Oracle® Application Server

Advanced Web Services Developer's Guide

10*g* Release 3 (10.1.3)

B25603-01

January 2006

ORACLE®

Oracle Application Server Advanced Web Services Developer's Guide, 10*g* Release 3 (10.1.3)

B25603-01

Copyright © 2006, Oracle. All rights reserved.

Primary Author: Thomas Pfaeffle

Contributing Author: Anirban Chattergee, Simeon M. Greene, Sumit Gupta, Bill Jones, Tim Julien, Sunil Kunisetty, Gigi Lee, Mike Lehmann, Jon Maron, Kevin Minder, Bob Naugle, Eric Rajkovic, Ekkehard Rohwedder, Shih-Chang Chen, Quan Wang

Contributor: Ellen Siegal, editor

The Programs (which include both the software and documentation) contain proprietary information; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. This document is not warranted to be error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose.

If the Programs are delivered to the United States Government or anyone licensing or using the Programs on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software—Restricted Rights (June 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and we disclaim liability for any damages caused by such use of the Programs.

Oracle, JD Edwards, PeopleSoft, and Retek are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

Contents

Preface xi

- Intended Audience xi
- Documentation Accessibility xi
- Related Documents xii
- Conventions xvi

1 Ensuring Interoperable Web Services

- **Why is Interoperability Necessary?** 1-1
- **Web Service Interoperability Organizations** 1-2
- **General Guidelines for Creating Interoperable Web Services** 1-3
- **Common Tips for Diagnosing and Solving Interoperability Issues** 1-4
  - Invalid or Improperly Formatted WSDL 1-6
  - WSDLs Containing Proprietary Data Binding Extensions 1-7
  - Illegal XML Characters 1-9
  - Out of Sync SOAPAction Values 1-10
    - Understanding the soapAction WSDL Attribute 1-11
      - Controlling the Value of soapAction in OracleAS Web Services 1-11
      - Controlling the Value of soapAction on the .NET Platform 1-12
  - Null Values in SOAP Messages 1-13
  - Unsigned Schema Numeric Types 1-15
  - Loss of Precision 1-17
- **Tool Support for Interoperability** 1-18
  - Capturing the Web Service Contract 1-18
  - Replaying the Message Payload 1-19
  - Analyzing the Interaction 1-20
  - Obtaining WS-I Tools 1-21
- **Limitations** 1-21
- **Additional Information** 1-21

2 Working with Message Attachments

- **Working with MIME Attachments** 2-1
  - Assembling a Web Service Using swaRef MIME Attachments 2-2
    - Assembling a Web Service Top Down 2-2
      - Constructing a WSDL with swaRef Attachments 2-4
      - Implementing a Service Endpoint Interface with Attachments 2-5

Creating a New Instance of AttachmentPart 2-6
Assembling a Web Service Bottom Up 2-7
Writing a Service Endpoint Interface that Handles Attachments 2-7
Assembling a WSDL File with swaRef Attachment References 2-8
Assembling a Web Service Using SWA MIME Attachments 2-8
Adding SOAP Faults with MIME Attachments 2-11
Specifying SOAP Faults with Attachments in the WSDL 2-13
Implementing a Method that Throws Faults with Attachments 2-13
Using SOAP Faults with Attachments on the Client 2-14
Working with Streaming Attachments 2-15
Assembling Streaming Attachments into a Web Service 2-16
Assembling a Web Service that Supports Streaming Attachments Bottom Up 2-16
Writing an Interface for Steaming Attachments 2-17
Implementing a Service Interface that Uses Streaming Attachments 2-17
WSDL Elements for a Service with Streaming Attachments 2-17
Writing Stub Code to Handle Streaming Attachments 2-18
Assembling a Web Service that Supports Streaming Attachments Top Down 2-19
WSDL Extensions for Streaming Attachments 2-19
Understanding the Streaming Attachments API 2-19
Interface for Attachments 2-20
Interface for Incoming Attachments 2-20
Interface for Outgoing Attachments 2-20
Interface for Attachment Objects 2-21
Factory Class for Attachment Objects 2-21
Working with DIME Attachments 2-21
Creating Interoperable DIME-Encoded Messages 2-21
Implementing Oracle-Proprietary DIME Encoding 2-23
Working with Attachments in WSIF 2-23
Limitations 2-23
Additional Information 2-23

3 Managing Web Services

Understanding Web Service Management 3-1
Web Services Management Environment 3-2
Web Service Management Life Cycle 3-4
Configuring Server-Side Management Information 3-4
Data Flow for Management Information in a J2SE Client 3-6
Configuring Management Information for a J2SE Client 3-7
Data Flow for Management Information in a J2EE Client 3-8
Configuring Management Information for a J2EE Client 3-8
Dynamic Client-Side Configuration 3-9
Providing Dynamic Configuration for a DII Web Service Client 3-10
Providing Dynamic Configuration for a Dynamic Proxy Web Service Client 3-11
Providing Dynamic Configuration for a Static Proxy Web Service Client 3-11
Providing Dynamic Configuration for a J2EE Web Service Client 3-12
Static Client-Side Configuration 3-12
Providing Static Configuration for a Servlet or JSP Web Service Client 3-14

Providing Static Configuration for an EJB Web Service Client 3-14
Providing Static Configuration for an Application Client Web Service Client 3-15
Application Server Control Support for Web Service Management 3-16
Working with Capability Assertions 3-18
How to Assemble Capability Assertions into a Web Service 3-18
Additional Information 3-19

4 Ensuring Web Services Security

Additional Information 4-3

5 Ensuring Web Service Reliability

Setting Up Reliability 5-2
Providing a Running Database 5-2
Installing SQL Tables for the Client and Server 5-2
Changing the Widths of Database Columns 5-2
Adding Reliable Messaging to a Web Service 5-3
Managing Reliability on the Server 5-4
Server-Side Reliability Configuration Elements 5-4
Port-Level Reliability Elements on the Server 5-5
Operation Level Reliability Elements on the Server 5-6
Capability Assertions and Reliability 5-7
Managing Reliability on the Client 5-8
Client-Side Reliability Configuration Elements 5-9
Port Level Reliability Elements on the Client 5-9
Operation Level Reliability Elements on the Client 5-10
Configuring Client-Side Database Support 5-12
Configuring Database Support for a J2SE Client 5-12
Configuring Database Support for a J2EE Client 5-12
Configuring a Listener for a J2EE Client 5-13
Dynamically Configuring Client-Side Reliability 5-13
Tool Support for Web Services Reliability 5-16
WebServicesAssembler Support for Web Service Reliability 5-16
Assembling Reliability into a Web Service Bottom Up 5-16
Assembling Reliability into a Web Service Top Down 5-17
Assembling Reliability into a J2SE Web Service Client Proxy 5-18
Assembling Reliability into a J2EE Web Service Client 5-19
Application Server Control Support for Web Service Reliability 5-19
JDeveloper Support for Web Service Reliability 5-20
Limitations 5-20
Additional Information 5-20

6 Auditing and Logging Messages

Understanding Auditing 6-1
Auditing and Performance 6-1
Processing Audit Messages 6-2
Auditing Request Messages 6-2

Auditing Response Messages ... 6-2
Auditing Fault Messages ... 6-3
Managing Auditing on the Server ... 6-3
Server-Side Auditing Configuration Elements ... 6-3
Managing Auditing on the Client ... 6-4
Understanding Logging ... 6-5
Logging and Performance ... 6-5
Processing Logging Messages ... 6-5
Logging Request Messages ... 6-6
Logging Response Messages ... 6-6
Logging Fault Messages ... 6-6
Managing Logging on the Server ... 6-6
Server-Side Logging Configuration Elements ... 6-6
Port-Level Logging Elements on the Server ... 6-7
Operation Level Logging Elements on the Server ... 6-7
Tool Support for Web Services Auditing and Logging ... 6-8
WebServicesAssembler Support for Web Service Auditing and Logging ... 6-9
Assembling Auditing and Logging into a Web Service Bottom Up ... 6-9
Assembling Auditing and Logging into a Web Service Top Down ... 6-11
Assembling Auditing into a J2SE Web Service Client Proxy ... 6-11
Assembling Auditing into a J2EE Web Service Client ... 6-12
Application Server Control Support for Auditing and Logging ... 6-14
JDeveloper Support for Auditing and Logging ... 6-14
Limitations ... 6-14
Additional Information ... 6-14

7 Custom Serialization of Java Value Types

The Custom Serialization Framework API ... 7-1
Using Custom Serialization in Web Service Development ... 7-2
Implementing a Custom Serializer ... 7-2
Defining a Custom Java Type Value Class ... 7-3
Developing a Custom Serializer Implementation for a Java Type Value Class ... 7-3
Developing a Service Endpoint Interface that Uses a Java Type Value Class ... 7-5
Creating an oracle-webservices Type Mapping Configuration ... 7-6
Using Custom Types in Client-Side Proxy Code ... 7-7
Using Custom Serialization in Top Down Web Service Development ... 7-7
Prerequisites ... 7-8
How to Use Custom Serialization in Top Down Web Service Development ... 7-8
Using Custom Serialization in Bottom Up Web Service Development ... 7-10
Prerequisites ... 7-10
How to Use Custom Serialization in Bottom Up Web Service Development ... 7-10
Ant Tasks for Generating a Web Service ... 7-11
Using Custom Serialization in Schema-Driven Web Service Development ... 7-12
Prerequisites ... 7-12
Schema-Driven Web Services Assembly with Custom Serialization ... 7-12
Sample Schema Document ... 7-15
Java Custom Type Implementation ... 7-15

Implementing a Serializer with Custom Marshalling Logic 7-16
Defining a Serializer Implementation with Marshalling Logic 7-16
Developing a Service Endpoint Interface and Implementation 7-19
Editing the Generated oracle-webservices Type Mapping Configuration XML File 7-21
Developing a Client for Custom Type Mapping and a Custom Serializer 7-23
How to Use Custom Serializers in Client Code 7-24
Editing the Server Side Custom Type Mapping File 7-24
Generating the Web Service Client Side Proxy 7-25
Writing a Web Service Client with Custom Datatypes 7-25
Limitations 7-26
Additional Information 7-26

8 Using JMS as a Web Service Transport

Understanding JMS as a Transport Mechanism 8-1
Data Flow for JMS Transport 8-2
Setting Up JMS Queues 8-3
Assembling a Web Service Bottom Up that Uses JMS Transport 8-4
WSDL Extensions for JMS Transport 8-5
JMS Address Element 8-5
JMS Property Value Element 8-6
Adding JMS Transport Configuration with Deployment Descriptors 8-8
Assembling a Web Service Top Down that Uses JMS Transport 8-8
Assembling a Proxy that Uses JMS as a Transport 8-9
Writing Client Code to Support JMS Transport 8-11
Writing Client Stub Code for JMS Transport 8-11
Setting the Send Queue Location and Connection Factory Programmatically 8-11
Writing DII Code for JMS Transport 8-12
Limitations 8-12
Additional Information 8-12

9 Using Web Services Invocation Framework

Understanding WSIF Architecture 9-1
Configuring a WSIF Endpoint for Java Classes 9-3
Configuring a WSIF Endpoint for a Single Java Port 9-3
Configuring a Single Java Port with wsifJavaBinding 9-3
Configuring a Single Java Port with wsifJavaPort 9-4
Configuring a WSIF Endpoint for Multiple Java Ports 9-5
WSIF Java Extensions to the WSDL 9-6
Configuring a WSIF Endpoint for EJBs 9-7
Configuring a WSIF Endpoint for a Single EJB Port 9-7
Configuring a Single EJB Port with wsifEjbBinding 9-7
Configuring a Single EJB Port with wsifEjbPort 9-8
Configuring a WSIF Endpoint for Multiple EJB Ports 9-10
WSIF EJB Extensions to the WSDL 9-10
Configuring a WSIF Endpoint for Database Resources 9-11
Configuring a WSIF Endpoint for a Single Database Resource Port 9-12

Configuring a Single Database Resource Port with wsifDbBinding 9-12
Configuring a Single Database Resource Port with wsifDbPort 9-14
Configuring a WSIF Endpoint for Multiple Database Resource Ports 9-15
WSIF SQL Extensions to the WSDL 9-16
Writing a WSIF Client 9-16
Writing a WSIF Client Using a Dynamic Proxy 9-18
Using genInterface to Generate a Service Endpoint Interface 9-20
Accessing the Database from a WSIF Client 9-20
Adding Management Configuration to a WSIF Client 9-21
Adding Message Attachments in WSIF 9-23
Adding Attachments with the WSIF API 9-23
Adding Attachments with the OracleCall API 9-24
Tool Support for WSIF 9-24
Limitations 9-25
Additional Information 9-25

10 Using Web Service Providers

What is a Provider? 10-1
Understanding the Provider API 10-1
Provider Interface 10-2
ProviderConfig Class 10-3
MessageContext Class 10-4
HTTPConstants Class 10-4
Provider Servlet 10-4
Making a Web Service Provider-Aware 10-6
Editing the oracle-webservices.xml Deployment Descriptor 10-6
Provider Elements in oracle-webservices.xml 10-7
Editing the web.xml Deployment Descriptor 10-8
Provider Elements in web.xml 10-9
Registering a Provider-Managed Endpoint 10-9
How to Register a Static Provider-Managed Endpoint 10-9
How to Register a Dynamic Provider-Managed Endpoint 10-9
Packaging Provider Web Application Provider Classes 10-10
Deploying Provider Web Applications 10-11
Testing Provider Web Application Deployment 10-11
Managing Provider Endpoints 10-11
Assembling Clients for Provider Web Service Applications 10-11
Additional Information 10-11

A Understanding the Web Services Management Schema

Levels of Web Service Management A-1
Global Level A-1
Port Level A-2
Operation Level A-3
wsmgmt.xml Listing A-3

B JAX-RPC Mapping File Descriptor

Producing a JAX-RPC Mapping File B-1
Naming Conventions for the JAX-RPC Mapping File B-2
Customizing the WSDL or Service Endpoint Interface Contents B-2
Customization Scenarios B-2
Changing Namespace-to-Java Mappings B-3
Changing the Names of Java or WSDL Artifacts B-3
Generating Code into a Single Package from a WSDL with Multiple Namespaces B-3
Wrapping or Unwrapping Mapping for Document-Literal Operations B-4
Mapping Between SOAP Headers and Java Method Parameters B-5

C Web Service MBeans

Web Services MBean Descriptions C-1
Understanding MBean Components C-2
WebServicePort C-2
WebServiceOperation C-2
WSMServiceConfig C-3
WSMOperationConfig C-3
WSMHandlerGlobalConfig C-3
WSMHandlerServiceConfig C-3
WSMHandlerOperationConfig C-3
Initializing MBeans C-4

D Mapping Java Types to XML and WSDL Types

Mapping Java Types to XML Types D-1
Using Java Null Values in Bottom Up Mapping D-2
Mapping Java Primitive Types to XML Types D-2
OC4J Support for Java Value Types D-3
Representing a Java Value Type as a Schema Type D-3
Mapping Support for Arrays D-3
All Formats D-3
Document-Literal and RPC-Literal Formats D-3
RPC-Encoded Format D-4
Mapping Java Collection Classes to XML Types D-5
Limitations on Using Collection and Map Data Types D-5
Definitions for Oracle-Proprietary Collection Data Types D-6
Support for Java Beans Components D-7

E Troubleshooting

OracleAS Web Services Messages E-1
Assembling Web Services from a WSDL E-4
Assembling Web Services from Java Classes E-4
Assembling Web Services From EJBs E-5
Assembling Web Services with JMS Destinations E-5
Developing Web Services From Database Resources E-5

Assembling Web Services with Annotations E-6
Assembling REST Web Services E-6
Testing Web Service Deployment E-7
Assembling a J2EE Web Service Client E-7
Understanding JAX-RPC Handlers E-7
Processing SOAP Headers E-7
Using WebServicesAssembler E-8
Packaging and Deploying Web Services E-9
Ensuring Interoperable Web Services E-9
Working with Message Attachments E-9
Managing Web Services E-10
Ensuring Web Service Reliability E-10
Auditing and Logging Messages E-10
Custom Serialization of Java Value Types E-10
Using JMS as a Web Service Transport E-11
Using the Web Service Invocation Framework E-11
Using Dynamic Invocation Interface to Invoke Web Services E-12
Basic Calls E-12
Configured Calls E-13
Examples of Web Service Clients that use DII E-14

F Third Party Licenses

Apache F-1
The Apache Software License F-2
Apache SOAP F-6
Apache SOAP License F-6
JSR 110 .... F-9
Jaxen F-9
The Jaxen License F-10
SAXPath F-10
The SAXPath License F-10
W3C DOM F-11
The W3C License F-11

Index

Preface

This book describes topics beyond basic Web service assembly. For example, it describes how to diagnose common interoperability problems, how to enable Web service management features (such as reliability, auditing, and logging), and how to use custom serialization of Java value types.

