespace="http://server.jaxws.example.com/",
81                      className = "clients.async.AddAirportResponse")
82     @Action(
83         input =
84         "http://server.jaxws.example.com/AirportManager/addAirportRequest",
85         output =
86         "http://server.jaxws.example.com/AirportManager/addAirportResponse")
87     public long addAirport(String code, String name);
88
89 }

```

Project: examples/jaxws/JAXWSClients
Path: src/clients/AsyncClientViaResponse.java

```
1  /*
2   */
3
4  package clients;
5
6  import clients.async.AsyncAirportManager;
7  import javax.xml.ws.Response;
8
9  /**
10  *
11  * @author education@oracle.com
12  */
13 public class AsyncClientViaResponse {
14     public static void main(String[] args)
15     throws Exception {
16         clients.async.AirportManagerService service =
17             new clients.async.AirportManagerService();
18         AsyncAirportManager port =
19             service.getAirportManagerPort();
20         java.lang.String code = "LGA";
21         java.lang.String name = "New_York_LaGuardia";
22         Response<clients.async.AddAirportResponse> holder =
23             port.addAirportAsync(code, name);
24         // some other work goes here...
25         long result = holder.get().getReturn();
26         System.out.println("Result_=_" + result);
27     }
28 }
```

Project: examples/jaxws/JAXWSClients
Path: src/clients/AsyncClientViaHandler.java

```

1  /*
2   */
3
4  package clients;
5
6  import clients.async.AsyncAirportManager;
7  import java.util.concurrent.Future;
8  import javax.xml.ws.AsyncHandler;
9  import javax.xml.ws.Response;
10
11 /**
12 *
13 * @author education@oracle.com
14 */
15
16 class AsyncClientHandler implements AsyncHandler {
17     public void handleResponse( Response res ) {
18         Response<clients.async.AddAirportResponse> response =
19             (Response<clients.async.AddAirportResponse>) res;
20         try { System.out.println(response.get().getReturn()); }
21         catch( Exception ex ) {
22             System.out.println( ex.getMessage() );
23         }
24     }
25 }
26
27 public class AsyncClientViaHandler {
28     public static void main(String[] args)
29     throws Exception {
30         clients.async.AirportManagerService service =
31             new clients.async.AirportManagerService();
32         AsyncAirportManager port =
33             service.getAirportManagerPort();
34         java.lang.String code = "BOS";
35         java.lang.String name = "Boston_Logan";
36         AsyncHandler handler = new AsyncClientHandler();
37         Future<?> holder =
38             port.addAirportAsync(code, name, handler);
39         // now we can go do anything else we need...
40         Thread.sleep(30 * 1000);
41     }
42 }
```

Project: examples/jaxrs/ResourceManagers
Path: src/com/example/jaxrs/resources/AirportRM.java

```

1  /*
2   */
3  package com.example.jaxrs.resources;
4
5  import com.example.traveller.dao.pojo.AirportDAO;
6  import com.example.traveller.model.Airport;
7  import com.sun.jersey.api.container.httpserver.HttpServerFactory;
8  import com.sun.jersey.spi.resource.Singleton;
9  import com.sun.net.httpserver.HttpServer;
10 import java.io.IOException;
11 import java.util.List;
12 import javax.ws.rs.FormParam;
13 import javax.ws.rs.GET;
14 import javax.ws.rs.POST;
15 import javax.ws.rs.Path;
16 import javax.ws.rs.PathParam;
17 import javax.ws.rs.Produces;
18 import javax.ws.rs.QueryParam;
19 import javax.ws.rs.core.MediaType;
20
21 /**
22 *
23 * @author education@oracle.com
24 */
25 @Singleton
26 @Path("/airports")
27 public class AirportRM {
28     // ...
29     @GET
30     @Path("/numAirports")
31     public String getNumAirports() {
32         List<String> codes = dao.getAllCodes( null );
33         return
34             (codes == null) ? "0" : "" + codes.size();
35     }
36     // ...
37     @POST
38     @Path("/add")
39     public String addAirport(
40             @FormParam("code") String code,
41             @FormParam("name") String name ) {
42         Airport newAirport = null;
43         try { newAirport = dao.add( null, code, name ); }
44         catch( Exception ex ) {}

```

```

45     return (newAirport != null) ? "ok" : "fail";
46 }
47 // ...
48 @GET
49 @Path ("/nameByCode/{code}")
50 public String getNameByCode(
51     @PathParam("code") String code ) {
52     Airport airport = null;
53     try { airport = dao.findByName( null, code ); }
54     catch( Exception ex ) {
55         System.out.println( "Code:_" + code + "_Ex:_" + ex );
56     }
57     return
58     (airport != null) ? airport.getName() : "(not_found)";
59 }
60
61 @GET
62 @Path ("/codeByName")
63 public String getCodeByName(
64     @QueryParam("name") String name ) {
65     Airport[] airports = dao.findByName( null, name );
66     if ((airports == null) || (airports.length == 0))
67         return "(no_such_airport)";
68     else if (airports.length == 1)
69         return airports[0].getCode();
70     else
71         return "(too_many_candidates)";
72 }
73
74 @GET
75 @Path ("/byCode/{code}")
76 @Produces({"application/xml","application/json"})
77 public Airport getByCode(
78     @PathParam("code") String code ) {
79     return dao.findByCode( null, code );
80 }
81
82
83 @GET
84 @Path ("/xmlByCode/{code}")
85 @Produces(MediaType.APPLICATION_XML)
86 public Airport getXmlByCode(
87     @PathParam("code") String code ) {
88     Airport airport = dao.findByCode( null, code );
89     return airport;
90 }
91

```

```
92     @GET
93     @Path("/jsonByCode/{code}")
94     @Produces(MediaType.APPLICATION_JSON)
95     public Airport getJsonByCode(
96         @PathParam("code") String code) {
97         Airport airport = dao.findByName( null, code );
98         return airport;
99     }
100
101    private AirportDAO dao = new AirportDAO();
102
103    static public void
104    main(String[] args) throws IOException {
105        String contextUrl =
106            "http://localhost:8080/jaxrs";
107        if (args.length > 0)
108            contextUrl = args[1];
109        HttpServer server =
110            HttpServerFactory.create( contextUrl );
111        server.start();
112    }
113    // ...
114 }
```

Oscar Hernando Nivia Aragon (onivia@hotmail.com) has a
non-transferable license to use this Student Guide.

Project: examples/jaxrs/ResourceManagers
Path: src/com/example/jaxrs/resources/MyApplication.java

```
1
2 package com.example.jaxrs.resources;
3
4 import com.example.jaxrs.resources.providers.DAOExceptionMapper;
5 import com.example.jaxrs.resources.providers.MapMessageBodyWriter;
6 import com.example.jaxrs.resources.providers.TravellerJAXBResolver;
7 import java.util.HashSet;
8 import java.util.Set;
9 import javax.ws.rs.core.Application;
10
11 /**
12  * Class used to indicate types that represent root resources.
13  * @author education@oracle.com
14  */
15 public class MyApplication
16     extends Application {
17     public Set<Class<?>> getClasses() {
18         Set<Class<?>> s = new HashSet<Class<?>>();
19         s.add(AirportRM.class);
20         s.add(BetterAirportRM.class);
21         s.add(PlannerRM.class);
22         s.add(MapMessageBodyWriter.class);
23         s.add(DAOExceptionMapper.class);
24         s.add(TravellerJAXBResolver.class);
25         return s;
26     }
27 }
```

Project: examples/jaxrs/ResourceManagers
Path: src/com/example/jaxrs/clients/SimplestClient.java

```
1  /*
2   */
3
4  package com.example.jaxrs.clients;
5
6  import java.io.InputStream;
7  import java.net.URL;
8  import java.util.Scanner;
9
10 /**
11  * Simplest JAX-RS client app
12  * @author education@oracle.com
13  */
14 public class SimplestClient {
15     static public void main( String[] args )
16         throws Exception {
17         String contextURL = "http://localhost:8080/jaxrs";
18         String resourcePath = "/airports";
19         String requestPath = "/numAirports";
20         String urlString =
21             contextURL + resourcePath + requestPath;
22         URL url = new URL( urlString );
23         InputStream result = (InputStream) url.getContent();
24         Scanner scanner = new Scanner( result );
25         System.out.println( "Result:" + scanner.next() );
26     }
27 }
```

Project: examples/jaxrs/ResourceManagers
Path: src/com/example/jaxrs/clients/PathParamClient.java

```
1  /*
2   */
3
4  package com.example.jaxrs.clients;
5
6  import java.io.BufferedReader;
7  import java.io.InputStream;
8  import java.io.InputStreamReader;
9  import java.net.URL;
10
11 /**
12  * JAX-RS client app
13  * @author education@oracle.com
14  */
15 public class PathParamClient {
16     static public void main( String[] args )
17         throws Exception {
18         String contextURL = "http://localhost:8080/jaxrs";
19         String resourcePath = "/airports";
20         String requestPath = "/nameByCode/";
21         String param = "LGA";           // need URL-encoding
22         String urlString =
23             contextURL + resourcePath + requestPath + param;
24         URL url = new URL( urlString );
25         InputStream result = (InputStream) url.getContent();
26         BufferedReader reader =
27             new BufferedReader( new InputStreamReader(result));
28         System.out.println( "Result:" + reader.readLine() );
29     }
30 }
```

Project: examples/jaxrs/ResourceManagers
Path: src/com/example/jaxrs/clients/FormParamClient.java

```

1  /*
2   */
3  package com.example.jaxrs.clients;
4
5  import java.io.BufferedReader;
6  import java.io.InputStream;
7  import java.io.InputStreamReader;
8  import java.io.OutputStream;
9  import java.io.PrintWriter;
10 import java.net.HttpURLConnection;
11 import java.net.URL;
12
13 /**
14  * JAX-RS client app
15  * @author education@oracle.com
16  */
17 public class FormParamClient {
18     static public void main( String[] args )
19         throws Exception {
20         String contextURL = "http://localhost:8080/jaxrs";
21         String resourcePath = "/airports";
22         String requestPath = "/add";
23         String code = "LGA"; // need URL-encoding
24         String name = "LaGuardia"; // need URL-encoding
25         String urlString =
26             contextURL + resourcePath + requestPath;
27         URL url = new URL( urlString );
28         HttpURLConnection connection =
29             (HttpURLConnection) url.openConnection();
30         connection.setRequestMethod( "POST" );
31         connection.setAllowUserInteraction( true );
32         connection.setDoOutput( true );
33         connection.setDoInput( true );
34         connection.connect();
35         OutputStream os = connection.getOutputStream();
36         PrintWriter writer = new PrintWriter( os );
37         writer.print( "code=" + code + "&name=" + name );
38         writer.close();
39         InputStream result = connection.getInputStream();
40         BufferedReader reader =
41             new BufferedReader( new InputStreamReader(result) );
42         System.out.println( "Result:_" + reader.readLine() );
43     }
44 }
```

Project: examples/jaxrs/ResourceManagers
Path: src/com/example/jaxrs/clients/SimplestJerseyClient.java

```
1  /*
2   */
3
4  package com.example.jaxrs.clients;
5
6  import com.sun.jersey.api.client.Client;
7  import com.sun.jersey.api.client.WebResource;
8
9 /**
10 * Simplest Jersey Client
11 * @author education@oracle.com
12 */
13 public class SimplestJerseyClient {
14     static public void main( String[] args ) {
15         String contextURL = "http://localhost:8080/jaxrs";
16         String resourcePath = "/airports";
17         String requestPath = "/numAirports";
18         String urlString =
19             contextURL + resourcePath + requestPath;
20         Client client = Client.create();
21         WebResource resource =
22             client.resource( urlString );
23         String result = resource.get( String.class );
24         System.out.println( "Result:" + result );
25     }
26 }
```

Project: examples/jaxrs/ResourceManagers
Path: src/com/example/jaxrs/clients/QueryParamJerseyClient.java

```
1  /*
2   */
3
4  package com.example.jaxrs.clients;
5
6  import com.sun.jersey.api.client.Client;
7  import com.sun.jersey.api.client.WebResource;
8
9  /**
10  * Simplest Jersey Client
11  * @author education@oracle.com
12  */
13 public class QueryParamJerseyClient {
14     static public void main( String[] args ) {
15         String contextURL = "http://localhost:8080/jaxrs";
16         String resourcePath = "/airports";
17         String requestPath = "/codeByName";
18         String name = "LaGuardia"; // No URL-Encoding!
19         String urlString =
20             contextURL + resourcePath + requestPath;
21         Client client = Client.create();
22         WebResource resource =
23             client.resource( urlString );
24         String result =
25             resource.queryParam("name", name).get(String.class);
26         System.out.println( "Result: " + result );
27     }
28 }
```

Project: examples/jaxrs/ResourceManagers
Path: src/com/example/jaxrs/clients/JSONObjectJerseyClient.java

```
1  /*
2   */
3
4  package com.example.jaxrs.clients;
5
6  import com.example.traveller.model.Airport;
7  import com.sun.jersey.api.client.Client;
8  import com.sun.jersey.api.client.WebResource;
9
10 /**
11  * Simplest Jersey Client
12  * @author education@oracle.com
13  */
14 public class JSONObjectJerseyClient {
15     static public void main( String[] args ) {
16         String urlString =
17             "http://localhost:8080/jaxrs/airports/byCode/LGA";
18         Client client = Client.create();
19         WebResource resource =
20             client.resource(urlString);
21         Airport result =
22             resource
23             .accept( "application/json" )
24             .get( Airport.class );
25         System.out.println( "Result:" + result );
26     }
27 }
```

Project: examples/jaxrs/ResourceManagers
Path: src/com/example/jaxrs/clients/FormSubmitJerseyClient.java

```
1  /*
2   */
3
4  package com.example.jaxrs.clients;
5
6  import com.sun.jersey.api.client.Client;
7  import com.sun.jersey.api.client.WebResource;
8  import com.sun.jersey.core.util.MultivaluedMapImpl;
9  import javax.ws.rs.core.MultivaluedMap;
10
11 /**
12  * Simplest Jersey Client
13  * @author education@oracle.com
14  */
15 public class FormSubmitJerseyClient {
16     static public void main( String[] args ) {
17         String url = "http://localhost:8080/jaxrs/airports/add";
18         Client client = Client.create();
19         WebResource resource = client.resource(url);
20         MultivaluedMap<String, String> params =
21             new MultivaluedMapImpl();
22         params.add( "code", "JFK" );
23         params.add( "name", "John F. Kennedy Airport" );
24         String result =
25             resource
26                 .type( "application/x-www-form-urlencoded" )
27                 .post( String.class, params );
28         System.out.println( "Result: " + result );
29     }
30 }
```

Project: examples/jaxrs/ResourceManagers
Path: src/com/example/jaxrs/clients/ClientResponseJerseyClient.java

```
1  /*
2   */
3
4  package com.example.jaxrs.clients;
5
6  import com.example.traveller.model.Airport;
7  import com.sun.jersey.api.client.Client;
8  import com.sun.jersey.api.client.ClientResponse;
9  import com.sun.jersey.api.client.WebResource;
10
11 /**
12  * Simplest Jersey Client
13  * @author education@oracle.com
14  */
15 public class ClientResponseJerseyClient {
16     static public void main( String[] args ) {
17         String urlString =
18             "http://localhost:8080/jaxrs/airports/byCode/LGA";
19         Client client = Client.create();
20         WebResource resource = client.resource(urlString);
21         ClientResponse response =
22             resource.accept( "application/json" )
23             .get( ClientResponse.class );
24         System.out.println("Code:" + response.getStatus());
25         System.out.println("Result:" + response.getEntity(Airport.class));
26     }
27 }
28 }
```

Project: examples/jaxrs/ResourceManagers
Path: src/com/example/jaxrs/clients/LoggingClient.java