This book also describes how to employ the Web Service Invocation Framework (WSIF), the Web Service Provider API, message attachments, and management features (reliability, logging, and auditing). It also describes alternative Web service strategies, such as using JMS as a transport mechanism.

This preface contains these topics:

- Intended Audience
- Documentation Accessibility
- Related Documents
- Conventions

Intended Audience

Oracle Application Server Advanced Web Services Developer's Guide is intended for application programmers and system administrators who perform the following tasks:

- Configure software installed on the Oracle Application Server.
- Create programs that implement Web services.
- Create programs that run as Web services Clients.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Related Documents

For more information, see these Oracle resources:

- *Oracle Application Server Web Services Developer's Guide*

  This book describes how to use the WebServicesAssembler tool to assemble Web services from a variety of resources: Java classes, EJBs, database resources, JMS destinations and J2SE 5.0 Annotations. You can also assemble REST-style Web services. The *Developers Guide* also describes how to assemble J2SE and J2EE clients to access these services. This book includes descriptions of the message formats and datatypes supported by OracleAS Web Services.

- *Oracle Application Server Web Services Security Guide*

  This book describes the different security strategies that can be applied to a Web service in Oracle Application Server Web Services. These strategies include username token, X.509 token, SAML token, XML encryption, and XML signature. The book describes the configuration options available for the client and the service, for inbound messages and outbound messages. It also describes how to configure these options for a number of different scenarios.

- *Oracle Containers for J2EE Security Guide*

  This book (not to be confused with the *Oracle Application Server 10g Security Guide*), describes security features and implementations particular to OC4J. This includes information about using JAAS, the Java Authentication and Authorization Service, as well as other Java security technologies.

- *Oracle Containers for J2EE Services Guide*

  This book provides information about standards-based Java services supplied with OC4J, such as JTA, JNDI, JMS, JAAS, and the Oracle Application Server Java Object Cache.

- *Oracle Containers for J2EE Configuration and Administration Guide*

  This book describes how to configure and administer applications for OC4J, including use of the Oracle Enterprise Manager 10g Application Server Control Console, use of standards-compliant MBeans provided with OC4J, and, where appropriate, direct use of OC4J-specific XML configuration files.

- *Oracle Containers for J2EE Deployment Guide*

This book covers information and procedures for deploying an application to an OC4J environment. This includes discussion of the deployment plan editor that comes with Oracle Enterprise Manager 10g.

- *Oracle Containers for J2EE Developer's Guide*

  This book discusses items of general interest to developers writing an application to run on OC4J—issues that are not specific to a particular container such as the servlet, EJB, or JSP container. (An example is class loading.)

From the Oracle Application Server core documentation group:

- *Oracle Application Server Security Guide*
- *Oracle Application Server Administrator's Guide*
- *Oracle Application Server Certificate Authority Administrator's Guide*
- *Oracle Application Server Single Sign-On Administrator's Guide*
- *Oracle Application Server Enterprise Deployment Guide*

For Oracle Web Services Manager:

- *Oracle Web Services Manager User and Administrator Guide*
- *Oracle Web Services Manager Extensibility Guide*
- *Oracle Web Services Manager Installation and Deployment Guide*
- *Oracle Web Services Manager Upgrade Guide*

Printed documentation is available for sale in the Oracle Store at:

http://oraclestore.oracle.com/

Contents of the Oracle Application Server Web Services Developer's Guide

This book is designed to be used with the *Oracle Application Server Web Services Developer's Guide*. The "*Developer's Guide*" describes how to use the WebServicesAssembler tool to assemble Web services from a variety of resources: Java classes, EJBs, database resources, JMS destinations and J2SE 5.0 Annotations.

For your convenience, the contents of the *Oracle Application Server Web Services Developer's Guide* are listed here.

- Chapter 1, "Web Services Overview"
- Chapter 2, "Oracle Application Server Web Services Architecture and Life Cycle"
- Chapter 3, "Getting Started"
- Chapter 4, "Oracle Application Server Web Services Messages"
- Chapter 5, "Assembling a Web Service from a WSDL"
- Chapter 6, "Assembling a Web Service with Java Classes"
- Chapter 7, "Assembling a Web Service with EJBs"
- Chapter 8, "Assembling Web Services with JMS Destinations"
- Chapter 9, "Developing Database Web Services"
- Chapter 10, "Assembling Web Services with Annotations"
- Chapter 11, "Assembling REST Web Services"
- Chapter 12, "Testing Web Service Deployment"

- Chapter 13, "Assembling a J2EE Web Service Client"
- Chapter 14, "Assembling a J2SE Web Service Client"
- Chapter 15, "Understanding JAX-RPC Handlers"
- Chapter 16, "Processing SOAP Headers"
- Chapter 17, "Using WebServicesAssembler"
- Chapter 18, "Packaging and Deploying Web Services"
- Appendix A, "Web Service Client APIs and JARs"
- Appendix B, "Oracle Implementation of the WSDL 1.1 API"
- Appendix C, "Troubleshooting"

Links to Related Specifications

The following sections collate references to documentation that appear in the text of this manual:

- Java Technology Documents
- OC4J-Related Documents
- SOAP-Related Documents
- WSDL-Related Documents
- UDDI-Related Documents
- Encryption-Related Documents

Java Technology Documents

- Java 2 Platform Enterprise Edition (J2EE), version 1.4 API specification:

  `http://java.sun.com/j2ee/1.4/docs/api/`

- XML Schemas for J2EE Deployment Descriptors lists the document formats used by the Java 2 Platform, Enterprise Edition (J2EE) deployment descriptors which are described by J2EE 1.4 and later specifications:

  `http://java.sun.com/xml/ns/j2ee/`

- J2EE client schema provides the XSD for a J2EE Web service client:

  `http://java.sun.com/xml/ns/j2ee/j2ee_web_services_client_1_1.xsd`

- Java API for XML-based RPC (JAX-RPC) to build Web applications and Web services. This functionality incorporates XML-based RPC functionality according to the SOAP 1.1 specification.

  `http://java.sun.com/webservices/jaxrpc/index.jsp`

- Java Servlet 2.4 specification:

  `http://www.jcp.org/aboutJava/communityprocess/final/jsr154/index.html`

OC4J-Related Documents

- A list of approved OC4J schemas, including proprietary deployment descriptors:

  `http://www.oracle.com/technology/oracleas/schema/index.html`

- Oracle implementation of UDDI V2.0 that runs on OC4J:

  http://www.oracle.com/technology/tech/webservices/htdocs/uddi/index.html

- *Oracle Database JPublisher User's Guide*

SOAP-Related Documents

- SOAP 1.1 and 1.2 specifications (main page):

  http://www.w3.org/TR/soap/

- SOAP 1.1 specifications:

  - specification:

    http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

  - SOAP 1.1 message encoding:

    http://schemas.xmlsoap.org/soap/encoding/

  - SOAP 1.1 binding schema:

    http://schemas.xmlsoap.org/wsdl/soap/2003-02-11.xsd

    The SOAP 1.2 binding schema is identical to the SOAP 1.1 binding schema, except that the target namespace is:

    http://schemas.xmlsoap.org/wsdl/soap12/

- SOAP 1.2 specification:

  - SOAP 1.2 Part 1: Primer:

    http://www.w3.org/TR/soap12-part0/

  - SOAP 1.2 Part 1: Messaging Format:

    http://www.w3.org/TR/soap12-part1/

  - SOAP 1.2 Part 2 Recommendation (Adjuncts):

    http://www.w3.org/TR/soap12-part2/

  - SOAP 1.2 message encoding:

    http://www.w3.org/2003/05/soap-encoding/

  - HTTP transport for SOAP 1.2:

    http://www.w3.org/2003/05/soap/bindings/HTTP/

  - SOAP binding schema:

    http://schemas.xmlsoap.org/wsdl/soap/2003-02-11.xsd

  - Definition of the fault code element in the SOAP schema:

    http://schemas.xmlsoap.org/soap/envelope/2004-01-21.xsd

WSDL-Related Documents

Web Services Description Language (WSDL) specifications:

http://www.w3.org/TR/wsdl

UDDI-Related Documents

Universal Description, Discovery and Integration specifications:

http://www.uddi.org/

Encryption-Related Documents

- Key Transport algorithms:
  - RSA-1_5:

    http://www.w3.org/2001/04/xmlenc#rsa-1_5
  - RSA-OAEP-MGF1P:

    http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
- Signature keys:
  - RSA-SHA1:

    http://www.w3.org/2000/09/xmldsig#rsa-sha1
  - RSA-MD5:

    http://www.w3.org/2001/04/xmldsig-more#rsa-md5
  - HMAC-SHA1:

    http://www.w3.org/2000/09/xmldsig#hmac-sha1
  - DSA-SHA1:

    http://www.w3.org/2000/09/xmldsig#dsa-sha1

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1 Ensuring Interoperable Web Services

This chapter contains the following sections.

- Why is Interoperability Necessary?
- Web Service Interoperability Organizations
- General Guidelines for Creating Interoperable Web Services
- Common Tips for Diagnosing and Solving Interoperability Issues
- Tool Support for Interoperability

Why is Interoperability Necessary?

The goal of the Web service architecture is to allow heterogeneous business applications to smoothly work together. The architecture is loosely coupled and based on XML standards. Web services are designed to work with each other by defining Web Service Description Language (WSDL) files as service contracts, regardless of which operating system and development technology are behind them. However, because of the complexity involved in service contracts, standards like WSDL and SOAP leave room for ambiguous interpretations. In addition, vendor-specific enhancements and extensions work against universal interoperability.

Business applications must invoke each other's services. These services are often implemented with different technologies. Interoperability failures tend to increase as Web service complexity increases. If you host a publicly available Web service, you will want to ensure that your clients from all over the world, using vastly different tool kits, can successfully invoke it. Likewise, your business application might need to integrate or interact with another vendor's Web service that was built on top of an existing legacy system and has an unusual interface design.

Interoperability issues can originate from any layer of the protocol stack. On the transport level, both parties involved in exchanging messages must agree on a specific physical transport mechanism. For example, you cannot expect to use JMS transport from a non-Java platform. This is why using basic HTTP protocol increases your chance of interoperability. On the message level, because SOAP allows virtually any type of data encoding to be used, interoperability can become difficult. For example, a standard `ArrayList` on a Java platform will not be automatically translated into a `System.collections.ArrayList` on the .NET platform. Also, interoperability issues arise at the basic WSDL and SOAP level: advanced Web service developers will find many more new challenges when they start implementing quality of service (QOS) features such as security, reliability, and transaction services.

Difficulties in interoperability do exist. However, with a few good guidelines, your Oracle Web service should work seamlessly with other J2EE vendor platforms or non-Java platforms like the Microsoft .NET platform.

Web Service Interoperability Organizations

As interoperability gains more and more importance in the Web service community, a number of organizations have been established to achieve this goal.

- SOAPBuilders Community
- WS-Interoperability

SOAPBuilders Community

SOAPBuilders is a loosely organized forum of developers working toward a test bed for interoperability testing between SOAP implementations. Interoperability is demonstrated by implementing a canonical set of tests that are collectively defined by the participants in the forum.

The tests developed by the SOAPBuilder community are, by and large, based on vendor practices. However, practices shift over time. Clean and well-defined rules organized in a formal manner are needed for Web service vendors, Web service developers, and Web service consumers.

You can find more information on SOAPBuilder tests at the following Web site.

`http://www.whitemesa.net/`

WS-Interoperability

The Web Services Interoperability organization (WS-I) is an open industry organization that creates, promotes, and supports generic protocols for the interoperable exchange of messages between Web services. WS-I profiles are guidelines and recommendations for how the standards should be used. These profiles aim to remove ambiguities by adding constraints to the underlying specifications.

WS-I deliverables are profiles, common or best practices, scenarios, testing software, and testing materials.

You should design your Web service so that it adheres to WS-I basic profiles. WS-I compliant services agree to clear contracts and have a greater chance of interoperability.

For example, a WS-I basic profile-compliant Web service should use the following features.

- Use HTTP or HTTPS as the transport binding. HTTP 1.1 is preferred over HTTP 1.0.
- Use literal style encoding. Do not use SOAP encoding.
- Use stricter fault message syntax. When a `MESSAGE` contains a `soap:Fault` element, its element children must be unqualified.
- Use XML version 1.0.
- The service should not declare array wrapper elements using the convention `ArrayOf`*XXX*.

There are many rules defined in the profiles. For more information about WS-I profiles, see the following Web site.

`http://www.ws-i.org`

Oracle is a member of the WS-I organization and is fully committed to helping our customers to achieve interoperability. The Oracle Application Server Web Services platform allows a high degree of flexibility to help you create interoperable Web services.

- The WebServicesAssembler tool and Oracle JDeveloper 10*g* generate WS-I compliant services when you create a bottom up, document-literal Web service. When receiving SOAP messages, the Oracle implementation is accommodating instead of strict so that your message exchanges with other SOAP stacks are more likely to succeed.
- Oracle JDeveloper 10*g* supports integrated testing of WSDL files and running Web services for WS-I Basic Profile conformance. It delivers an enhanced HTTP Analyzer for monitoring and logging, and provides a built-in analysis and reporting tool to better diagnose interoperability issues.

General Guidelines for Creating Interoperable Web Services

The first general guideline is to create Web services which are WS-I compliant, if possible.

The WS-I profiles, however, do not solve all interoperability problems. Many Web services were implemented before WS-I profiles existed. Also, the legacy systems that you are enabling as a Web service might have placed restrictions on your designs.

Thus, good practice in designing Web services should always be adopted from the beginning of the development process, whenever possible. The following sections describe these guidelines.

- Design your Web service top down
- Design data types using XSD first
- Keep data types simple
- Be careful with null values
- Use a compliance testing tool to validate the WSDL
- Understand the differences between platform native types
- Avoid using RPC-encoded message format

Design your Web service top down

The top down approach enables you to design your Web service from service contracts that are not tied to any platform or language-specific characteristics. Your WSDL is less likely to be affected by existing legacy APIs.

Design data types using XSD first

If possible, use an XSD schema editor to design your data types with schema types. Resist using platform-specific data types such as the .NET `DataSet` data type, Java collections, and so on.

Keep data types simple

Avoid unnecessarily complex schema data types such as `xsd:choice`. Simple types provide the best interoperability and have the added benefit of improved performance.

Be careful with null values

Decide what you want to do with null values. For example, should an array be allowed to be null? Should you use a null string or an empty string? If you are sending a null value across the platform, will it cause exceptions on the receiver side?

Avoid sending null values if possible. If you must use null values in your application, design your schema types to clearly indicate that a null value is allowed.

Use a compliance testing tool to validate the WSDL

If your Web service is designed to be WS-I compliant, use the WS-I monitor tool to log messages and the analyzer tool to validate for conformance. WS-I tools can be downloaded for free from the following Web site.

http://www.ws-i.org/deliverables/workinggroup.aspx?wg=testingtools

Understand the differences between platform native types

Some schema types, such as `xsd:unsignedshort` and `xsd:unsignedint`, do not always have a direct native type mapping. For example, there are no Java platform equivalent unsigned types. Schema numeric types such as `xsd:double`, `xsd:float`, and `xsd:decimal` might have different precisions once mapped to their platform native types. There are also limitations on the `xsd:string` type. The strings must not contain illegal XML characters and the `\r` (carriage return) character will typically be mapped to the `\n` (line feed) character.

Avoid using RPC-encoded message format

By itself, the RPC-encoded message format does not imply that you will not be able to interoperate with other platform and clients. In many cases, there are RPC-encoded Web services which are working today. The reason to move away from RPC-encoded message formats is to avoid some of the corner cases where different interpretations of the underlying specification and implementation choices break interoperability. Some examples of these corner cases include the treatment of sparse arrays, multi-dimensional arrays, custom fault code QNames, un-typed payloads, and so on.

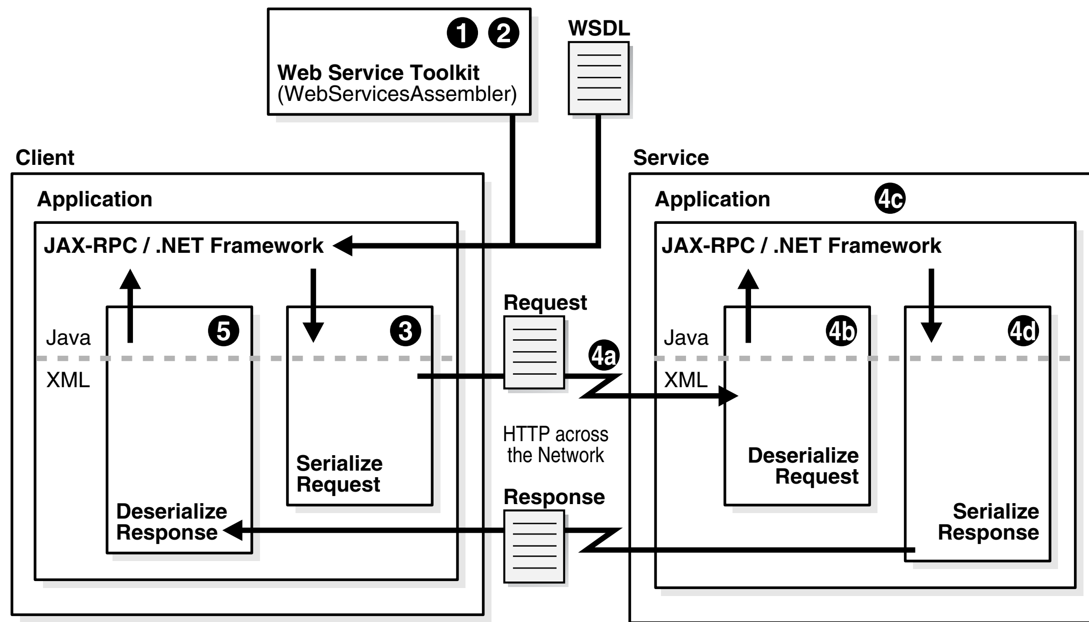
Common Tips for Diagnosing and Solving Interoperability Issues

Many interoperability problems can be recognized and solved by understanding the relationship between the WSDL, the wire format, and the validity of the data representation. Once you can identify where the issue is, correcting it can be straightforward.

Figure 1–1 illustrates the interaction between the elements in a client stack and a service stack. The WSDL is used to generate the client artifacts. The client application, which could be implemented for either the JAX-RPC or .NET platform, forms an XML SOAP request message. The application serializes the request and passes it to the service over HTTP. The service, which could be implemented for either the JAX-RPC or .NET platform, deserializes the request and processes it.

After processing the client request, the service application forms the XML SOAP response message, serializes it, and passes it to the client over HTTP. The client application deserializes the response and processes the result.

Figure 1–1 Points of Possible Interoperability Failures



The numbers and letters in the illustration identify the points, from the client's perspective, where interoperability failures can occur. On the server side, the failure points are similar. The following list describes these problems.

1. The utility which generates the client artifacts fails to consume the WSDL (the contract). The WSDL is either invalid, not compliant with a WS-I Profile, or not supported by the client platform. For example, .NET 1.1 does not support RPC-literal style of Web service. "Invalid or Improperly Formatted WSDL" on page 1-6 provides more information on this type of failure.

2. The utility which generates the client artifacts consumes the WSDL successfully, but the artifacts it generates cannot be used or are not very useful. For example, the WSDL might contain proprietary schema extensions that the tool cannot process. "WSDLs Containing Proprietary Data Binding Extensions" on page 1-7 provides more information on this type of failure.

3. At runtime, an exception is thrown to the client application before any payload is actually processed. Usually, the error will be converted into a SOAP fault with a fault code set as `Client` in SOAP 1.1 or `Sender` in SOAP 1.2. The reason could be that the service endpoint is unavailable or authentication negotiation is not successful. "Illegal XML Characters" on page 1-9 provides more information on this type of failure.

4. A SOAP fault is thrown from the server side due to one of the following reasons.

   a. The request is not routed to the correct operation. The SOAP fault code would be set as `Client` in SOAP 1.1 or `Sender` in SOAP 1.2. "Out of Sync SOAPAction Values" on page 1-10 provides more information on this type of failure.

   b. The request is not deserialized successfully. The SOAP fault code would be set as `Client` in SOAP 1.1 or `Sender` in SOAP 1.2. "Null Values in SOAP Messages" on page 1-13 and "Unsigned Schema Numeric Types" on page 1-15 provide more information on this type of failure.

c. The request is not processed by the application code. The internal exception which is thrown could be an unhandled exception or an application-specific SOAP fault at the business logic level. The SOAP fault code will be set as `Server` or `Receiver` unless the server application considers the error to be related to the validation of input parameters.

d. The response is not serialized. The SOAP fault code will be set as `Server` or `Receiver`.

5. At runtime, an exception is thrown by the client when it tries to deserialize the response to convert the content of the SOAP envelope back into Java instances. "Loss of Precision" on page 1-17 provides more information on this failure.

The following sections provide more information and examples of the failures that can occur at each of these points.

Invalid or Improperly Formatted WSDL

Interoperability failures can be caused by errors that arise in processing the WSDL. This is indicated in Point 1 in Figure 1–1. Different tool kits are not consistent in how they process WSDLs. For example, an improperly formatted WSDL might be graciously accepted by one tool kit but rejected by another.

Example

The following code samples illustrate how an improperly formatted WSDL can cause an interoperability failure.

Example 1–1 illustrates a fragment from an improperly formatted WSDL. The fragment contains an improperly named input parameter, `getQuote`. Notice that the `portType operation` has defined an input parameter named `getQuote`, but the corresponding binding operation does not provide input or output names. If you use the default naming pattern (<*operationName*>`Request` and <*operationName*>`Response`) to create the missing binding operation input and output names, then the WSDL will not be valid because the `getQuote` operation will have inconsistent input parameters.

Example 1–1 WSDL Fragment with an Improperly Named Input Parameter

```
<portType name="qotdPortType">
  <operation name="getQuote">
    <input name="getQuote" message="tns:getQuoteRequest"/>
    <output name="getQuoteResponse" message="tns:getQuoteResponse"/>
  </operation>
</portType>
<binding name="qotdBinding" type="tns:qotdPortType">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="getQuote">
    <soap:operation soapAction="urn:xmethods-qotd#getQuote"/>
    <input>
      <soap:body use="encoded"
                 encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                 namespace="urn:xmethods-qotd"/>
    </input>
    <output>
      <soap:body use="encoded"
                 encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                 namespace="urn:xmethods-qotd"/>
    </output>
  </operation>
```

```
</binding>
```

If you attempt to generate client proxy classes from this WSDL, problems will arise regardless of whether you are on the .NET or the Oracle Application Server Web Services platforms. If you attempt to generate client proxy C# classes on the .NET platform, you will get errors rejecting the WSDL as not having a matching binding. On the Oracle Application Server Web Services platform, Java client proxy classes will be generated, however, you will get warnings about the improper input and output names used in the WSDL.

Unfortunately, the WSDL specification does not clearly specify that the default naming pattern for the `portType` operation should be used for the binding operation when input and output names are missing. It is not clear whether the WSDL should be rejected. The OracleAS Web Services tool kit will graciously accept this WSDL and output warnings.

The potential interoperability problem illustrated in this example can be corrected by specifying the `name` attribute for the input in the binding operation and in the `portType`.

WSDLs Containing Proprietary Data Binding Extensions

Point 2 in Figure 1–1 represents a failure point where the tool kit is unable to generate useful artifacts from the given WSDL. This problem can arise if the WSDL contains proprietary data binding extensions.

Example

The following code samples illustrate how working with a WSDL that contains proprietary data binding extensions can cause interoperability failures.

Note: The following code samples use the ADO.NET `System.Data.DataSet` data type as an example of a proprietary data binding that can cause interoperability failures. However, because XML is extensible by nature, you will likely face this issue when dealing with other proprietary data bindings as well.

Assume that you are developing a Web service for the .NET platform and you have the following C# method that returns a `System.Data.DataSet` data type.

```
public System.Data.DataSet ListBooks (  )
```

Example 1–2 illustrates an XML schema fragment that represents the response output element when the method is exposed as a .NET Web service.

Example 1–2 XML Schema Fragment for a .NET Web Service

```
<s:element name="ListBooksResponse">
   <s:complexType>
    <s:sequence>
      <s:element minOccurs="0" maxOccurs="1" name="ListBooksResult">
        <s:complexType>
        <s:sequence>
          <s:element ref="s:schema"/>
          <s:any/>
        </s:sequence>
        </s:complexType>
       </s:element>
```

```
      </s:sequence>
    </s:complexType>
</s:element>
```

Suppose you want to use the WebServicesAssembler tool to generate client proxy code based on this WSDL. WebServicesAssembler will consume the WSDL successfully and will generate a set of proxy classes including the client code that contains the following Java method.

```
public javax.xml.soap.SOAPElement listBooks(ListBooks parameters) throws
java.rmi.RemoteException
```

The response is returned as a `SOAPElement` because OracleAS Web Services cannot infer any more detailed information from the schema definition. Unfortunately, the schema in the .NET WSDL in Example 1–2 does not fully describe the payload you will receive on the wire representing the .NET `DataSet` data type. The schema describes only that the SOAP message contains two parts: the first part is the schema definition for the payload and the payload follows next.

Example 1–3 illustrates the SOAP message that the client receives. Notice that the first child of the `xs:schema` element (`<xs:element name="NewDataSet"...>`) describes the payload of the message. The second child, (`<diffgr:diffgram ...>`) contains the payload.

Example 1–3 SOAP Message Generated from a .NET DataSet Data Type

```
<soap:Body>
      <ListBooksResponse xmlns="http://francisshanahan.com/">
         <ListBooksResult>
            <xs:schema id="NewDataSet" xmlns=""
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
               <xs:element name="NewDataSet" msdata:IsDataSet="true">
                  <xs:complexType>
                     <xs:choice maxOccurs="unbounded">
                        <xs:element name="bible_content">
                           <xs:complexType>
                              <xs:sequence>
                                 <xs:element name="Book" type="xs:string"
minOccurs="0"/>
                                 <xs:element name="BookTitle" type="xs:string"
minOccurs="0"/>
                              </xs:sequence>
                           </xs:complexType>
                        </xs:element>
                     </xs:choice>
                  </xs:complexType>
               </xs:element>
            </xs:schema>
            <diffgr:diffgram xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1">
               <NewDataSet xmlns="">
                  <bible_content diffgr:id="bible_content1" msdata:rowOrder="0">
                     <Book>01</Book>
                     <BookTitle>The First Book of Moses, called
Genesis</BookTitle>
                  </bible_content>
                  <bible_content diffgr:id="bible_content2" msdata:rowOrder="1">
                     <Book>02</Book>
                     <BookTitle>The Second Book of Moses, Called
```

```
Exodus</BookTitle>
                  </bible_content>
                  …
               </NewDataSet>
            </diffgr:diffgram>
         </ListBooksResult>
      </ListBooksResponse>
   </soap:Body>
</soap:Envelope>
```

Since the WebServicesAssembler tool is unable to provide you with Java type classes that capture the payload schema, how would you write a Java client to process the dataset records appropriately?

The solution is a two step process.

1. Obtain the schema. The code must parse the incoming `SOAPElement`, extract the schema child element and save the schema definition.

   ```
   <xs:schema id="NewDataSet" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
   ```

2. Process the actual payload based on the schema you have.

   You can use tools such as JAXB and Oracle TopLink to process the payload. JAXB provides a Java API and command line tools to help you generate Java type-safe classes directly from the schema.