```
1  /*
2   */
3
4  package com.example.jaxrs.clients;
5
6  import com.example.traveller.model.Airport;
7  import com.sun.jersey.api.client.Client;
8  import com.sun.jersey.api.client.WebResource;
9  import com.sun.jersey.api.client.filter.LoggingFilter;
10
11 /**
12  * Simplest Jersey Client
13  * @author education@oracle.com
14  */
15 public class LoggingClient {
16     static public void main( String[] args ) {
17         String urlString =
18             "http://localhost:8080/jaxrs/airports/byCode/LGA";
19         Client client = Client.create();
20         client.addFilter( new LoggingFilter() );
21         WebResource resource = client.resource(urlString);
22         Airport result =
23             resource
24                 .accept( "application/json" )
25                 .get( Airport.class );
26         System.out.println( "Result:" + result );
27     }
28 }
```

Project: examples/jaxrs/JavaEEJAXRS**Path: web/WEB-INF/web.xml**

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app version="3.0"
3   xmlns="http://java.sun.com/xml/ns/javaee"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xsi:schemaLocation=
6     "http://java.sun.com/xml/ns/javaee
7     http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
8   <servlet>
9     <servlet-name>ServletAdaptor</servlet-name>
10    <servlet-class>
11      com.sun.jersey.spi.container.servlet.ServletContainer
12    </servlet-class>
13    <init-param>
14      <param-name>javax.ws.rs.Application</param-name>
15      <param-value>com.example.jaxrs.MyApp</param-value>
16    </init-param>
17  </servlet>
18  <servlet-mapping>
19    <servlet-name>ServletAdaptor</servlet-name>
20    <url-pattern>/resources/*</url-pattern>
21  </servlet-mapping>
22  <security-constraint>
23    <display-name>SecureAirportRM</display-name>
24    <web-resource-collection>
25      <web-resource-name>
26        SecureAirportRM
27      </web-resource-name>
28      <url-pattern>/resources</url-pattern>
29      <http-method>GET</http-method>
30      <http-method>POST</http-method>
31    </web-resource-collection>
32    <auth-constraint>
33      <role-name>administrator</role-name>
34      <role-name>client</role-name>
35    </auth-constraint>
36  </security-constraint>
37  <security-constraint>
38    <display-name>SecureServlet</display-name>
39    <web-resource-collection>
40      <web-resource-name>
41        SecureServlet
42      </web-resource-name>
43      <url-pattern>/SecureServlet</url-pattern>
44      <http-method>GET</http-method>
```

```
45      </web-resource-collection>
46      <auth-constraint>
47          <role-name>administrator</role-name>
48          <role-name>client</role-name>
49      </auth-constraint>
50  </security-constraint>
51  <login-config>
52      <auth-method>BASIC</auth-method>
53      <realm-name>file</realm-name>
54  </login-config>
55  <security-role>
56      <role-name>client</role-name>
57  </security-role>
58  <security-role>
59      <role-name>administrator</role-name>
60  </security-role>
61  <session-config>
62      <session-timeout>
63          30
64      </session-timeout>
65  </session-config>
66 </web-app>
```

Project: examples/jaxrs/JavaEEJAXRS
Path: src/java/com/example/jaxrs/resources/SecureAirportRM.java

```

1  /*
2   */
3  package com.example.jaxrs.resources;
4
5  import com.example.traveller.dao.pojo.AirportDAO;
6  import com.example.traveller.model.Airport;
7  import java.util.List;
8  import javax.annotation.security.PermitAll;
9  import javax.annotation.security.RolesAllowed;
10 import javax.servlet.ServletContext;
11 import javax.ws.rs.FormParam;
12 import javax.ws.rs.GET;
13 import javax.ws.rs.POST;
14 import javax.ws.rs.Path;
15 import javax.ws.rs.PathParam;
16 import javax.ws.rs.Produces;
17 import javax.ws.rs.QueryParam;
18 import javax.ws.rs.core.Context;
19 import javax.ws.rs.core.SecurityContext;
20 import javax.xml.bind.annotation.XmlRootElement;
21
22 /**
23  *
24  * @author education@oracle.com
25  */
26
27 @Path("/secureAirports")
28 @PermitAll
29 public class SecureAirportRM {
30     // ...
31     @GET
32     @Path("/numAirports")
33     public String getNumAirports() {
34         List<String> codes = dao.getAllCodes( null );
35         return
36             (codes == null) ? "0" : "" + codes.size();
37     }
38     // ...
39     @POST
40     @Path("/add")
41     @RolesAllowed("administrator")
42     public String addAirport(
43         @FormParam("code") String code,
44         @FormParam("name") String name ) {

```

```

45     Airport newAirport = null;
46     try {
47         newAirport = dao.add( null, code, name );
48         webContext.log("add:_" +
49                         secContext.getUserPrincipal());
50     }
51     catch( Exception ex ) {}
52     return (newAirport != null) ? "ok" : "fail";
53 }
54 // ...
55 @GET
56 @Path("/nameByCode/{code}")
57 public String getNameByCode(
58     @PathParam("code") String code ) {
59     Airport airport = null;
60     try { airport = dao.findByName( null, code ); }
61     catch( Exception ex ) {
62         System.out.println( "Code:_" + code + "_Ex:_" + ex );
63     }
64     return
65     (airport != null) ? airport.getName() : "(not_found)";
66 }
67 @GET
68 @Path("/codeByName")
69 public String getCodeByName(
70     @QueryParam("name") String name ) {
71     Airport[] airports = dao.findByName( null, name );
72     if ((airports == null) || (airports.length == 0))
73         return "(no_such_airport)";
74     else if (airports.length == 1)
75         return airports[0].getCode();
76     else
77         return "(too_many_candidates)";
78 }
79
80 @GET
81 @Path("/byCode/{code}")
82 @Produces({ "application/xml", "application/json" })
83 public Airport getByCode(
84     @PathParam("code") String code ) {
85     Airport airport = dao.findByName( null, code );
86     return airport;
87 }
88
89 @GET
90 @Path("/xmlByCode/{code}")
91 @Produces({ "application/xml" })

```

```
92  public Airport getXmlByCode(  
93      @PathParam("code") String code ) {  
94      Airport airport = dao.findByCode( null, code );  
95      return airport;  
96  }  
97  
98  @GET  
99  @Path( "/jsonByCode/{code}" )  
100 @Produces ({"application/json"})  
101 public Airport getJsonByCode(  
102     @PathParam("code") String code ) {  
103     Airport airport = dao.findByCode( null, code );  
104     return airport;  
105 }  
106  
107 private AirportDAO dao = new AirportDAO();  
108 @Context SecurityContext secContext;  
109 @Context ServletContext webContext;  
110 }
```

Project: examples/jaxrs/JavaEEJAXRS
Path: web/WEB-INF/sun-web.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <sun-web-app error-url=" " >
3   <context-root>/JavaEEJAXRS</context-root>
4   <security-role-mapping>
5     <role-name>client</role-name>
6     <principal-name>tracy</principal-name>
7   </security-role-mapping>
8   <security-role-mapping>
9     <role-name>administrator</role-name>
10    <principal-name>kelly</principal-name>
11  </security-role-mapping>
12  <class-loader delegate="true"/>
13  <jsp-config>
14    <property name="keepgenerated" value="true"/>
15  </jsp-config>
16 </sun-web-app>
```

Project: examples/jaxrs/JavaEEJAXRS
Path: src/java/com/example/servlets/SecureServlet.java

```

1  /*
2   */
3
4  package com.example.servlets;
5
6  import java.io.IOException;
7  import java.io.PrintWriter;
8  import java.security.Principal;
9  import javax.servlet.ServletException;
10 import javax.servlet.annotation.WebServlet;
11 import javax.servlet.http.HttpServlet;
12 import javax.servlet.http.HttpServletRequest;
13 import javax.servlet.http.HttpServletResponse;
14
15 /**
16 *
17 * @author education@oracle.com
18 */
19 @WebServlet(name="SecureServlet",
20             urlPatterns={"/SecureServlet"})
21 public class SecureServlet extends HttpServlet {
22
23     /**
24      * Processes requests for both HTTP GET and POST methods.
25      * @param request servlet request
26      * @param response servlet response
27      * @throws ServletException if a servlet-specific error occurs
28      * @throws IOException if an I/O error occurs
29     */
30     protected void
31     processRequest(HttpServletRequest request,
32                     HttpServletResponse response)
33     throws ServletException, IOException {
34         response.setContentType("text/html;charset=UTF-8");
35         PrintWriter out = response.getWriter();
36         String user = request.getRemoteUser();
37         Principal principal = request.getUserPrincipal();
38         try {
39             out.println("<html>");
40             out.println("<body>");
41             out.println("<h1>Servlet_SecureServlet_at_"
42                         + request.getContextPath() + "</h1>");
43             out.println("User:_" + user);
44             out.println("Principal:_" + principal);

```