   Oracle TopLink is an advanced object to relational persistence framework that includes support for JAXB. With TopLink, however, you also get a visual mapping editor and many other advanced features. For more information on TopLink, see the following Web site.

   http://www.oracle.com/technology/products/ias/toplink/index.html

   Once you have generated Java types based on the schema with your preferred tools, you can now process the actual SOAP payload and extract the `any` element that comes after the schema. This element has the tag name `diffgr` and the actual payload starts as the first child of the `<diffgr>` element.

   ```
   <diffgr:diffgram xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
   xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1">
   ```

Illegal XML Characters

Sometimes, even though you have a perfectly valid WSDL that can successfully generate client artifacts, an application might fail before reaching the server side. This is illustrated in Figure 1–1 as Point 3. Usually, this failure occurs because the service is currently unavailable. There are other, harder to detect, cases as well. For example, the application might have produced invalid XML data in the SOAP message. The application could also be attempting to pass characters which are invalid in XML. In these cases, the server transport layer immediately rejects the message before it reaches the Web service.

If your application must send special characters or data in the SOAP `Body` that could cause XML parsing errors, then the application should be designed to use a basic encoding scheme such as `BASE64Encoding`.

Example

The following code samples illustrate how an attempt to pass an invalid XML character in a SOAP message can cause an interoperability failure.

Assume that your C# client application code sends a command string and the service executes the command for the client. Also assume that one of the commands uses the backspace character (`\b`).

```
MyCommand request = new MyCommand("\b");
MyCommandResponse response = soapClient.runCommand(request);
```

If the client application tries to send a command string containing the backspace character (`\b`), OracleAS Web Services will reject it because the backspace character is an invalid XML character. This can return an HTTP transport error and a HTML reply similar to the following.

```
<HTML><HEAD><TITLE>Web Service</TITLE></HEAD><BODY><H1>Bad Request</H1><PRE>Error
parsing envelope: (2, 237) Invalid char in text.</PRE></BODY></HTML>
```

Out of Sync SOAPAction Values

In OracleAS Web Services, the value of the `soapAction` attribute of the WSDL `operation` element is a URI constructed from the target namespace and the operation name. It has the format (`http://`*<target-namespace>*`/`*<operation-name>*). This same value should also be used as the value of the `SOAPAction` HTTP header for this operation in the SOAP request message. If it is not, then unexpected behavior can occur. This type of failure is referred to as Point 4a in Figure 1–1.

Most SOAP processors use the `SOAPAction` HTTP header as a hint to route the request to the appropriate operation without having to decode the full body. Some implementations support the `SOAPAction` value of quoted string (`""`, that is, the empty string). The `SOAPAction` may even be missing from the HTTP header. You should always use a quoted string to avoid interoperability problems.

This problem indicates that the client is out of sync with the WSDL published by the service. To solve this problem, regenerate your client so that it is consistent with the WSDL for the service.

Example

The following code samples illustrate how a mismatch between the values of the `soapAction` attribute of the WSDL `operation` element and the `SOAPAction` HTTP header for this operation can cause an interoperability failure.

Assume that you have the following `add` operation defined in WSDL.

```
<soapbind:operation soapAction= "http://ws.oracle.com/demo/:add"/>
```

Example 1–4 illustrates a SOAP request that is sent at runtime. Note that the message sets a `multiply` `SOAPAction` for an `add` operation.

Example 1–4 SOAP Request Fragment, Illustrating Value of SOAPAction

```
POST http://localhost/khub/MathService.asmx HTTP/1.1

Host: localhost
Proxy-Connection: Keep-Alive
Connection: TE
TE: trailers, deflate, gzip, compress
User-Agent: Oracle HTTPClient Version 10h
SOAPAction: "http://ws.oracle.com/demo/:multiply"
```

```
Accept-Encoding: gzip, x-gzip, compress, x-compress
Content-type: text/xml; charset="UTF-8"
Content-length: 347

<?xml version = '1.0' encoding = 'UTF-8'?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ns0="http://ws.oracle.com/demo/">
   <env:Body>
      <ns0:addElement>
         <ns0:a>3</ns0:a>
         <ns0:b>2</ns0:b>
      </ns0:addElement>
   </env:Body>
</env:Envelope>
```

OracleAS Web Services will throw an error and quit processing because the server detects that there is a mismatch between the `SOAPAction` HTTP header value and the first child element of the SOAP body element (`<ns0:addElement>`). The error will look similar to the following.

```
unexpected element name: expected={http://ws.oracle.com/demo/}addElement,
actual={http://ws.oracle.com/demo/}multiplyElement
```

In a .NET 1.1 implementation, because the `SOAPAction` header is used by default for dispatching SOAP messages, the `multiply` operation will be invoked by mistake instead of the `add` operation. A fault will not be generated. Example 1–5 illustrates the incorrect response that .NET returns from the `multiply` service. It should return a fault because the wrong Web service was invoked. Another possible behavior is to ignore the invalid value of the `SOAPAction`, and route the message to the `add` operation, based on the payload of the request.

Example 1–5 Incorrect Response Returned by .NET

```
<multiplyResponseElement xmlns="http://ws.oracle.com/demo/">
         <MultiplyResult>0</MultiplyResult>
      </multiplyResponseElement>
```

The Apache Axis platform does not rely on the `SOAPAction` header to route the operation. It will always use the top element of the SOAP `Body` payload to find a matching operation.

Understanding the soapAction WSDL Attribute

To prevent unexpected behaviors from occurring in Web service interactions, it is useful to understand how the value of the `soapAction` attribute in the WSDL is set, and its effect on the value of the `SOAPAction` header in the SOAP message. The following sections describe how this attribute is set and used on the OracleAS Web Services and .NET platforms.

- Controlling the Value of soapAction in OracleAS Web Services
- Controlling the Value of soapAction on the .NET Platform

Controlling the Value of soapAction in OracleAS Web Services When you use the Oracle WebServicesAssembler tool to assemble Web services from Java code, a unique `soapAction` attribute value is generated for each `operation` in the WSDL.

You can control the value of this attribute with the Boolean emptySoapAction argument. If you set the argument to true, then the soapAction attribute for each SOAP binding operation in the generated WSDL is set to an empty string and the SOAPAction header in the SOAP message is set to a quoted ("") string. Setting this argument to true increases the chances of interoperability with tools from other vendors.

The default value of the emptySoapAction argument is false. In this case, the value of the soapAction attribute is the concatenation of the targetNamespace and the operation name. This will also be the value of the SOAPAction header in the SOAP message.

Example

The following code samples illustrate how the value of the emptySoapAction argument can be used to control the value of the soapAction attribute in OracleAS Web Services.

Assume that you have the following TimeService Java interface that defines a getDateTime service.

```
public interface TimeService extends Remote {

    public String getDateTime(String name) throws RemoteException;
}
```

This interface can be used as input to the WebServicesAssembler assemble command or Ant task without the emptySoapAction argument. Since the emptySoapAction argument will not be used in the command or task, it is assumed that its value is false by default. The resulting WSDL will have the soapAction value soapAction="http://timeService/getDateTime". This will also be the value of the SOAPAction header in the SOAP message.

Controlling the Value of soapAction on the .NET Platform On the .NET platform, you can specify the RoutingStyle of the XML Web service as SoapAction. Then, depending on whether the Web service is document or RPC style, set the SoapAction by using the Action parameter of the SoapDocumentMethod or SoapRpcMethod attribute. By default, the value for the Action parameter is http://<*web service namespace*>/*MethodName*, where *MethodName* is the name of the XML Web service method.

Note: Instead of defining the SoapAction parameter to dispatch SOAP messages, you can configure the .NET platform to use the RequestElement parameter to set the request element's name. To do this, set RoutingStyle=SoapServiceRoutingStyle.RequestElement and set [SOAPDocumentMethod(Action="")].

A more detailed discussion of this topic is beyond the scope of this manual. For more information on the RequestElement parameter, see your .NET platform documentation.

Example

The following code sample illustrates how you can use the RoutingStyle parameter to control the value of SoapAction for a .NET Web service.

In Example 1–6, the `RoutingStyle` of the XML Web service is set to `SoapServiceRoutingStyle.SoapAction`. The `SoapDocumentMethod` attribute identifies the service as being of document style, and the `Action` parameter identifies the URI of the Web services as `myAddService` and `myMultiplyService`.

Example 1–6 Setting the Routing Style of a Service on the .NET Platform

```
<%@ WebService Language="C#" Class="MathService" %>

using System;
using System.Web.Services;
using System.Web.Services.Protocols;

[SoapDocumentService(SoapBindingUse.Literal,
                 SoapParameterStyle.Wrapped,
                 RoutingStyle=SoapServiceRoutingStyle.SoapAction)]

public class MathService : WebService {
   [SoapDocumentMethod(Action="http://localhost/myAddService")]
   [WebMethod]
   public float Add(float a, float b)
   {
       return a + b;
   }

   [SoapDocumentMethod(Action="http://localhost/myMultiplyService")]
   [WebMethod]
   public float multiply(float a, float b)
   {
       return a * b;
   }
}
```

Null Values in SOAP Messages

Data type mismatch is by far the most common cause of interoperability failures. These failures usually appear when the server or the client tries to deserialize the SOAP message. This is shown in Figure 1–1 as Point 4b and Point 5.

Data type mismatch failure can occur when null values are sent across different platforms. If possible, you should avoid passing null values. If you must send null values from one Web service application to another, you should understand what the data types are mapped to on the other system and whether the data type can handle the null value correctly.

For example, the `xsd:dateTime` schema type is mapped to `System.DateTime` on the .NET platform. On the Java platform, it is mapped to `java.util.Calendar` or `java.util.Date`. If the `Calendar` or `Date` object is initialized with a null value in a Java program, then a null `xsd:dateTime` is sent in the SOAP message. If a Web service built on the .NET platform receives the SOAP message, then it will not be able to correctly deserialize the message, because the `System.DateTime` type is not nillable.

Example

The following code samples illustrate how passing a null value between the sending and receiving platform can result in an interoperability failure.

An XML element is a null element when its `nil` attribute is set to `true`. The WSDL must account for this by setting the `nillable` attribute of the corresponding element to `true` (the default value of `nillable` is `false`).

In the WSDL in Example 1–7, the `source` element has omitted the `nillable` attribute. Thus, by default, it has a `nillable=false` attribute.

Example 1–7 WSDL Fragment, Where the Source Element is Not Nillable

```
<s:element name="Src2html">
    <s:complexType>
    <s:sequence>
<s:element minOccurs="0" maxOccurs="1" name="login" type="s:string"/>
<s:element minOccurs="0" maxOccurs="1" name="passe" type="s:string"/>
<s:element minOccurs="0" maxOccurs="1" name="lan" type="s:string"/>
<s:element minOccurs="0" maxOccurs="1" name="source" type="s:string"/>
</s:sequence>
</s:complexType>
</s:element>
```

Suppose the client application assigns a null value to the Java object representing the source element. Assuming that the Apache Axis platform generates the client proxy code from this WSDL, Example 1–8 illustrates the resulting request SOAP message on the wire. Note that the `source` element has the `xsi:nil` attribute set to `true`.

Example 1–8 Request Message, With xsi:nil Set to true

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
  <Src2html xmlns="http://www.dotnetisp.com/webservices/dotnetisp/src2html.asmx">
   <login>Gigi</login>
   <passe>oracle</passe>
   <lan>C#</lan>
   <source xsi:nil="true"/>
  </Src2html>
 </soapenv:Body>
</soapenv:Envelope>
```

Because the WSDL definition does not allow the `source` element to contain a `nil` value, the Web service built on the OracleAS Web Services platform will fail to deserialize the incoming SOAP message. Example 1–9 illustrates a sample SOAP fault response that OracleAS Web Services generates for the preceding request.

Example 1–9 Sample SOAP Fault Response from OracleAS Web Services

```
<env:Body>
<env:Fault>
    <faultcode>env:Client</faultcode>
    <faultstring>caught exception while handling request: unexpected
null</faultstring>
</env:Fault>
</env:Body>
</env:Envelope>
```

On the other hand, if the same client application assigning a null `source` element is developed with OracleAS Web Services, it will omit the `source` element completely, because the schema defines it to be optional. Example 1–10 illustrates a request SOAP message on the wire sent by a client developed with OracleAS Web Services.

Example 1–10 Request Message, Omitting Optional Elements

```
<env:Envelope xmlns:env=http://schemas.xmlsoap.org/soap/envelope/
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xmlns:ns0="http://www.dotnetisp.com/webservices/dotnetisp/src2html.asmx">
<env:Body>
<ns0:Src2html>
    <ns0:login>Gigi</login>
    <ns0:passe>oracle</passe>
    <ns0:lan>C#</lan>
</ns0:Src2html>
</env:Body>
</env:Envelope>
```

Because the request in Example 1–10 does not contain an invalid null element, the server returns a response without error. Example 1–11 illustrates a sample response to the request.

Example 1–11 Sample SOAP Message

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xmlns:ns0="http://www.dotnetisp.com/webservices/dotnetisp/src2html.asmx">
<env:Body>
<ns0:Src2htmlResponse/>
</env:Body>
</env:Envelope>
```

Unsigned Schema Numeric Types

Another type of deserialization failure that occurs at Point 4b or Point 5 in Figure 1–1 involves using unsigned schema types. On the .NET platform, unsigned types are directly mapped to unsigned native types, while on the Java platform, unsigned types are not defined. As a result, using unsigned schema types in your WSDL can cause unexpected interoperability failures.

If possible, avoid using unsigned numeric data types, such as `unsignedShort`, `unsignedInt`, `double`, or `float` in your client applications. If you do use these data types, you must check their range and precision limit on each system.

Example

The following code samples illustrate how using unsigned data types in a client application can cause unexpected interoperability failures.

The following C# Web method takes an unsigned input argument, `ui`, and returns the same value back to the caller.

```
[WebMethod]
Public uint getUint(uint ui) {
    Return ui
}
```

Example 1–12 illustrates the WSDL fragment containing the input and output parameters for the `getUnit` Web method. These parameters are directly mapped to the `xsd:unsignedInt` type.

Example 1–12 WSDL Fragment, with Mappings for Input and Output Parameters

```
...
<s:element name="getUint">
    <s:complexType>
<s:sequence>
         <s:element minOccurs="1" maxOccurs="1" name="ui" type="s:unsignedInt" />
       </s:sequence>
    </s:complexType>
 </s:element> <s:element name="getUintResponse">
    <s:complexType>
        <s:sequence>
         <s:element minOccurs="1" maxOccurs="1" name="getUintResult"
type="s:unsignedInt" />
        </s:sequence>
    </s:complexType>
  </s:element>
...
```

Using the WebServicesAssembler tool, you can generate JAX-RPC-compliant client code that consumes the `getUnit` service. The generated client code maps the `request` input type to the Java native type `long`.

Now, assume that your Java client application, similar to the code fragment illustrated in Example 1–13, passes in a very large `long` value without knowing the type expected on the other end of the Web service.

Example 1–13 Client Application Fragment that Passes a Large long Value

```
...
stubs.Service1SoapClient myPort = new stubs.Service1SoapClient();
System.out.println("calling " + myPort.getEndpoint());
long l1 = 9223372036854775807L;
GetUint request = new GetUint(l1);
GetUintResponse response = myPort.getUint(request);
long l2 = response.getGetUintResult();
...
```

When the message is sent across the wire, the .NET platform will reject the value and return a SOAP fault similar to the one illustrated in Example 1–14.

Example 1–14 Sample SOAP Fault Sent by the .NET Platform

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Client</faultcode>
      <faultstring>Server was unable to read request. --&gt; There is an error in
XML document (2, 261). --&gt; Value was either too large or too small for a
UInt32.</faultstring>
      <detail />
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

Loss of Precision

Loss of precision can occur at the point where messages are deserialized. This is depicted as Point 4b and Point 5 in Figure 1–1. Failures of this kind are not always obvious, because SOAP faults are not always thrown.

Loss of precision can occur when translating values of XML data types to different platforms. For example, the value of an XML `dateTime` simple type might be different on the Java and .NET platforms. This is because these platforms use different precisions when interpreting the value this XML type.

Example

The following code samples illustrate how passing the `xsd:dateTime` XML simple type between Java and .NET Web services can cause a loss of precision, which translates into a potential interoperability issue.

The following C# Web method returns a `System MAX_VALUE` of the `DateTime` data type.

```
[WebMethod]
Pubic DateTime getDateTime() {
    Return DateTime.MaxValue;
}
```

Example 1–15 illustrates a Java client code fragment that accesses the `getDateTime` .NET Web service.

Example 1–15 Java Code Fragment to Access a .NET Web Service

```
...
GetDateTime request = new GetDateTime();
GetDateTimeResponse response = myPort.getDateTime(request);
Calendar cal = response.getGetDateTimeResult();
...
```

Example 1–16 illustrates the SOAP response that the Java client receives on the wire. Note the values given for the date (highlighted in bold).

Example 1–16 SOAP Response Message from a .NET Web Service that Returns a Date

```
<?xml version="1.0" encoding="utf-8" ?>
  <dateTime
xmlns="http://tempuri.org/">9999-12-31T23:59:59.9999999-08:00</dateTime>
```

The Java client deserializes the response into the `java.util.Calendar` class. Example 1–17 illustrates the printout of the contents of the `Calendar` object. Notice that the `dateTime` value is slightly later than the value passed over the wire. This is because the two native `dateTime` types have different precisions. On the .NET platform, four digits are used for the year value and seven digits are used for the milliseconds. On the Java platform, five digits are used for the year value and three digits are for milliseconds. Moreover, the Java Web service maintains only three digits of milliseconds and rounds up the date. The new value, on the receiving side is January 1, 10000.

Example 1–17 Printout of a Java Calendar Object

```
[java] ERA: 1
[java] YEAR: 10000
[java] MONTH: 0
[java] WEEK_OF_YEAR: 1
```

```
[java] WEEK_OF_MONTH: 1
[java] DATE: 1
[java] DAY_OF_MONTH: 1
[java] DAY_OF_YEAR: 1
[java] DAY_OF_WEEK: 7
[java] DAY_OF_WEEK_IN_MONTH: 1
[java] AM_PM: 0
[java] HOUR: 0
[java] HOUR_OF_DAY: 0
[java] MINUTE: 0
[java] SECOND: 0
[java] MILLISECOND: 0
[java] ZONE_OFFSET: -8
[java] DST_OFFSET: 0
```

Tool Support for Interoperability

The previous sections suggest that there are distinct steps which you can follow to investigate Web services interoperability. These steps can be summarized as the Capture, Replay, Analyze process:

- Capturing the Web Service Contract—capture the WSDL, the XSD files, and the data on the wire: the SOAP payload and the HTTP headers.
- Replaying the Message Payload—re-send the message payload to replay the interaction between proxy and service endpoint. You can also edit the payload to replay the interaction in a different context.
- Analyzing the Interaction—analyze the results of the interaction between the service consumer and provider. You need to make sense out of these broken interactions, sometimes by comparing them with a message payload that transmits successfully.

Oracle provides tools that can assist you in each step. Which tools you use will depend on the environment where you are developing the Web service endpoints and proxies. The following sections describe the tools which can assist you in each step.

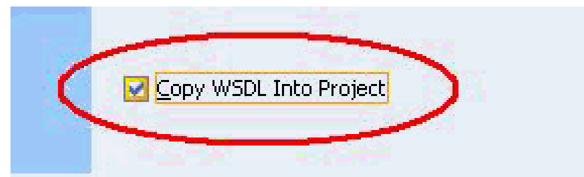
Capturing the Web Service Contract

The Web service contract consists of the WSDL file, the SOAP message payload, and the HTTP headers.

To capture the WSDL and the XSD files, you can use the following tools:

- Oracle JDeveloper wizards provide an option that will make a local copy of the WSDL file in your project. Figure 1–2 illustrates the JDeveloper **Copy WSDL Into Project** option.
- The WebServicesAssembler tool provides a command, `fetchWsdl`, that will copy the base (or top-level) WSDL file and all of its imported and included WSDLs and schemas into a specified output directory. You can run this command on the command line or as an Ant task. See "fetchwsdl" in the *Oracle Application Server Web Services Developer's Guide* for more information on this command.
- The generic Ant task `get` can be used to return a WSDL. If you use this Ant task, you must make a local copy of all the remote resources that have been referenced using `wsdl:import` or `xsd:import`.

Figure 1–2 Oracle JDeveloper Option to Make a Local WSDL Copy



Capturing the SOAP Payload and HTTP Headers

To capture the SOAP payload and HTTP headers, you can use the following tools:

- The HTTP Analyzer tool which is included with Oracle JDeveloper 10*g* offers a convenient way to capture the HTTP traffic between the service consumer and the service provider. This technique assumes that you can easily introduce an HTTP proxy between the two nodes. To change the behavior of the client, you must use the `http.proxyHost` and `http.proxyPort` system properties, while running the client.
- If inserting an HTTP proxy is not practical, you can use an intermediary node, which will redirect the incoming traffic to the final destination. The WS-I monitor is an example of a tool that can do this. Unlike an HTTP proxy, using an intermediary node requires you to change the URL of the endpoint on the client side.
- If you cannot change the client code or route the HTTP traffic through a proxy, set up an TCP sniffer. Ethereal is an example of a network protocol analyzer for Unix and Windows that can be used in this context.

Replaying the Message Payload

Once you have captured the payload, you must be able to either re-send the same payload, or to edit the payload and send it to see the effects of your changes.

- JUnit can help you with the process of re-sending the same payload. You can take advantage of the test classes generated by the WebServicesAssembler `genProxy` command. You can also generate these test classes with Oracle JDeveloper 10g. Figure 1–3 illustrates the option to generate JUnit classes, as it appears in JDeveloper wizards.
- The HTTP Analyzer tool, included with Oracle JDeveloper 10g, offers a convenient way to re-send either the same payload, or the edited payload. Figure 1–4 illustrates the **Resend Request** command in the HTTP Analyzer tool included with JDeveloper.

Figure 1–3 Oracle JDeveloper Option to Generate JUnit Classes

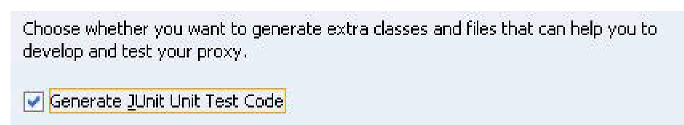
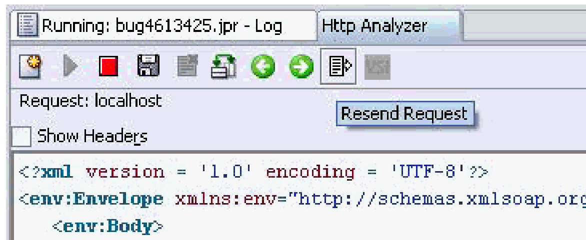


Figure 1–4 HTTP Analyzer Tool's Resend Request Command, Included with JDeveloper



Analyzing the Interaction

Analyze the interaction between the service consumer and the provider. To help you analyze the interaction, you can use the following tools.

- Oracle JDeveloper 10*g* HTTP Analyzer lets you use the WS-I Analyzer directly from the IDE. To use the Analyzer from the IDE, you must first download the Java version of the WS-I tools. The WS-I Analyzer generates the XML configuration files for you, and displays the progress as the reports are generated. Figure 1–5 illustrates the JDeveloper HTTP Analyzer with the focus on the integrated WS-I Analyzer command.

- The WebServicesAssembler tool provides `analyze` command which performs a full scan of the WSDL and confirms whether the WSDL can be processed by this version of the WebServicesAssembler. The `genProxy` and `genInterface` commands perform the same task, but unlike these commands, `analyze` does not generate any code. See "analyze" in the *Oracle Application Server Web Services Developer's Guide* for more information about this command.

  The WSDL validation command is also integrated into the JDeveloper IDE. It is available from the Context menu, based on the type of resource selected. Figure 1–6 illustrates the JDeveloper command to compare files. In this case, the command compares two WSDL files.

Figure 1–5 Oracle JDeveloper with the Integrated WS-I Analyzer Command

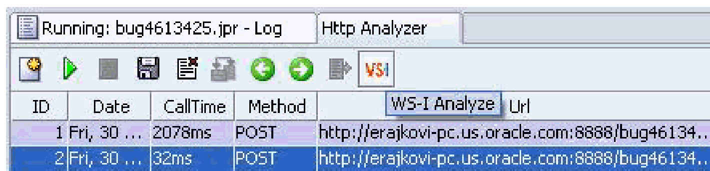
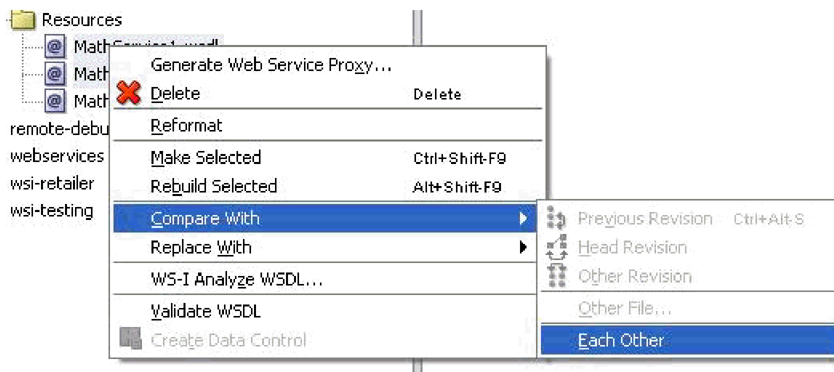


Figure 1–6 Oracle JDeveloper Compare Files Command



Obtaining WS-I Tools

For more information on the tools mentioned in this section, see the following Web addresses.

Using WS-I Test Tools:

http://www.oracle.com/technology/products/jdev/howtos/10g/WS_WSI/WSI_HowTo.html

WS-I Basic Profile 1.0 Compliance Analyzer demo:

http://www.oracle.com/technology/products/jdev/viewlets/905p_j2ee_wsi/wsi_webservices_10g_viewlet_swf.html

Downloading the WS-I tool:

http://www.ws-i.org/deliverables/workinggroup.aspx?wg=testingtools

Limitations

See "Ensuring Interoperable Web Services" on page E-9.

Additional Information

For more information on:

- WebServicesAssembler commands, see "Using WebServicesAssembler", in the *Oracle Application Server Web Services Developer's Guide*

2

Working with Message Attachments

This chapter has the following sections.

- Working with MIME Attachments
- Working with Streaming Attachments
- Working with DIME Attachments
- Understanding the Streaming Attachments API

Working with MIME Attachments

This section contains the following subsections.

- Assembling a Web Service Using swaRef MIME Attachments
- Assembling a Web Service Using SWA MIME Attachments
- Adding SOAP Faults with MIME Attachments

Oracle Application Server Web Services enables you to pass Multipurpose Internet Mail Extension (MIME) attachments with SOAP messages. The SOAP with Attachments (SWA) specification defines the SOAP messages with attachments that are supported by OracleAS Web Services. You can find the SOAP with Attachments specification at this Web site:

http://www.w3.org/TR/SOAP-attachments

In addition to the SOAP With Attachments specification, OracleAS Web Services also supports the Web Service-Interoperability (WS-I) Attachments Profile 1.0 that defines interoperable SOAP messages with attachments. In the interests of interoperability, the WS-I Attachments Profile adds certain constraints to the SWA specification. The profile also defines a new schema type, `swaRef`, for referencing MIME parts from within the SOAP message body. Using this schema type for sending interoperable SOAP messages with attachments is not mandated, nor does it guarantee conformance to the profile. The type is only an optional convenience mechanism. OracleAS Web Services support both the `swaRef` type and plain `SWA` for sending SOAP messages with MIME attachments. You can find the WS-I Attachments Profile 1.0 at this Web site:

http://www.ws-i.org/Profiles/AttachmentsProfile-1.0-2004-08-24.html

OracleAS Web Services supports both the `SWA` and `swaRef` formats and their handling is transparent to the developer.

The following list describes the main differences between `swaRef` and `SWA`.

- `swaRef` attachments are uniquely identified MIME attachments, referenced in the SOAP body by a URL. `SWA` attachments are uniquely identified, but there is no standard way for referencing them from within a SOAP body.
- `SWA` MIME parts must be defined by a `mime:content` element in the corresponding `mime:part`. The following example illustrates a `ClaimPhoto` JPEG file, referenced as a SWA in a `mime:content` element.

```
<mime:part>
     <mime:content part="ClaimPhoto"
         type="image/jpeg"/>
</mime:part>
```

  In contrast, `swaRef` attachments should not be defined in the `mime:content` element. Instead, they should be referenced in the `soapbind:body` of the `wsdl:binding`. The following example illustrates the `ClaimDetail` file referenced as a `swaRef` in the `soapbind:body` element. Note that the file is referenced by a URL.

```
<mime:part>
   <soapbind:body use="literal" parts="ClaimDetail"
             namespace="http://example.com/mimetypes"/>
</mime:part>
```

- `SWA` attachments are mapped to a corresponding Java type based on the MIME content type defined in the WSDL. For example, if the MIME content type is `image/jpeg`, a `java.awt.Image` will be used to represent the attachment. In contrast, `swaRef` will always be mapped to `javax.xml.soap.AttachmentPart`.

Assembling a Web Service Using swaRef MIME Attachments

The `swaRef` type allows MIME attachments to be referenced by a URL within the SOAP body. WS-I publishes a public schema which defines the `swaRef` type. The public schema is defined by the following XSD:

`http://ws-i.org/profiles/basic/1.1/xsd/`

For more information on how to use this type to define attachments in a WSDL, see Section 4.4 of the WS-I Attachments Profile 1.0.