```

45         out.println("</body>");
46         out.println("</html>");
47     } finally {
48         out.close();
49     }
50 }
51 /**
52 * Handles the HTTP <code>GET</code> method.
53 * @param request servlet request
54 * @param response servlet response
55 * @throws ServletException if a servlet-specific error occurs
56 * @throws IOException if an I/O error occurs
57 */
58
59 @Override
60 protected void doGet(HttpServletRequest request,
61                         HttpServletResponse response)
62 throws ServletException, IOException {
63     processRequest(request, response);
64 }
65 /**
66 * Handles the HTTP <code>POST</code> method.
67 * @param request servlet request
68 * @param response servlet response
69 * @throws ServletException if a servlet-specific error occurs
70 * @throws IOException if an I/O error occurs
71 */
72
73 @Override
74 protected void doPost(HttpServletRequest request,
75                         HttpServletResponse response)
76 throws ServletException, IOException {
77     processRequest(request, response);
78 }
79 /**
80 * Returns a short description of the servlet.
81 * @return a String containing servlet description
82 */
83
84 @Override
85 public String getServletInfo() {
86     return "Short_description";
87 } // </editor-fold>
88
89 }

```

5 Project: examples/jaxrs/JavaEEJAXRS

Path: src/java/com/example/jaxrs/resources/AirportRM.java

```

1  /*
2   */
3  package com.example.jaxrs.resources;
4
5  import com.example.traveller.dao.pojo.GenericDAO;
6  import com.example.traveller.dao.pojo.AirportDAO;
7  import com.example.traveller.model.Airport;
8  import com.sun.jersey.spi.resource.Singleton;
9  import java.util.List;
10 import javax.persistence.EntityManager;
11 import javax.persistence.EntityTransaction;
12 import javax.ws.rs.DELETE;
13 import javax.ws.rs.FormParam;
14 import javax.ws.rs.GET;
15 import javax.ws.rs.POST;
16 import javax.ws.rs.Path;
17 import javax.ws.rs.PathParam;
18 import javax.ws.rs.Produces;
19 import javax.ws.rs.QueryParam;
20 import javax.ws.rs.core.MediaType;
21
22 /**
23  *
24  * @author education@oracle.com
25  */
26 @Singleton
27 @Path("/airports")
28 public class AirportRM {
29     // ...
30     @GET
31     @Path("/numAirports")
32     public String getNumAirports() {
33         List<String> codes = dao.getAllCodes( null );
34         return
35             (codes == null) ? "0" : "" + codes.size();
36     }
37     // ...
38     @POST
39     @Path("/add")
40     public String addAirport(
41         @FormParam("code") String code,
42         @FormParam("name") String name ) {
43         Airport newAirport = null;
44         try { newAirport = dao.add( null, code, name ); }
```

```

45     catch( Exception ex ) {}
46     return (newAirport != null) ? "ok" : "fail";
47 }
48 // ...
49 @GET
50 @Path("/nameByCode/{code}")
51 public String getNameByCode(
52     @PathParam("code") String code ) {
53     Airport airport = null;
54     try { airport = dao.findByName( null, code ); }
55     catch( Exception ex ) {
56         System.out.println( "Code:_" + code + "_Ex:_" + ex );
57     }
58     return
59         (airport != null) ? airport.getName() : "(not_found)";
60     }
61
62 @GET
63 @Path("/codeByName")
64 public String getCodeByName(
65     @QueryParam("name") String name ) {
66     Airport[] airports = dao.findByName( null, name );
67     if ((airports == null) || (airports.length == 0))
68         return "(no_such_airport)";
69     else if (airports.length == 1)
70         return airports[0].getCode();
71     else
72         return "(too_many_candidates)";
73     }
74
75 @GET
76 @Path("/byCode/{code}")
77 @Produces({ "application/xml", "application/json" })
78 public Airport getByCode(
79     @PathParam("code") String code ) {
80     Airport airport = dao.findByName( null, code );
81     return airport;
82 }
83
84 @GET
85 @Path("/xmlByCode/{code}")
86 @Produces(MediaType.APPLICATION_XML)
87 public Airport getXmlByCode(
88     @PathParam("code") String code ) {
89     Airport airport = dao.findByName( null, code );
90     return airport;
91 }

```

```
92
93     @GET
94     @Path ( "/ jsonByCode/{code}" )
95     @Produces ( MediaType.APPLICATION_JSON )
96     public Airport getJsonByCode (
97         @PathParam ("code") String code ) {
98         Airport airport = dao.findByCode( null, code );
99         return airport;
100    }
101
102    @DELETE
103    @Path ( "/ removeByCode/{code}" )
104    public void removeByCode (
105        @PathParam ("code") String code ) {
106        EntityManager em =
107            GenericDAO.getEMF ().createEntityManager ();
108        EntityTransaction tx = em.getTransaction ();
109        tx.begin ();
110        dao.remove (em, dao.findByCode (em, code));
111        tx.commit ();
112        em.close ();
113    }
114
115    private AirportDAO dao = new AirportDAO ();
116    // ...
117 }
```

Oscar Hernando Nivia Aragon (onivia@hotmail.com) has a
non-transferable license to use this Student Guide.

Project: examples/jaxrs/JavaEEJAXRS
Path: src/java/com/example/jaxrs/ejb/AirportRM.java

```

1  /*
2   */
3  package com.example.jaxrs.ejb;
4
5  import com.example.traveller.dao.ejb.AirportDAO;
6  import com.example.traveller.model.Airport;
7  import java.util.List;
8  import javax.ejb.EJB;
9  import javax.ejb.Stateless;
10 import javax.ws.rs.DELETE;
11 import javax.ws.rs.FormParam;
12 import javax.ws.rs.GET;
13 import javax.ws.rs.POST;
14 import javax.ws.rs.Path;
15 import javax.ws.rsPathParam;
16 import javax.ws.rs.Produces;
17 import javax.ws.rsQueryParam;
18 import javax.xml.bind.annotation.XmlRootElement;
19
20 /**
21 *
22 * @author education@oracle.com
23 */
24 @Path("/airportsSSSB")
25 @Stateless
26 public class AirportRM {
27     // ...
28     @GET
29     @Path("/numAirports")
30     public String getNumAirports() {
31         List<String> codes = dao.getAllCodes();
32         return
33             (codes == null) ? "0" : "" + codes.size();
34     }
35     // ...
36     @POST
37     @Path("/add")
38     public String addAirport(
39         @FormParam("code") String code,
40         @FormParam("name") String name) {
41         Airport newAirport = null;
42         try { newAirport = dao.add(code, name); }
43         catch( Exception ex ) {}
44         return (newAirport != null) ? "ok" : "fail";

```

```

45     }
46     // ...
47     @GET
48     @Path ("/nameByCode/{code}")
49     public String getNameByCode(
50         @PathParam("code") String code ) {
51         Airport airport = null;
52         try { airport = dao.findByName(code); }
53         catch( Exception ex ) {
54             System.out.println( "Code:_" + code + "_Ex:_" + ex );
55         }
56         return
57             (airport != null) ? airport.getName() : "(not_found)";
58     }
59
60     @GET
61     @Path ("/codeByName")
62     public String getCodeByName(
63         @QueryParam("name") String name ) {
64         Airport[] airports = dao.findByName(name);
65         if ((airports == null) || (airports.length == 0))
66             return "(no_such_airport)";
67         else if (airports.length == 1)
68             return airports[0].getCode();
69         else
70             return "(too_many_candidates)";
71     }
72
73     @GET
74     @Path ("/byCode/{code}")
75     @Produces ({"application/xml","application/json"})
76     public Airport getByCode(
77         @PathParam("code") String code ) {
78         Airport airport = dao.findByName(code);
79         return airport;
80     }
81
82     @GET
83     @Path ("/xmlByCode/{code}")
84     @Produces ({"application/xml"})
85     public Airport getXmlByCode(
86         @PathParam("code") String code ) {
87         Airport airport = dao.findByName(code);
88         return airport;
89     }
90
91     @GET

```

```
92     @Path("/jsonByCode/{code}")
93     @Produces({"application/json"})
94     public Airport getJsonByCode(
95         @PathParam("code") String code) {
96         Airport airport = dao.findByCode(code);
97         return airport;
98     }
99
100    @DELETE
101    @Path("/removeByCode/{code}")
102    public void removeByCode(
103        @PathParam("code") String code) {
104        dao.remove(dao.findByCode(code));
105    }
106
107    @EJB private AirportDAO dao;
108 }
```

Project: examples/jaxrs/JavaEEJAXRS
Path: src/java/com/example/jaxrs/ejb/SingletonAirportRM.java

```

1  /*
2   */
3  package com.example.jaxrs.ejb;
4
5  import com.example.traveller.dao.ejb.AirportDAO;
6  import com.example.traveller.model.Airport;
7  import java.util.List;
8  import javax.ejb.EJB;
9  import javax.ejb.Lock;
10 import javax.ejb.LockType;
11 import javax.ejb.Singleton;
12 import javax.ws.rs.DELETE;
13 import javax.ws.rs.FormParam;
14 import javax.ws.rs.GET;
15 import javax.ws.rs.POST;
16 import javax.ws.rs.Path;
17 import javax.ws.rs.PathParam;
18 import javax.ws.rs.Produces;
19 import javax.ws.rs.QueryParam;
20 import javax.xml.bind.annotation.XmlRootElement;
21
22 /**
23  *
24  * @author education@oracle.com
25  */
26 @Path("/airportsSingletonEJB")
27 @Singleton
28 @Lock(LockType.READ)
29 public class SingletonAirportRM {
30     // ...
31     @GET
32     @Path("/numAirports")
33     public String getNumAirports() {
34         List<String> codes = dao.getAllCodes();
35         return
36             (codes == null) ? "0" : "" + codes.size();
37     }
38     // ...
39     @POST
40     @Path("/add")
41     @Lock(LockType.WRITE)
42     public String addAirport(
43         @FormParam("code") String code,
44         @FormParam("name") String name ) {

```

```

45     Airport newAirport = null;
46     try { newAirport = dao.add(code, name); }
47     catch( Exception ex ) {}
48     return (newAirport != null) ? "ok" : "fail";
49 }
50 // ...
51 @GET
52 @Path("/nameByCode/{code}")
53 public String getNameByCode(
54     @PathParam("code") String code ) {
55     Airport airport = null;
56     try { airport = dao.findByName(code); }
57     catch( Exception ex ) {
58         System.out.println( "Code:_" + code + "_Ex:_" + ex );
59     }
60     return
61     (airport != null) ? airport.getName() : "(not_found)";
62 }
63
64 @GET
65 @Path("/codeByName")
66 public String getCodeByName(
67     @QueryParam("name") String name) {
68     Airport[] airports = dao.findByName(name);
69     if ((airports == null) || (airports.length == 0))
70         return "(no_such_airport)";
71     else if (airports.length == 1)
72         return airports[0].getCode();
73     else
74         return "(too_many_candidates)";
75 }
76
77 @GET
78 @Path("/byCode/{code}")
79 @Produces({ "application/xml", "application/json" })
80 public Airport getByCode(
81     @PathParam("code") String code ) {
82     Airport airport = dao.findByName(code);
83     return airport;
84 }
85
86 @GET
87 @Path("/xmlByCode/{code}")
88 @Produces({ "application/xml" })
89 public Airport getXmlByCode(
90     @PathParam("code") String code ) {
91     Airport airport = dao.findByName(code);

```

```
92     return airport;
93 }
94
95 @GET
96 @Path (" / jsonByCode / { code } ")
97 @Produces ({ "application/json" })
98 public Airport getJsonByCode (
99     @PathParam ("code") String code ) {
100    Airport airport = dao.findByCode (code);
101    return airport;
102 }
103
104 @DELETE
105 @Path (" / removeByCode / { code } ")
106 public void removeByCode (
107     @PathParam ("code") String code ) {
108    dao.remove (dao.findByCode (code));
109 }
110
111 @EJB private AirportDAO dao;
112 }
```

Project: examples/jaxrs/ResourceManagers
Path: src/com/example/jaxrs/clients/AuthenticatingJerseyClient.java

```
1  /*
2   */
3
4  package com.example.jaxrs.clients;
5
6  import com.sun.jersey.api.client.Client;
7  import com.sun.jersey.api.client.WebResource;
8  import com.sun.jersey.api.client.filter.ClientFilter;
9  import com.sun.jersey.api.client.filter.HTTPBasicAuthFilter;
10
11 /**
12  * Simplest Jersey Client
13  * @author education@oracle.com
14  */
15 public class AuthenticatingJerseyClient {
16     static public void main( String[] args ) {
17         String contextURL = "http://localhost:8080/jaxrs";
18         String resourcePath = "/airports";
19         String requestPath = "/numAirports";
20         String urlString =
21             contextURL + resourcePath + requestPath;
22         Client client = Client.create();
23         ClientFilter authFilter =
24             new HTTPBasicAuthFilter("login", "password");
25         client.addFilter(authFilter);
26         WebResource resource =
27             client.resource( urlString );
28         String result = resource.get( String.class );
29         System.out.println( "Result:" + result );
30     }
31 }
```

Project: examples/jaxrs/ResourceManagers
Path: src/com/example/jaxrs/resources/BetterAirportRM.java

```
1  /*
2   */
3  package com.example.jaxrs.resources;
4
5  import com.example.traveller.dao.pojo.AirportDAO;
6  import com.example.traveller.model.Airport;
7  import com.sun.jersey.api.container.httpserver.HttpServerFactory;
8  import com.sun.net.httpserver.HttpServer;
9  import java.io.IOException;
10 import java.net.URI;
11 import java.util.ArrayList;
12 import java.util.List;
13 import javax.ws.rs.FormParam;
14 import javax.ws.rs.GET;
15 import javax.ws.rs.POST;
16 import javax.ws.rs.Path;
17 import javax.ws.rs.PathParam;
18 import javax.ws.rs.Produces;
19 import javax.ws.rs.QueryParam;
20 import javax.ws.rs.core.Context;
21 import javax.ws.rs.core.MediaType;
22 import javax.ws.rs.core.Response;
23 import javax.ws.rs.core.UriBuilder;
24 import javax.ws.rs.core.UriInfo;
25
26 /**
27 *
28 * @author education@oracle.com
29 */
30 @Path("/betterAirports")
31 public class BetterAirportRM {
32     // ...
33     @GET
34     @Path("/numAirports")
35     public String getNumAirports() {
36         List<String> codes = dao.getAllCodes( null );
37         return
38             (codes == null) ? "0" : "" + codes.size();
39     }
40     // ...
41     @POST
42     @Path("/add")
43     public Response addAirport(
44         @FormParam("code") String code,
```

```

45      @FormParam("name") String name ) {
46      Airport newAirport = dao.add(null, code, name);
47      UriBuilder ub = uriInfo.getBaseUriBuilder();
48      URI uri =
49          ub.path(AirportRM.class, "getByCode").build(code);
50      return Response.created(uri).entity(newAirport).build();
51  }
52  // ...
53  @GET
54  @Path("/nameByCode/{code}")
55  public String getNameByCode(
56      @PathParam("code") String code ) {
57      Airport airport = dao.findByName(null, code);
58      return
59          (airport != null) ? airport.getName() : "(not_found)";
60  }
61
62  @GET
63  @Path("/codeByName")
64  public String getCodeByName(
65      @QueryParam("name") String name ) {
66      Airport[] airports = dao.findByName( null, name );
67      if ((airports == null) || (airports.length == 0))
68          return "(no_such_airport)";
69      else if (airports.length == 1)
70          return airports[0].getCode();
71      else
72          return "(too_many_candidates)";
73  }
74
75  @GET
76  @Path("/byCode/{code}")
77  @Produces({ "application/xml", "application/json" })
78  public Airport getByCode(
79      @PathParam("code") String code ) {
80      return dao.findByName( null, code );
81  }
82
83
84  @GET
85  @Path("/xmlByCode/{code}")
86  @Produces(MediaType.APPLICATION_XML)
87  public Airport getXmlByCode(
88      @PathParam("code") String code ) {
89      Airport airport = dao.findByName( null, code );
90      return airport;
91  }

```

```

92
93     @GET
94     @Path( "/jsonByCode/{code}" )
95     @Produces( MediaType.APPLICATION_JSON )
96     public Airport getJsonByCode(
97         @PathParam("code") String code ) {
98         Airport airport = dao.findByCode( null, code );
99         return airport;
100    }
101
102    @Path( "/list" )
103    @GET
104    @Produces( { "application/xml", "application/json" } )
105    public List<com.example.jaxrs.resources.helpers.Airport>
106    listWithLinks() {
107        UriBuilder ub = uriInfo.getBaseUriBuilder();
108        List<Airport> candidates = dao.list(null);
109        List<com.example.jaxrs.resources.helpers.Airport> result =
110            new ArrayList<com.example.jaxrs.resources.helpers.Airport>();
111        for (Airport a : candidates) {
112            com.example.jaxrs.resources.helpers.Airport proxy =
113                new com.example.jaxrs.resources.helpers.Airport(a, ub);
114            result.add(proxy);
115        }
116        return result;
117    }
118
119    @Path( "/listComplete" )
120    @GET
121    @Produces( { "application/xml" } )
122    public List<Airport>
123    listComplete() {
124        List<Airport> result = dao.list(null);
125        return result;
126    }
127
128    @GET
129    @Produces( { "application/xml", "application/json" } )
130    public List<com.example.jaxrs.resources.helpers.Airport>
131    getDefault() {
132        return listWithLinks();
133    }
134
135    private AirportDAO dao = new AirportDAO();
136    @Context UriInfo uriInfo;
137
138    static public void

```