Note: Only RPC-literal and document-literal Web services are supported by the WS-I Attachments Profile 1.0. Thus, only those types of services can use `swaRef`.

OracleAS Web Services support both top down and bottom up development of Web services using MIME attachments with `swaRef`.

- Assembling a Web Service Top Down
- Assembling a Web Service Bottom Up

Assembling a Web Service Top Down

You can use WebServicesAssembler to assemble a Web service top down that can process and generate `swaRef` MIME attachments. One of the ways in which you could do this is to provide a WSDL that contains `swaRef` references as input to the `genInterface` command. This command generates a service endpoint interface containing methods with parameters that will be passed as attachments.

Implementing the generated service endpoint interface includes working with classes and methods in the `oracle.webservices.attachments` package. "Understanding the Streaming Attachments API" on page 2-19 describes this API.

The following general steps describe how to assemble a Web service top down that exposes methods that pass `swaRef` MIME attachments.

1. Provide a WSDL from which you want to assemble a Web service.
2. Enter the import statement for the `swaRef` XSD in the WSDL. The following is an example import statement.

   ```
   <xsd:import namespace="http://ws-i.org/profiles/basic/1.1/xsd"
      schemaLocation="http://ws-i.org/profiles/basic/1.1/xsd"/>
   ```

3. For each part in the `wsdl:message` that needs to be a `swaRef` attachment, set the `type` attribute to the `swaRef` complex type in the imported schema. An example of setting `swaRef` to a complex type is `type="wsi:swaRef"`. "Constructing a WSDL with swaRef Attachments" provides an example of a WSDL that references a `swaRef` attachment.

   Note: For document-literal services you should define an element in the schema with the type `wsi:swaRef` and reference that element in the `wsdl:message` part. This is because the Basic Profile requires document-literal message parts to reference elements, not types.

   In the following example, the `wsdl:part name="status"` in the message part references the `xsd:element name="Status"` of type `swaRef` in the schema section of the WSDL.

   ```
   <!--This is defined in the schema section of the WSDL-->
   <xsd:element name="Status" type="wsi:swaRef" />

   <!--This is the message part that references the swaRef element-->
   <wsdl:message name="replacePhotoResponse">
       <wsdl:part name="status" type="types:Status"/>
     </wsdl:message>
   ```

4. Use the WSDL as input to the WebServicesAssembler `genInterface` command. Following is a sample `genInterface` command.

   ```
   java -jar wsa.jar genInterface -wsdl mySwaRefWsdl -output c:\appDir
   ```

   This command generates a service endpoint interface with methods that handle `swaRef` attachments and a Java class for each `complexType` in the WSDL. The `AttachmentPart` class will be used for all instances of `swaRef`.

   "Implementing a Service Endpoint Interface with Attachments" on page 2-5 illustrates a sample generated service endpoint interface.

5. Implement the generated service endpoint interface.

   In the implementation class, methods that receive attachments will have an `AttachmentPart` parameter while methods that send `swaRef` attachments will have `AttachmentPart` as the return type. As part of the implementation of the methods, create a new instance of the `AttachmentPart` and add an attachment using the `setContent` method. "Creating a New Instance of AttachmentPart" provides an example of creating an instance of `AttachmentPart`.

Example 2–4 illustrates the implementation of a generated service endpoint interface, with code that creates a new instance of `AttachmentPart`.

Constructing a WSDL with swaRef Attachments A WSDL that uses `swaRef` attachments must have the following characteristics.

- The WSDL must import the `swaRef` XSD file.
- Each part of the `wsdl:message` that must be a `swaRef` attachment must set its type attribute to the `swaRef` complex type in the imported schema (or the element attribute set to an element that is of type `swaRef` for document-literal services).
- The `swaRef` attachment should be referenced in the `soapbind:body` of the `wsdl:binding`. Example 2–1 illustrates the format of the `swaRef` reference. The `soapbind:body` element has three attributes. The `use` attribute indicates whether the message format is literal or encoded. The `parts` attribute indicates the part of the SOAP message that the attachment references and the `namespace` attribute indicates the attachment.

Example 2–1 swaRef Attachment Referenced in the soapbind:body Element

```
<mime:part>
   <soapbind:body use="literal|encoded" parts="string" namespace="URL"/>
</mime:part>
```

Example 2–2 illustrates a WSDL that contains a `swaRef` reference that can be used as input to `genInterface`. The `ClaimDetailType` complex type which is referenced by the `ClaimDetail` part of the `ClaimIn` message has an element `ClaimForm` that is of type `swaRef`. The `ClaimDetail` part is referenced in the `soapbind:body` of the binding as opposed to a `mime:content` element which is not referenced as part of the SOAP body. The elements described in the preceding list, as well as the `xsd:import` statement for the `swaRef` schema, are highlighted in bold.

Example 2–2 WSDL that References a swaRef Attachment

```
<?xml version="1.0"?>
<wsdl:definitions xmlns:types="http://example.com/mimetypes"
    xmlns:ref="http://ws-i.org/profiles/basic/1.1/xsd"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    targetNamespace="http://example.com/mimewsdl"
xmlns:tns="http://example.com/mimewsdl">
    <wsdl:types>
        <xsd:schema targetNamespace="http://example.com/mimetypes"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
            <xsd:import namespace="http://ws-i.org/profiles/basic/1.1/xsd"
                        schemaLocation="http://ws-i.org/profiles/basic/1.1/xsd"/>
            <xsd:complexType name="ClaimDetailType">
                <xsd:sequence>
                    <xsd:element name="Name" type="xsd:string"/>
                    <xsd:element name="ClaimForm" type="ref:swaRef"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:schema>
    </wsdl:types>
    <wsdl:message name="ClaimIn">
        <wsdl:part name="ClaimDetail" type="types:ClaimDetailType"/>
    </wsdl:message>
```

```
    <wsdl:message name="ClaimOut">
        <wsdl:part name="ClaimRefNo" type="ref:swaRef"/>
    </wsdl:message>
    <wsdl:portType name="ClaimPortType">
        <wsdl:operation name="SendClaim">
            <wsdl:input message="tns:ClaimIn"/>
            <wsdl:output message="tns:ClaimOut"/>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="ClaimBinding" type="tns:ClaimPortType">
        <soapbind:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="SendClaim">
            <soapbind:operation soapAction="http://example.com/soapaction"/>
            <wsdl:input>
                <mime:multipartRelated>
                    <mime:part>
                        <soapbind:body use="literal"
                                       parts="ClaimDetail"
                                       namespace="http://example.com/mimetypes"/>
                    </mime:part>
                </mime:multipartRelated>
            </wsdl:input>
            <wsdl:output>
                <mime:multipartRelated>
                    <mime:part>
                        <soapbind:body use="literal"
namespace="http://example.com/mimetypes"/>
                    </mime:part>
                </mime:multipartRelated>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
</wsdl:definitions>
```

From this WSDL, the `genInterface` command generates a service endpoint interface with `swaRef` attachments, and a Java class for each `complexType` in the WSDL. The `AttachmentPart` class will be used for all instances of `swaRef`.

For example, WebServicesAssembler will generate a Java class for the `ClaimDetail` complex type. Its `claimForm` parameter, which is of type `swaRef` in the WSDL, will be generated as type `AttachmentPart` in the Java code. The following code sample illustrates part of the class signature.

```
public class ClaimDetail{
   private String name;
   private javax.xml.soap.AttachmentPart claimForm;
   ...
}
```

Implementing a Service Endpoint Interface with Attachments Given a WSDL with references to `swaRef` attachments, WebServicesAssembler generates a service endpoint interface that contains a Java class for each `complexType` in the WSDL. It also contains Java methods that return a `java.xml.soap.AttachmentPart` for each WSDL operation that has a `swaRef` in the output method.

An implementation of a generated service endpoint interface must create a new instance of `AttachmentPart`. OracleAS Web Services uses the `AttachmentPart` class for creating and accessing `swaRef` attachments. Since `AttachmentPart` is an

abstract class, it cannot be instantiated directly. You must create a new instance of `AttachmentPart` as a part of the implementation of methods that send attachments.

Example 2–3 illustrates a generated service endpoint interface and Example 2–4 illustrates its implementation. In the generated service endpoint interface in Example 2–3, the `sendClaim` method sends an attachment as a return parameter.

Example 2–3 Sample Generated Service Endpoint Interface with an Attachment

```
package oracle.j2ee.ws.client.attachments;
public interface claimPortType extends java.rmi.Remote{
   public AttachmentPart sendClaim(ClaimDetail claimDetail) throws
     java.rmi.RemoteException;
}
```

The implementation class in Example 2–4 creates a new instance of `AttachmentPart` in the `sendClaim` method. "Creating a New Instance of AttachmentPart" describes how to use the SAAJ API to create a new instance of `AttachmentPart`.

Once the `AttachmentPart` instance is created, an image or any other object can be added using the `setContent(Object obj, String mimeType)` method. The first parameter of the method takes the object to be attached and the second parameter specifies the MIME content type of that object. For example, if a GIF image is being sent, the `setContent` method would be `setContent(newPhoto,"image/gif")`.

Example 2–4 Implementation of a Service Endpoint Interface

```
public class ClaimPortTypeImpl implements ClaimPortType {

   public AttachmentPart sendClaim(ClaimDetail claimDetail) throws
RemoteException{
   AttachmentPart impl = null;
   try{
   MessageFactory factory = MessageFactory.newInstance();
   SOAPMessage  message = factory.createMessage();
   impl = message.createAttachmentPart();
} catch(SOAPException ex){
   return null;
}
return impl;
}
```

Creating a New Instance of AttachmentPart This section illustrates the basic code for creating a new instance of `AttachmentPart`. OracleAS Web Services uses the `AttachmentPart` class for creating and accessing `swaRef` attachments.

Example 2–5 illustrates sample code that uses the SAAJ API to create a new instance of `AttachmentPart`. Note that the `createAttachmentPart` method of `javax.xml.soap.SOAPMessage` takes an object and a MIME type as parameters.

Example 2–5 Creating a New Instance of AttachmentPart

```
javax.xml.soap.MessageFactory mf = javax.xml.soap.MessageFactory.newInstance();
javax.xml.soap.SOAPMessage message = mf.createMessage();
javax.xml.soap.AttachmentPart ap =
message.createAttachmentPart(attachmentObj,"image/jpeg");
```

The generated service interface can now be implemented and `swaRef` attachments can be sent and received using `AttachmentPart` from the SAAJ API.

Assembling a Web Service Bottom Up

You can use WebServicesAssembler to assemble a Web service bottom up that can pass `swaRef` MIME attachments. One of the ways in which you could do this is to provide a Java class that contains method parameters that represent attachments to the `assemble` or `genWsdl` commands. These commands produce a WSDL with references to `swaRef` attachments.

Writing and implementing the Java classes that contain attachments includes working with classes and methods in the `oracle.webservices.attachments` package. "Understanding the Streaming Attachments API" on page 2-19 describes this API.

Note: WebServicesAssembler will not be able to assemble a Web service that can pass `swaRef` MIME attachments if you specify an RPC-encoded message format. To assemble the service, you must select a different format.

The following general steps describe how to assemble a Web service bottom up that exposes elements in the WSDL that represent `swaRef` MIME attachments.

1. Provide the Java class you want to expose as a Web service.

   In the class file, any methods that use parameters that represent attachments should be identified as type `AttachmentPart`. In your implementation of the methods that use attachments, create a new instance of `AttachmentPart`. "Creating a New Instance of AttachmentPart" illustrates sample code that uses the SAAJ API to create this.

   "Writing a Service Endpoint Interface that Handles Attachments" on page 2-7 illustrates a service endpoint interface with a method that passes attachments.

2. Use the WebServicesAssembler to generate a WSDL file.

   - Use the `assemble` command if you want to assemble the WSDL and all of the service artifacts for a Web service.
   - Use the `genWsdl` command if you want to generate only the WSDL file.

   The generated WSDL file will contain an `<xsd:element name="..." type="ref:swaRef"/>` element for each parameter in the Java class identified as type `AttachmentPart`.

   "Assembling a WSDL File with swaRef Attachment References" illustrates a WSDL file generated with the `genWsdl` command that contains `swaRef` attachment references.

3. If you used the `genWsdl` command to generate the WSDL, you can now inspect the file.

   If you used the `assemble` command to assemble the Web service, you can now deploy the service, generate a proxy, and write a client. "Assembling a Web Service with Java Classes" in the *Oracle Application Server Web Services Developer's Guide* provides more information on these steps.

Writing a Service Endpoint Interface that Handles Attachments If methods in your service endpoint interface pass attachments, and you want the attachments to be passed as a `swaRef`, then the parameters that represent the attachments must be of type `AttachmentPart`. In bottom up Web service assembly scenarios, WebServicesAssembler generates a `swaRef` type reference into the WSDL for every `AttachmentPart` type it encounters in an implementation class or interface.

Example 2–6 illustrates an interface that contains a parameter, `claimDetail`, of type `AttachmentPart`. When passed to WebServicesAssembler using the `genWsdl` or `assemble` command, this interface yields a WSDL file.

Example 2–6 Service Endpoint Interface with a Method that Passes Attachments

```
public interface ClaimServicePortType extends java.rmi.Remote{
   public String sendClaim(int id, AttachmentPart claimDetail) throws
RemoteException;
   }
```

Assembling a WSDL File with swaRef Attachment References Example 2–7 illustrates a fragment of the WSDL file, displaying only the type definitions. The example assumes that the interface in Example 2–6 was used as input to the `genWsdl` command to generate the WSDL with document-wrapped style. The WSDL fragment identifies the `claimDetail` element as being of type `swaRef` (highlighted in bold).

Note that the same WSDL fragment would be created if the `assemble` command was used to generate the WSDL. This is the default behavior when WebServicesAssembler encounters the `AttachmentPart` data type.

Example 2–7 WSDL Fragment, Displaying only Type Definitions

```
      ...
        <xsd:import namespace="http://ws-i.org/profiles/basic/1.1/xsd" />
        <xsd:complexType name="sendClaimType">
            <xsd:sequence>
                <xsd:element name="id" type="xsd:string"/>
                <xsd:element name="claimDetail" type="ref:swaRef"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:schema>
</wsdl:types>
...
```

Assembling a Web Service Using SWA MIME Attachments

SWA attachments refer to attachments that conform to the SOAP With Attachments (SWA) specification. Currently, WebServicesAssembler supports only top down assembly of Web services with WSDL files that contain `SWA` attachments. `SWA` MIME parts are defined in the `<wsdl:binding>` clause of the WSDL. The attachment is defined by a `mime:content` element in the corresponding `mime:part`.

Example 2–8 illustrates the format of the `mime:content` element and its `part` and `type` attributes. The `part` attribute identifies the part of the SOAP message to which the attachment belongs. The `type` attribute identifies the content type in the attachment.

Example 2–8 SWA Attachment Referenced in the mime:content Element

```
<mime:part>
     <mime:content part="string"
         type="content type"/>
</mime:part>
```

WebServicesAssembler processes each `mime:content` element found in the WSDL file and maps it to a Java type that corresponds to the MIME-content type.

Table 2–1 displays the MIME content types and corresponding Java types to which they will be mapped. The default type for any MIME-content type that is not recognized is `javax.activation.DataHandler`.

Table 2–1 Mapping Content Types for SWA Attachments

Content Type	Java Type
image/jpeg, image/gif, image/tif	`java.awt.Image`. If these content types are used in combination, or if `image/*` is specified, then they are mapped to the `java.activation.DataHandler` Java type.
text/plain, application/plain	`java.lang.String`. If these types are used in combination, then they are mapped to the `java.activation.DataHandler` Java type.
text/xml	`java.activation.DataHandler`

1. Provide the WSDL from which you want to generate a Web service.
2. Each element that represents a SWA attachment must appear in a `<mime:content...>` element in the `<wsdl:binding...>` clause of the WSDL.

   In Example 2–9, a `photo` JPEG attachment appears in a `<mime:content...>` element:

```
<mime:part>
    <mime:content part="photo"
                  type="image/jpeg"/>
</mime:part>
```

3. Use the WSDL as input to the WebServicesAssembler `genInterface` command. Following is a sample `genInterface` command.

```
java -jar wsa.jar genInterface -wsdl mySwaWsdl -output c:\appDir
```

   The `genInterface` command generates a service endpoint interface with parameters that represent `SWA` attachments and a Java class for each `complexType` in the WSDL. A Java class will be used for all instances of `SWA` attachments.

   Example 2–10 illustrates a generated service endpoint interface with a parameter that represents a SWA attachment as a `java.awt.Image` Java data type.

4. Implement the generated service endpoint interface.

Example 2–9 illustrates a WSDL that contains a `SWA` reference that can be used as input to `genInterface`. Note that this WSDL example omits the `wsdl:service` element for the sake of brevity.

Example 2–9 WSDL that References a SWA Attachment

```
<wsdl:definitions
  name="PhotoCatalogService"
  targetNamespace="http://examples.com/PhotoCatalog"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:types="http://examples.com/PhotoCatalog/types"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:tns="http://examples.com/PhotoCatalog">
  <wsdl:message name="addPhotoRequest">
    <wsdl:part name="photo" type="xsd:hexBinary"/>
```

```
  </wsdl:message>
  <wsdl:message name="addPhotoResponse">
    <wsdl:part name="status" type="xsd:string"/>
  </wsdl:message>
  <wsdl:message name="replacePhotoRequest">
    <wsdl:part name="oldPhoto" type="xsd:string"/>
    <wsdl:part name="newPhoto" type="xsd:hexBinary"/>
  </wsdl:message>
  <wsdl:message name="replacePhotoResponse">
    <wsdl:part name="status" type="xsd:string"/>
  </wsdl:message>
  <wsdl:portType name="PhotoCatalog">
    <wsdl:operation name="addPhoto">
      <wsdl:input message="tns:addPhotoRequest"/>
      <wsdl:output message="tns:addPhotoResponse"/>
    </wsdl:operation>
    <wsdl:operation name="replacePhoto">
      <wsdl:input message="tns:replacePhotoRequest"/>
      <wsdl:output message="tns:replacePhotoResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="PhotoCatalogBinding" type="tns:PhotoCatalog">
    <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="addPhoto">
      <wsdl:input>
        <mime:multipartRelated>
          <mime:part>
            <soap:body use="literal"/>
          </mime:part>
          <mime:part>
            <mime:content part="photo"
                          type="image/jpeg"/>
          </mime:part>
        </mime:multipartRelated>
      </wsdl:input>
      <wsdl:output>
        <mime:multipartRelated>
          <mime:part>
            <soap:body use="literal"/>
          </mime:part>
          <mime:part>
            <mime:content part="status" type="text/plain"/>
            <mime:content part="status" type="text/xml"/>
          </mime:part>
        </mime:multipartRelated>
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="replacePhoto">
      <wsdl:input>
        <mime:multipartRelated>
          <mime:part>
            <soap:body parts="oldPhoto" use="literal"/>
          </mime:part>
          <mime:part>
            <mime:content part="newPhoto"
                          type="image/jpeg"/>
          </mime:part>
        </mime:multipartRelated>
      </wsdl:input>
```

```
      <wsdl:output>
            <soap:body parts="status" use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
</wsdl:definitions>
```

Given the WSDL displayed in Example 2–9, the WebServicesAssembler `genInterface` command generates the interface in Example 2–10. The `photo` and `newPhoto` parameters are generated with the Java data type `java.awt.Image` because their `mime:content` is defined with `image/jpeg`. An implementation of the interface can now pass JPEG files as attachments for the `photo` part. This is the same for RPC-literal and document-literal (wrapped and bare).

Example 2–10 Generated Interface with a SWA Attachment

```
public interface PhotoCatlog extends java.rmi.Remote{
   public javax.activation.Datahandler addPhoto (java.awt.Image photo) throws
java.rmi.RemoteException;

  public String replacePhoto(java.awt.Image newPhoto, String oldPhoto) throws
java.rmi.RemoteException;
   }
```

Adding SOAP Faults with MIME Attachments

This section describes how to add MIME type attachments to SOAP fault messages.

Note: In OracleAS Web Services, you can add only MIME type attachments to SOAP fault messages.

The attachment is added to the fault message when the Web service implementation class throws an exception. The WS-I Attachments Profile 1.0 specifies that a SOAP fault can contain attachments only if a `<mime:part>` element appears in an operation's output message in the WSDL. Thus, if custom faults appear in a WSDL `operation` that has an output message with an attachment, WebServicesAssembler generates a special exception class that handles attachments in the fault.

The WSDL fragment in Example 2–11 illustrates the `<mime:part>` in the `<wsdl:operation>` clause that declares that an operation can output MIME content. The `namespace` attribute is required only when you are working with RPC-literal messages. The WSDL fragment also illustrates the `<wsdl:fault>` element that defines the fault that the operation can throw.

Example 2–11 WSDL Fragment, that Defines a SOAP Fault

```
<wsdl:operation>
...
<wsdl:output>
  <mime:multipartRelated>
    <mime:part>
      <soap:body parts="string" use="literal|encoded" [namespace="URL"]/>
    </mime:part>
  </mime:multipartRelated>
</wsdl:output>
<wsdl:fault name="string">
    <soap:fault name="string" use="literal|encoded"/>
```

```
</wsdl:fault>
</wsdl:operation>
```

When WebServicesAssembler encounters a `<wsdl:fault>` element, it generates a `com.examples.types.`*Fault_Name*`Type` exception class, where *Fault_Name* is the value of the `name` attribute in the `wsdl:fault` element.

An implementation of a method that throws a generated fault class must also include an implementation of a new instance of the class. It must also include code to add the attachment to the exception.

Writing and implementing the Java classes that contain attachments includes working with classes and methods in the `oracle.webservices.attachments` package. "Understanding the Streaming Attachments API" on page 2-19 describes this API.

Note: Faults with attachments can be added to a Web service only when you are assembling it from a WSDL (top down). They cannot be added when assembling a Web service bottom up.

The following general steps describe how to add attachments to SOAP fault messages and generate them into the service endpoint interface.

1. Provide the WSDL that you want to work with.
2. Edit the WSDL operation that you want to throw a fault with attachment content.
   a. Declare the output element that will contain attachment content in the `<wsdl:output.../>` clause. The clause must contain `<mime:multipartRelated>` content.
   b. Define a `<wsdl:fault.../>` element.

   "Specifying SOAP Faults with Attachments in the WSDL" provides an example of a WSDL that has been edited to include the `wsdl:output` and `wsdl:fault` elements.
3. Use `genInterface` to generate a service endpoint interface.

   ```
   java -jar wsa.jar genInterface -wsdl myWsdl -output c:\appDir
   ```

   The methods in the service endpoint interface that use the attachment data will have a parameter that takes an `attachment` data type. The method will throw an exception that implements `oracle.webservices.attachments.AttachmentFault`.
4. Implement the interface.

   In the implementation, include code to perform the following tasks.
   a. Create a new instance of the exception that implements `oracle.webservices.attachments.AttachmentFault`
   b. Add an attachment to the exception with the `addAttachment` method.

   "Implementing a Method that Throws Faults with Attachments" on page 2-13 illustrates sample code that creates a new instance of the exception and adds the attachment to it.

After this step, you can compile the interface and the implementation, deploy the service, and generate the client code. "Assembling a Web Service from a WSDL" in the *Oracle Application Server Web Services Developer's Guide* provides more detail on the steps for assembling and deploying a Web service top down.

Specifying SOAP Faults with Attachments in the WSDL

To specify a SOAP fault that can throw attachment content, the WSDL operation must identify the namespace of the attachment content, whether its message use is literal or encoded, and the part of the SOAP message it belongs to. The WSDL operation is specified in the `<mime:part>` of the `<wsdl:operation>` clause.

The WSDL fault itself must identify its name and whether its message use is literal or encoded. The `<wsdl:fault>` clause appears as a peer to the WSDL input and WSDL output clauses.

Example 2–12 illustrates a WSDL fragment for the `replacePhoto` binding operation with `wsdl:output` and `wsdl:fault` elements. Because `wsdl:output` identifies MIME content, the fault message defined by `wsdl:fault` will be able to contain attachments.

Example 2–12 WSDL Fragment, Displaying only Port Types and Binding Declarations

```
<wsdl:operation name="replacePhoto">
      <wsdl:input>
        <mime:multipartRelated>
          <mime:part>
            <soap:body parts="oldPhoto" use="literal"
namespace="http://examples.com/PhotoCatalog"/>
          </mime:part>
          <mime:part>
            <mime:content part="newPhoto" type="image/jpeg"/>
          </mime:part>
        </mime:multipartRelated>
      </wsdl:input>
      <wsdl:output>
        <mime:multipartRelated>
          <mime:part>
            <soap:body parts="status" use="literal"
namespace="http://examples.com/PhotoCatalog"/>
          </mime:part>
        </mime:multipartRelated>
      </wsdl:output>
      <wsdl:fault name="InvalidPhoto">
          <soap:fault name="InvalidPhoto" use="literal"/>
      </wsdl:fault>
    </wsdl:operation>
```

Implementing a Method that Throws Faults with Attachments

A method in the generated service endpoint interface that can throw a fault with attachment content contains a parameter that takes an attachment data type and throws a exception that implements `oracle.webservices.attachments.AttachmentFault`. An implementation of the method must create a new instance of the exception. It must also add the attachment to the exception. The `oracle.webservices.attachments` package provides an `FaultWithAttachments.addAttachment` method to do this. "Understanding the Streaming Attachments API" on page 2-19 describes this method.

Example 2–13 illustrates an implementation of the `replacePhoto` operation defined by the WSDL fragment in Example 2–12. The `InvalidPhoto` fault defined in the WSDL is mapped to the `com.examples.types.InvalidPhotoType` exception class.

As required, the implementation of `replacePhoto` creates a new instance of the `InvalidPhotoType` exception. It also adds the attachment to the exception with the

FaultWithAttachments.addAttachment method. The InvalidPhotoType can then be thrown as an exception which causes it to be transmitted as a SOAP fault with attachments.

Example 2–13 Implementing a Method that Throws Faults with Attachments

```
public AttachmentPart replacePhoto(PhotoInfoType oldPhoto, Image newPhoto) throws
            RemoteException,com.examples.types.InvalidPhotoType {
        if(newPhoto == null){
            return null;
        }
        if(oldPhoto.getPhotoID() == -1){
            InvalidPhotoType typeException = new
InvalidPhotoType(-1,"InvalidPhotoType","The PhotoId specified is invalid");
                try{
                    Image image = javax.imageio.ImageIO.read(new File("myImage"));
                    typeException.addAttachment(image,"image/jpeg");
                }catch(Exception e){
                throw new RemoteException("unexpected exeption",e);
            }
                throw typeException;
        }
         AttachmentPart impl = null;
             try{
                MessageFactory factory = MessageFactory.newInstance();
                SOAPMessage message = factory.createMessage();
                impl = message.createAttachmentPart();
            }catch(SOAPException ex){
                return null;
          }
          return impl;
    }
```

Using SOAP Faults with Attachments on the Client

A J2SE or J2EE Web service client can be coded to retrieve SOAP faults with attachments thrown by the service. The oracle.webservices.attachments package provides methods that a client can use to retrieve attachment content from the fault. For example, the getAttachments method can be used to get an iterator of the list of attachments in the fault. The hasAttachments method can be used to query if a fault has an attachment.

Example 2–14 illustrates a client stub, photoCatalog, that can retrieve attachment content. The example assumes that the oldPhoto and newPhoto parameters have been previously defined, and the InvalidPhotoType exception implements oracle.webservices.attachments.AttachmentFault. Since InvalidPhotoType implements AttachmentFault, it is a SOAP fault that can contain attachments.

The hasAttachments method tests for attachments in the fault. If this method returns true, the SOAP fault has attachments. The getAttachments method retrieves all of the attachments in a SOAP fault. This method returns an iterator containing an AttachmentPart object for each attachment in the SOAP fault.

Example 2–14 Web Service Client with Code for Retrieving Attachments

```
public class MyClientExample{

PhotoCatalog photoCatalog = getPhotoCatalogPortType(); //create a client stub
for the web service
```

```
try{
    //assuming oldPhoto and newPhoto are defined
    photoCatalog.replacePhoto(oldPhoto, newPhoto);

   // assuming InvalidPhotoType implements AttachmentFault
}catch(InvalidPhotoType type){
    if(type.hasAttachments()){
       java.util.Iterator it = type.getAttachments();
       AttachmentPart p = (AttachmentPart)it.next();
       //do something useful with the attachment.
    }
}
```

Working with Streaming Attachments

OracleAS Web Services enables you to pass large attachments as a stream. As opposed to the JAX-RPC API, which treats attachments as if they were entirely in memory, streams make the programming model more efficient to use. It also enhances performance and scalability in that there is no need to load the attachment into memory before service execution.

Like embedded attachments, streamed attachments conform to the multipart-MIME binary format. On the wire, messages with streamed attachments are identical to any other SOAP message with attachments.

> **Note:** Consider using streamed attachments if the messages you are trying to pass are over one Megabyte in size. Messages over 100 Megabytes in size should definitely be passed as streamed attachments.

Example 2–15 provides a sample message with a streamed attachment. The first "part" in the message is the SOAP envelope (`<SOAP-ENV:Envelope...`). The second "part" is the attachment, in this case, `myImage.gif`.