```
139     main(String[] args) throws IOException {
140         String contextUrl =
141             "http://localhost:8080/jaxrs";
142         if (args.length > 0)
143             contextUrl = args[1];
144         HttpServer server =
145             HttpServerFactory.create(contextUrl);
146         server.start();
147     }
148     // ...
149 }
```

Project: examples/jaxrs/ResourceManagers
Path: src/com/example/jaxrs/resources/helpers/Airport.java

```
1  /*
2   */
3
4  package com.example.jaxrs.resources.helpers;
5
6  import com.example.jaxrs.resources.BetterAirportRM;
7  import java.net.URI;
8  import javax.ws.rs.core.UriBuilder;
9  import javax.xml.bind.annotation.XmlAttribute;
10 import javax.xml.bind.annotation.XmlRootElement;
11 import javax.xml.bind.annotation.XmlTransient;
12 import javax.xml.bind.annotation.XmlValue;
13
14 /**
15  *
16  * @author education@oracle.com
17  */
18 @XmlRootElement(name="airport")
19 public class Airport {
20     @XmlValue public String code;
21     @XmlAttribute public String ref;
22     @XmlTransient public com.example.traveller.model.Airport airport;
23     public Airport() {}
24     public
25         Airport(com.example.traveller.model.Airport a, UriBuilder ub) {
26             code = a.getCode();
27             ref = buildRef(a, ub);
28             airport = a;
29         }
30     private String
31     buildRef(com.example.traveller.model.Airport a, UriBuilder ub) {
32         URI result =
33             ub.path(BetterAirportRM.class, "getByCode").build(a.getCode());
34         return result.toString();
35     }
36 }
```

Project: examples/jaxrs/ResourceManagers
Path: src/com/example/jaxrs/resources/PlannerRM.java

```

1  /*
2   */
3
4  package com.example.jaxrs.resources;
5
6  import com.example.traveller.dao.pojo.AirportDAO;
7  import com.example.traveller.dao.pojo.FlightDAO;
8  import com.example.traveller.model.Airport;
9  import com.example.traveller.model.Flight;
10 import com.sun.jersey.api.container.httpserver.HttpServerFactory;
11 import com.sun.net.httpserver.HttpServer;
12 import java.io.IOException;
13 import java.util.ArrayList;
14 import java.util.HashMap;
15 import java.util.List;
16 import java.util.Map;
17 import javax.ws.rs.GET;
18 import javax.ws.rs.Path;
19 import javax.ws.rs.Produces;
20 import javax.ws.rs.QueryParam;
21 import javax.ws.rs.core.Context;
22 import javax.ws.rs.core.Response;
23 import javax.ws.rs.core.UriBuilder;
24 import javax.ws.rs.core.UriInfo;
25
26 /**
27 *
28 * @author education@oracle.com
29 */
30 @Path("/planner")
31 public class PlannerRM {
32     @Path("/neighborsSummary") @GET
33     @Produces({"application/json"})
34     public List<String>
35     getNeighborsSummary(@QueryParam("code") String code) {
36         Airport[] candidates =
37             airportDAO.findNeighbors(null, code);
38         List<String> result = new ArrayList<String>();
39         for (Airport a : candidates)
40             result.add(a.getCode());
41         if (result.size() == 0) {
42             result.add("LAX");
43             result.add("JFK");
44         }
45     }

```

```

45     return result;
46 }
47
48     @Path("/neighborsWithLinks") @GET
49     @Produces({"application/json"})
50     public List<com.example.jaxrs.resources.helpers.Airport>
51     getNeighborsWithLinks(@QueryParam("code") String code) {
52         UriBuilder ub = uriInfo.getBaseUriBuilder();
53         Airport[] candidates =
54             airportDAO.findNeighbors(null, code);
55         List<com.example.jaxrs.resources.helpers.Airport> result =
56             new ArrayList<com.example.jaxrs.resources.helpers.Airport>();
57         for (Airport a : candidates) {
58             com.example.jaxrs.resources.helpers.Airport proxy =
59                 new com.example.jaxrs.resources.helpers.Airport(a, ub);
60             result.add(proxy);
61         }
62         return result;
63     }
64
65     @Path("routesSummary") @GET
66     @Produces({"application/xml", "application/json"})
67     public Map<String, String>
68     getRoutesFromSummary(@QueryParam("start") String code) {
69         Airport start = airportDAO.findByCode(null, code);
70         Map<Flight, Airport> candidates =
71             flightDAO.findRoutesFrom(null, start);
72         Map<String, String> result = new HashMap<String, String>();
73         for(Flight f : candidates.keySet()) {
74             Airport dest = candidates.get(f);
75             result.put(f.getNumber(), dest.getCode());
76         }
77         if (result.size() == 0) {
78             result.put("AA_001", "LAX");
79             result.put("AA_002", "JFK");
80         }
81         return result;
82     }
83
84     @Path("routesWithLinks") @GET
85     @Produces({"application/xml", "application/json"})
86     public Response
87     getRoutesFromWithLinks(@QueryParam("start") String code) {
88         UriBuilder ub = uriInfo.getBaseUriBuilder();
89         Airport start = airportDAO.findByCode(null, code);
90         Map<Flight, Airport> candidates =
91             flightDAO.findRoutesFrom(null, start);

```

```
92     Map<com.example.jaxrs.resources.helpers.Flight,
93         com.example.jaxrs.resources.helpers.Airport> result =
94     new HashMap<com.example.jaxrs.resources.helpers.Flight,
95             com.example.jaxrs.resources.helpers.Airport>();
96     for(Flight f : candidates.keySet()) {
97         Airport dest = candidates.get(f);
98         com.example.jaxrs.resources.helpers.Flight fh =
99             new com.example.jaxrs.resources.helpers.Flight(f, ub);
100        com.example.jaxrs.resources.helpers.Airport ah =
101            new com.example.jaxrs.resources.helpers.Airport(dest, ub);
102        result.put(fh, ah);
103    }
104    return Response.ok(result).build();
105 }
106
107
108 private AirportDAO airportDAO = new AirportDAO();
109 private FlightDAO flightDAO = new FlightDAO();
110 @Context UriInfo uriInfo;
111
112 static public void
113 main(String[] args) throws IOException {
114     String contextUrl =
115         "http://localhost:8081/jaxrs";
116     if (args.length > 0)
117         contextUrl = args[1];
118     HttpServer server =
119         HttpServerFactory.create( contextUrl );
120     server.start();
121 }
122 }
```

Project: examples/jaxrs/ResourceManagers
Path: src/com/example/jaxrs/resources/providers/MapMessageBodyWriter.java

```

1  /*
2   */
3
4  package com.example.jaxrs.resources.providers;
5
6  import java.io.IOException;
7  import java.io.OutputStream;
8  import java.lang.annotation.Annotation;
9  import java.lang.reflect.ParameterizedType;
10 import java.lang.reflect.Type;
11 import java.util.Map;
12 import javax.ws.rs.Produces;
13 import javax.ws.rs.WebApplicationException;
14 import javax.ws.rs.core.MediaType;
15 import javax.ws.rs.core.MultivaluedMap;
16 import javax.ws.rs.ext.MessageBodyWriter;
17 import javax.ws.rs.ext.Provider;
18
19 /**
20  *
21  * @author HSen@vertrax.com (modified by education@oracle.com)
22  */
23 @Provider
24 @Produces(MediaType.APPLICATION_XML)
25 public class MapMessageBodyWriter
26 implements MessageBodyWriter<Map<String, String>>{
27     ...
28     @Override
29     public long getSize(Map<String, String> m, Class<?> type,
30                         Type genericType, Annotation[] annotations,
31                         MediaType mediaType) {
32         return -1;
33     }
34     @Override
35     public boolean isWriteable(Class<?> type, Type generic,
36                               Annotation[] annotations, MediaType mediaType) {
37         MediaType targetMedia = MediaType.APPLICATION_XML_TYPE;
38         return
39             (generic.equals(targetType()) &&
40             mediaType.isCompatible(targetMedia));
41     }
42     @Override
43     public void writeTo(Map<String, String> m,
44                         Class<?> type, Type genericType,

```

```
45     Annotation[] annotations, MediaType mediaType,
46     MultivaluedMap<String, Object> httpHeaders,
47     OutputStream entityStream)
48   throws IOException, WebApplicationException {
49     StringBuffer sb = new StringBuffer("<map>");
50     for (Map.Entry<String, String> entry : m.entrySet()) {
51       String key = entry.getKey(), val = entry.getValue();
52       sb.append("<entry>");
53       sb.append("<key>").append(key).append("</key>");
54       sb.append("<value>").append(val).append("</value>");
55       sb.append("</entry>");
56     }
57     sb.append("</map>");
58     entityStream.write(sb.toString().getBytes());
59   }
60   private Type targetType() {
61     Type[] types = getClass().getGenericInterfaces();
62     ParameterizedType myIf = (ParameterizedType) types[0];
63     return myIf.getActualTypeArguments()[0];
64   }
65 }
```

Project: examples/jaxrs/ResourceManagers**Path: src/com/example/jaxrs/resources/providers/DAOExceptionMapper.java**

```
1  /*
2   */
3
4  package com.example.jaxrs.resources.providers;
5
6  import javax.persistence.EntityExistsException;
7  import javax.persistence.PersistenceException;
8  import javax.ws.rs.core.Response;
9  import javax.ws.rs.ext.ExceptionMapper;
10 import javax.ws.rs.ext.Provider;
11
12 /**
13 *
14 * @author education@oracle.com
15 */
16 @Provider
17 public class DAOExceptionMapper
18     implements ExceptionMapper<PersistenceException> {
19     public Response toResponse(PersistenceException ex) {
20         if (ex instanceof EntityExistsException) {
21             return Response.notAcceptable(null)
22                 .header("DAO-Message", ex)
23                 .build();
24         } else {
25             return
26                 Response.serverError().header("DAO-Message", ex)
27                 .build();
28         }
29     }
30 }
```

Project: examples/jaxrs/JavaEEJAXRS
Path: src/java/com/example/jaxrs/resources/FlightRM.java

```

1  /*
2   */
3  package com.example.jaxrs.resources;
4
5  import com.example.traveller.dao.ejb.FlightDAO;
6  import com.example.traveller.model.Flight;
7  import com.sun.jersey.api.container.httpserver.HttpServerFactory;
8  import com.sun.jersey.spi.resource.Singleton;
9  import com.sun.net.httpserver.HttpServer;
10 import java.io.IOException;
11 import javax.ws.rs.GET;
12 import javax.ws.rs.Path;
13 import javax.ws.rs.PathParam;
14 import javax.ws.rs.Produces;
15
16 /**
17 *
18 * @author education@oracle.com
19 */
20 @Singleton
21 @Path("/flights")
22 public class FlightRM {
23     @GET @Path("/byNumber/{number}")
24     @Produces({"application/xml", "application/json"})
25     public Flight getByNumber(
26         @PathParam("number") String number) {
27         return dao.findByNumber(number);
28     }
29     @Path("/byNumber/{number}/departs")
30     public AirportResource getDepartsByNumber(
31         @PathParam("number") String number) {
32         Flight flight = dao.findByNumber(number);
33         return new AirportResource(flight.getDeparts());
34     }
35
36     private FlightDAO dao = new FlightDAO();
37
38     static public void
39     main(String[] args) throws IOException {
40         String contextUrl =
41             "http://localhost:8080/jaxrs";
42         if (args.length > 0)
43             contextUrl = args[1];
44         HttpServer server =

```

```
45         HttpServerFactory.create( contextUrl );
46         server.start();
47     }
48     // ...
49 }
```

Project: examples/jaxrs/JavaEEJAXRS
Path: src/java/com/example/jaxrs/resources/AirportResource.java

```

1  /*
2   */
3
4  package com.example.jaxrs.resources;
5
6  import com.example.traveller.model.Airport;
7  import javax.ws.rs.GET;
8  import javax.ws.rs.Path;
9  import javax.ws.rs.Produces;
10 import javax.ws.rs.core.Context;
11 import javax.ws.rs.core.UriBuilder;
12 import javax.ws.rs.core.UriInfo;
13
14 /**
15 *
16 * @author education@oracle.com
17 */
18 public class AirportResource {
19     AirportResource(Airport airport) {
20         this.airport = airport;
21     }
22     @GET @Produces({"application/xml", "application/json"})
23     public
24         com.example.jaxrs.resources.helpers.Airport getDefault() {
25             UriBuilder ub = uriInfo.getBaseUriBuilder();
26             return
27                 new com.example.jaxrs.resources.helpers.Airport(airport, ub);
28         }
29     @GET @Path("/code") public String getCode() {
30         return airport.getCode();
31     }
32     @GET @Path("name") public String getName() {
33         return airport.getName();
34     }
35     private Airport airport;
36     @Context UriInfo uriInfo;
37 }
```

Project: examples/jaxrs/JavaEEJAXRS**Path: src/java/com/example/jaxrs/resources/SingletonSecureAirportRM.java**

```

1  /*
2   */
3  package com.example.jaxrs.resources;
4
5  import com.example.traveller.dao.pojo.AirportDAO;
6  import com.example.traveller.model.Airport;
7  import com.sun.jersey.spi.resource.Singleton;
8  import java.util.List;
9  import javax.annotation.security.PermitAll;
10 import javax.annotation.security.RolesAllowed;
11 import javax.servlet.ServletContext;
12 import javax.ws.rs.FormParam;
13 import javax.ws.rs.GET;
14 import javax.ws.rs.POST;
15 import javax.ws.rs.Path;
16 import javax.ws.rsPathParam;
17 import javax.ws.rs.Produces;
18 import javax.ws.rsQueryParam;
19 import javax.ws.rs.core.Context;
20 import javax.ws.rs.core.SecurityContext;
21
22 /**
23  *
24  * @author education@oracle.com
25  */
26 @Singleton
27 @Path("/singletonSecureAirports")
28 @PermitAll
29 public class SingletonSecureAirportRM {
30     // ...
31     @GET
32     @Path("/numAirports")
33     public String getNumAirports() {
34         List<String> codes = dao.getAllCodes( null );
35         return
36             (codes == null) ? "0" : "" + codes.size();
37     }
38     // ...
39     @POST
40     @Path("/add")
41     @RolesAllowed("administrator")
42     public String addAirport(
43         @FormParam("code") String code,
44         @FormParam("name") String name,

```

```

45             @Context SecurityContext secContext) {
46     Airport newAirport = null;
47     try {
48         newAirport = dao.add( null, code, name );
49         webContext.log("add:"+secContext.getUserPrincipal());
50     }
51     catch( Exception ex ) {}
52     return (newAirport != null) ? "ok" : "fail";
53 }
54 // ...
55 @GET
56 @Path("/nameByCode/{code}")
57 public String getNameByCode(
58     @PathParam("code") String code ) {
59     Airport airport = null;
60     try { airport = dao.findByName( null, code ); }
61     catch( Exception ex ) {
62         System.out.println( "Code:_" + code + "_Ex:_" + ex );
63     }
64     return
65     (airport != null) ? airport.getName() : "(not_found)";
66 }
67 @GET
68 @Path("/codeByName")
69 public String getCodeByName(
70     @QueryParam("name") String name ) {
71     Airport[] airports = dao.findByName( null, name );
72     if ((airports == null) || (airports.length == 0))
73     return "(no_such_airport)";
74     else if (airports.length == 1)
75     return airports[0].getCode();
76     else
77     return "(too_many_candidates)";
78 }
79
80 @GET
81 @Path("/byCode/{code}")
82 @Produces({ "application/xml", "application/json" })
83 public Airport getByCode(
84     @PathParam("code") String code ) {
85     Airport airport = dao.findByName( null, code );
86     return airport;
87 }
88
89 @GET
90 @Path("/xmlByCode/{code}")
91 @Produces({ "application/xml" })

```

```
92  public Airport getXmlByCode(  
93      @PathParam("code") String code ) {  
94      Airport airport = dao.findByCode( null, code );  
95      return airport;  
96  }  
97  
98  @GET  
99  @Path( "/jsonByCode/{code}" )  
100 @Produces ({"application/json"})  
101 public Airport getJsonByCode(  
102     @PathParam("code") String code ) {  
103     Airport airport = dao.findByCode( null, code );  
104     return airport;  
105 }  
106  
107 private AirportDAO dao = new AirportDAO();  
108 @Context ServletContext webContext;  
109 }
```

Project: examples/jaxws/DispatchExamples
Path: src/com/example/jaxws/MessagingAPIClient.java

```

1  /*
2   */
3
4  package com.example.jaxws;
5
6  import javax.xml.bind.JAXBContext;
7  import javax.xml.namespace.QName;
8  import javax.xml.ws.AsyncHandler;
9  import javax.xml.ws.Dispatch;
10 import javax.xml.ws.Response;
11 import javax.xml.ws.Service;
12 import javax.xml.ws.Service.Mode;
13 import javax.xml.ws.soap.SOAPBinding;
14
15 /**
16 *
17 * @author education@oracle.com
18 */
19 public class MessagingAPIClient {
20     public static void main(String[] args) throws Exception {
21         QName serviceName =
22             new QName("urn:examples", "Examples");
23         QName portName = new QName("urn:examples", "Hello");
24         Service service = Service.create(serviceName);
25         String url = "http://127.0.0.1:8081/MessagingAPI";
26         service.addPort(portName, SOAPBinding.SOAP11HTTP_BINDING,
27                         url);
28         Class msgClass = MessagingAPIMessage.class;
29         JAXBContext jaxbCtx =
30             JAXBContext.newInstance(msgClass);
31         Dispatch<Object> port =
32             service.createDispatch(portName, jaxbCtx, Mode.PAYLOAD);
33         MessagingAPIMessage request =
34             new MessagingAPIMessage("sayHello", "Tracy");
35         // synchronous
36         MessagingAPIMessage response =
37             (MessagingAPIMessage) port.invoke(request);
38         System.out.println("Response: " + response.getResult());
39         // asynchronous
40         AsyncHandler<Object> responseHandler =
41             new AsyncHandler<Object>() {
42                 public void handleResponse(Response<Object> resp) {
43                     try {
44                         MessagingAPIMessage result =

```

```
45             (MessagingAPIMessage) resp.get());
46             System.out.println(result.getResult());
47         }
48         catch( Exception e )
49         {
50         }
51     };
52     port.invokeAsync( request, responseHandler );
53 }
54 }
```

Project: examples/jaxws/DispatchExamples
Path: src/com/example/jaxws/MessagingAPIServer.java

```

1  /*
2   */
3
4  package com.example.jaxws;
5
6  import javax.xml.bind.JAXBContext;
7  import javax.xml.bind.JAXBException;
8  import javax.xml.bind.Unmarshaller;
9  import javax.xml.bind.util.JAXBSource;
10 import javax.xml.transform.Source;
11 import javax.xml.ws.Endpoint;
12 import javax.xml.ws.Provider;
13 import javax.xml.ws.Service.Mode;
14 import javax.xml.ws.ServiceMode;
15 import javax.xml.ws.WebServiceException;
16 import javax.xml.ws.WebServiceProvider;
17
18 /**
19 *
20 * @author education@oracle.com
21 */
22 @ServiceMode (Mode.PAYLOAD)
23 @WebServiceProvider (portName="Hello", serviceName="Examples",
24                     targetNamespace="urn:examples")
25 public class MessagingAPIServer implements Provider<Source> {
26     MessagingAPIServer() throws JAXBException {
27         Class msgClass = MessagingAPIMessage.class;
28         JAXBContext jaxbContext = JAXBContext.newInstance( msgClass );
29     }
30     // ...
31     public Source invoke(Source payload) {
32         try {
33             Unmarshaller u = jaxbContext.createUnmarshaller();
34             MessagingAPIMessage message =
35                 (MessagingAPIMessage) u.unmarshal( payload );
36             message.setResult("Hello, " + message.getArgument());
37             return new JAXBSource( jaxbContext, message );
38         }
39         catch( Exception ex )
40         { throw new WebServiceException( ex ); }
41     }
42     private JAXBContext jaxbContext;
43     public static void main(String[] args) throws Exception {
44         String url = "http://127.0.0.1:8081/MessagingAPI";

```

```
45     MessagingAPIServer server = new MessagingAPIServer();
46     Endpoint endpoint = Endpoint.publish( url, server );
47   }
48 }
```

Project: examples/jaxws/JAXWSClients
Path: src/clients/AuthSimpleClient.java

```
1  /*
2   */
3
4  package clients;
5
6  import java.util.Map;
7  import javax.xml.ws.BindingProvider;
8
9  /**
10  *
11  * @author education@oracle.com
12  */
13 public class AuthSimpleClient {
14     public static void main(String[] args) {
15         AirportManagerService service =
16             new AirportManagerService();
17         AirportManager port = service.getAirportManagerPort();
18         Map<String, Object> reqCtx =
19             ((BindingProvider) port).getRequestContext();
20         reqCtx.put(BindingProvider.USERNAME_PROPERTY,
21                     "tracy");
22         reqCtx.put(BindingProvider.PASSWORD_PROPERTY,
23                     "password");
24         java.lang.String code = "LGA";
25         java.lang.String name = "New_York_LaGuardia";
26         long result = port.addAirport(code, name);
27         System.out.println("Result = " + result);
28     }
29 }
```

Project: examples/jaxws/Managers
Path: src/com/example/jaxws/common/AuthenticationHandler.java

```

1  /*
2   */
3
4  package com.example.jaxws.common;
5
6  import java.util.Collections;
7  import java.util.Iterator;
8  import java.util.Set;
9  import javax.xml.namespace.QName;
10 import javax.xml.soap.Name;
11 import javax.xml.soap.SOAPEnvelope;
12 import javax.xml.soap.SOAPException;
13 import javax.xml.soap.SOAPHeader;
14 import javax.xml.soap.SOAPHeaderElement;
15 import javax.xml.soap.SOAPMessage;
16 import javax.xml.ws.handler.MessageContext;
17 import javax.xml.ws.handler.soap.SOAPHandler;
18 import javax.xml.ws.handler.soap.SOAPMessageContext;
19
20 /**
21 *
22 * @author education@oracle.com
23 */
24 public class AuthenticationHandler
25     implements SOAPHandler<SOAPMessageContext> {
26     public boolean
27         handleMessage(SOAPMessageContext smc) {
28         Boolean outboundProperty =
29             (Boolean) smc.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
30         SOAPMessage msg = smc.getMessage();
31         if (outboundProperty) {
32             try { embedAuthenticationData(msg); }
33             catch (Exception ex)
34             {}
35         } else {
36             try { validateAuthenticationData(msg); }
37             catch (Exception ex )
38             {}
39         }
40         return true;
41     }
42     private void
43         embedAuthenticationData(SOAPMessage msg)
44             throws SOAPException {

```

```

45     SOAPEnvelope envelope = msg.getSOAPPart().getEnvelope();
46     Name authName = envelope.createName("auth");
47     Name userName = envelope.createName("user");
48     Name passwordName = envelope.createName("password");
49     SOAPHeader header = msg.getSOAPHeader();
50     SOAPHeaderElement newHeader =
51         header.addHeaderElement(authName);
52     newHeader.addAttribute(userName, "tracy");
53     newHeader.addAttribute(passwordName, "password");
54 }
55 private void
56 validateAuthenticationData(SOAPMessage msg)
57     throws SOAPException {
58     SOAPEnvelope envelope = msg.getSOAPPart().getEnvelope();
59     Name authName = envelope.createName("auth");
60     Name userName = envelope.createName("user");
61     Name passwordName = envelope.createName("password");
62     SOAPHeader header = msg.getSOAPHeader();
63     for ( Iterator authNodes = header.getChildElements(authName);
64             authNodes.hasNext(); ) {
65         SOAPHeaderElement authHeader =
66             (SOAPHeaderElement) authNodes.next();
67         String user = authHeader.getAttributeValue(userName);
68         String password =
69             authHeader.getAttributeValue(passwordName);
70         validate(user, password);
71     }
72 }
73 private void
74 validate(String userName, String password) {
75 }
76 }

77 public Set<QName> getHeaders() {
78     return Collections.EMPTY_SET;
79 }

80 public boolean handleFault(SOAPMessageContext messageContext) {
81     return true;
82 }

83 public void close(MessageContext context) {
84 }

85
86
87
88
89 }

```