Example 2–15 Sample Message with a Streamed Attachment

```
MIME-Version: 1.0
Content-Type: Multipart/Related; boundary=MIME_boundary; type=text/xml;
Content-Description: This is the optional message description.

--MIME_boundary
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: NotSure/DoesntMatter

<?xml version='1.0' ?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
..
<DocumentName>MyImage.gif</DocumentName>
..
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

--MIME_boundary
Content-Type: image/gif
```

```
Content-Transfer-Encoding: binary
Content-ID: AnythingYoudLike

...binary GIF image...
--MIME_boundary--
```

There are some limitations on the use of streamed attachments in OracleAS Web Services.

- A Web service port can send only embedded attachments or streamed attachments; not both.
- Handlers cannot obtain access to streamed attachments.
- Any message with streamed attachments will appear to all handlers as a SOAP message without attachments.

Streaming is enabled by adding an extra parameter to each service or stub operation that wants to stream attachments. This parameter must be of type `Attachments`. The `Attachments` interface provides methods for reading incoming attachment streams (service requests or client responses) and for adding outgoing attachment streams (service responses or client requests).

The `Attachments` interface belongs to the `oracle.webservices.attachments` package. "Understanding the Streaming Attachments API" on page 2-19 describes this package.

Assembling Streaming Attachments into a Web Service

This section describes how to assemble a Web Service that uses streaming attachments.

- Assembling a Web Service that Supports Streaming Attachments Bottom Up
- Assembling a Web Service that Supports Streaming Attachments Top Down

Assembling a Web Service that Supports Streaming Attachments Bottom Up

The model for assembling a Web service bottom up that supports streaming attachments, is the same as any other service based on Java classes. "Assembling a Web Service with Java Classes" in the *Oracle Application Server Web Services Developer's Guide* provides more information on assembling a Web service bottom up with Java classes.

1. Write a public interface that contains the remote methods that you want to expose as a service. In this case, one or more of the methods will use the Attachment API.

   "Writing an Interface for Steaming Attachments" on page 2-17 provides more information on this step.

2. Write the implementation of the service.

   "Implementing a Service Interface that Uses Streaming Attachments" on page 2-17 provides more information on this step.

3. Write the stub code for the Web service client. To use a stub to send or receive attachments, you must create the `Attachments` object before the call.

   "Writing Stub Code to Handle Streaming Attachments" on page 2-18 provides more information on this step.

4. Generate the service artifacts by running the WebServicesAssembler with the `assemble` command.

5. Deploy the service and bind the application.

6. Generate the client-side code.
7. Compile and run the client.

Writing an Interface for Steaming Attachments If you want to stream attachments as a part of your Web service, then at least one of the methods within the service interface must contain a parameter of type `Attachments`. The `Attachments` type indicates objects will be streamed by OracleAS Web Services.

Example 2–16 illustrates the service interface for a Web service that stores large image files in a local database. The request includes an image name, a description, and the image to be stored. Note that the `attachments` parameter of type `Attachments` indicates that the image will be streamed.

Example 2–16 Service that Uses Streaming Attachments

```
public interface ImageStore extends java.rmi.Remote {
    public void storeImage (String name, String desc, Attachments attachments)
throws java.rmi.RemoteException;
}
```

If you use this interface to generate a Web service bottom up, WebServicesAssembler will detect the parameter of type `Attachments`. Support for streaming attachments will automatically be generated into the Web service. Example 2–18 illustrates the relevant parts of the WSDL generated by WebServicesAssembler.

Implementing a Service Interface that Uses Streaming Attachments Implementations of interfaces that include data as streaming attachments use the methods and classes of the `oracle.webservices.attachments` API to work with the attachment data.

Example 2–17 illustrates the implementation of the interface in Example 2–16 for a Web service that stores large image files in a local database. The implementation uses the `getIncomingAttachments` method to capture the incoming streaming attachment as an `Attachment` object. The `getId` method retrieves the attachment's metadata and `getInputStream` stores the stream image bytes.

Example 2–17 Service Implementation that Employs Streaming Attachments

```
class ImageStoreImpl {
    public void storeImage(String name, String desc, Attachments attachments)
throws Exception {
        IncomingAttachments incomingAtts = attachments.getIncomingAttachments();
        {
            if (incomingAtts == null || !incomingAtts.hasNextAttachment())
                throw new Exception("Expected request attachments");
            Attachment imageAtt = incomingAtts.nextAttachment();
            String id = imageAtt.getId();
            DataHandler dataHandler = imageAtt.getDataHandler();
            InputStream imageStream = dataHandler.getInputStream();
            //-- Store image metadata and stream image bytes
            if (incomingAtts.hasNextAttachment())
                throw new Exception("Expected only one attachment");
        }
    }
}
```

WSDL Elements for a Service with Streaming Attachments The presence of the `<stream-attachments>` extension (in the namespace

http://oracle.com/schemas/webservices/streaming-attachments) in the WSDL indicates that the Web service supports streaming attachments.

Example 2–18 illustrates the relevant parts of the WSDL that WebServicesAssembler generates from the interface in Example 2–16. The <stream-attachments> element that identifies streaming is highlighted in bold. "WSDL Extensions for Streaming Attachments" on page 2-19 contains more information about the <stream-attachments> element and its schema.

Example 2–18 WSDL Elements for a Streaming Attachment

```
<message name="storeImageRequest">
    <part name="name" type="string"/>
    <part name="desc" type="string"/>
</message>
<portType name="ImageStorePortType">
    <operation name="storeImage">
        <input name="storeImageRequest" message="tns:storeImageRequest"/>
        <output name="storeImageResponse" message="tns:storeImageResponse"/>
    </operation>
</portType>
<binding name="ImageStoreBinding" type="tns:ImageStorePortType">
    <soap:binding style="rpc" />
    <operation name="addPerson">
        <soap:operation style="encoded" wsdl:required="true" />
        <sa:stream-attachments name="attachments" />
        <input name="storeImageRequest">
            <soap:body use="encoded" />
        </input>
        <output name="storeImageResponse">
            <soap:body use="encoded" />
        </output>
    </operation>
</binding>
```

Writing Stub Code to Handle Streaming Attachments The stub is the client side of the Web service. If the service wants to receive streamed attachments, the client must send them. If the server sends streamed attachments, the client must read them.

To handle streaming attachments, client code must create a new instance of an AttachementFactory and create an Attachments object. For a request attachment, the addAttachment method can be used to add the attachment to the output stream.

For a response attachment, the client code must create a new instance of an AttachementFactory and create an Attachments object. The client can capture the attachments with the iterator methods in the IncomingAttachments interface.

Example 2–19 illustrates stub code to store a streaming attachment. In the example, an AttachmentFactory is instantiated with the newInstance method and an Attachments object is created with the createAttachments method. The addAttachment method adds the data to the output stream.

Example 2–19 Client Code to Store a Streaming Attachment

```
public void storeImageFile (String fileName) {
    //--  Create the attachment objects
    AttachmentFactory factory = AttachmentFactory.newInstance();
    Attachments atts = factory.createAttachments();
    Attachment imageAtt = factory.createAttachment (fileName, "image/gif",
              new FileInputStream(fileName));
    atts.getOutgoingAttachments().addAttachment(imageAtt);
```

```
    storeImagePort.storeImage (fileName, "File stored at " + fileName, atts);
}
```

Assembling a Web Service that Supports Streaming Attachments Top Down

Assembling a Web service that supports streaming attachments follows the general steps described in "Assembling a Web Service from a WSDL" in the *Oracle Application Server Web Services Developer's Guide*. This section summarizes these steps.

To support streaming attachments, you must also edit the WSDL to add the `<stream-attachments>` element. Example 2–18 illustrates a WSDL that contains the `<stream-attachments>` element.

1. Provide a WSDL from which the Web service will be generated.
2. Edit the WSDL to add the elements that allow the Web service to support streaming attachments. "WSDL Extensions for Streaming Attachments" describes the `<stream-attachments>` element.
3. Use the WSDL as input to the WebServicesAssembler `genInterface` command.
4. Compile the generated interface and type classes.
5. Write the Java Service Endpoint interface for the Web service you want to provide.
6. Compile the Java Service Endpoint interface.
7. Generate the service by running the WebServicesAssembler tool with the `topDownAssemble` command.
8. Deploy the service.
9. Generate the client code.

WSDL Extensions for Streaming Attachments

Streaming attachments uses a WSDL extension, `<stream-attachments>`, to notify the WebServicesAssembler tool that attachment data will be streamed. The extension can be used only at the binding operation level. The element is specified in an Oracle namespace and takes a single string attribute, `name`. The `name` attribute is used as the parameter name when generating Java classes and interfaces. Example 2–18 illustrates a WSDL that contains the `<stream-attachments>` element. Example 2–20 illustrates the `<stream-attachments>` schema.

Example 2–20 XML Schema for Streaming Attachments

```
<schema
    xmlns="http://www.w3.org/2000/10/XMLSchema"
    xmlns:sa="http://oracle.com/schemas/webservices/streaming-attachments"
    targetNamespace="http://oracle.com/schemas/webservices/streaming-attachments">
  <element name="stream-attachments" type="sa:streamAttachemntsType"/>
  <complexType name="streamAttachemntsType">
    <attribute name="name" type="string" use="required"/>
  </complexType>
</schema>
```

Understanding the Streaming Attachments API

The Attachments API provides a programmatic interface to streaming attachments. Streamed attachments are directional; they can be classified as "incoming" and "outgoing". Incoming streams can only be read and outgoing streams can only be written to. The API resides in the `oracle.webservices.attachments` package.

Table 2–2 summarizes the interfaces and classes in the Attachments API. The sections following the table provide more detail.

Table 2–2 Interfaces in the Attachments API

Interface or Class	Description
`Attachments` interface	Root interface for managing attachments. This interface contains methods for storing attachments as either incoming or outgoing attachments. See "Interface for Attachments" for more information.
`IncomingAttachments` interface	Provides iterators to process incoming attachments. These can be request attachments on the server or response attachments on the client. See "Interface for Incoming Attachments" for more information.
`OutgoingAttachments` interface	Collects outgoing attachments for streaming after the message is serialized. These can be request attachments on the client or response attachments on the server. See "Interface for Outgoing Attachments" for more information.
`Attachment` interface	The `Attachment` object consists of a `String` ID to identify the attachment and a data handler to retrieve the data. See "Interface for Attachment Objects" for more information.
`AttachmentFactory` class	This factory class creates instances of `Attachments` and `Attachment` objects. See "Factory Class for Attachment Objects" for more information.

Interface for Attachments

`Attachments` is the root interface for managing attachments. The `getIncomingAttachments` method returns an `IncomingAttachments` object. It returns `null` if there are no incoming attachments. The `getOutgoingAttachments` method always returns an `OutgoingAttachments` object.

```
public interface Attachments {
    IncomingAttachments getIncomingAttachments();
    OutgoingAttachments getOutgoingAttachments();
}
```

Interface for Incoming Attachments

The `IncomingAttachments` interface is used for request attachments on the server and for response attachments on the client. It is, essentially, an iterator over attachments. The interface includes a `close` method that tells the runtime to flush any remaining attachments from the stream. This must be called after all attachments have been processed so the connection can be cleared for the next request.

```
interface IncomingAttachments {
    public boolean hasNextAttachment() throws IOException;
    public Attachment nextAttachment() throws IOException;
    public void close() throws IOException;
}
```

Interface for Outgoing Attachments

The `OutgoingAttachments` interface is used for request attachments on the client and for response attachments on the server. It collects attachments for streaming after the message is serialized. The interface includes an `addAttachment` method to add attachments to the output stream. You can create attachments using the methods described in "Factory Class for Attachment Objects".

```
interface OutgoingAttachments {
    public void addAttachment (Attachment attachment);
}
```

Interface for Attachment Objects

The `Attachment` object consists of a `String` ID to identify the attachment and a data handler to retrieve the data. It is used by the `addAttachment` method in the `OutgoingAttachments` interface to add an attachment to the output stream.

```
interface Attachment {
    public String getId();
    public DataHandler getDataHandler();
}
```

Factory Class for Attachment Objects

The package provides a factory class for creating instances of `Attachments` and `Attachment` objects. When you are using streaming attachments in a stub, you must create the `Attachments` object before you call the service.

```
abstract class AttachmentFactory {
    public Attachments createAttachments();
    public Attachment createAttachment (String id,
                String contentType, InputStream attachmentStream);
    public Attachment createAttachment (String id, DataHandler handler);
    public static AttachmentFactory newInstance();
}
```

Working with DIME Attachments

Direct Internet Message Encapsulation (DIME) is a format for streaming multi-part message attachments over the wire. DIME can be applied to SOAP messages with attachments. DIME separates each record, such as a SOAP message or attachment, with a simple binary SOAP header that describes the size and type of the payload. DIME also allows a given piece of payload to be broken into multiple records. This allows the sender to stream data through a buffer of constrained size.

Compared to the multipart-MIME encoding format which OracleAS Web Services uses by default, DIME requires much less processing effort to encode or decode.

OracleAS Web Services lets you choose between supporting interoperable DIME-encoded messages and messages with Oracle-proprietary DIME encoding. Interoperable DIME encoding for messages with attachments is implemented by adding extensions to the WSDL. Oracle-proprietary DIME encoding is implemented by allowing WebServicesAssembler to generate code that applies DIME encoding to all SOAP messages with attachments.

The following sections provide more detailed information on each of these techniques.

- Creating Interoperable DIME-Encoded Messages
- Implementing Oracle-Proprietary DIME Encoding

Creating Interoperable DIME-Encoded Messages

These WSDL extensions are an implementation of Microsoft's DIME support. This technique creates messages with attachments that are interoperable with other vendors, particularly Microsoft, who implement DIME extensions in the WSDL. The

extensions cause any binary data in the envelope to be extracted and placed in attachments. These messages and their attachments are then encoded in DIME.

The WSDL extensions are defined in the "WSDL Extension for SOAP in DIME" specification. This specification is located at:

http://www.gotdotnet.com/team/xml_wsspecs/dime/WSDL-Extension-for-DIME.htm

The schema that defines the extensions is located at:

http://schemas.xmlsoap.org/ws/2002/04/dime/wsdl/

The specification defines a DIME element, `<dime:message>`. If this element is added to the WSDL, then any binary data in the SOAP message is automatically removed and put into attachments. The message and its attachments are then sent in DIME format instead of the default multipart-MIME format.

The `<dime:message>` element can be inserted into the `<wsdl:input>` and/or `<wsdl:output>` elements of a Web service operation bindings definition. Table 2–3 describes the attributes that `<dime:message>` must include.

Table 2–3 <dime:message> Required Attributes

Attribute	Description
wsdl:required="true"	Indicates that the presence of a WSDL is required. This attribute must be set to `true`.
layout="uri"	Specifies how the primary SOAP message references the attachments in a DIME message. The following are the possible values for the URI. ▪ `http://schemas.xmlsoap.org/ws/2002/04/dime/closed-layout`—Specifies that the primary SOAP message references all parts of a DIME message in the proper order. ▪ `http://schemas.xmlsoap.org/ws/2002/04/dime/open-layout`—Specifies that the DIME message can include additional attachments, even though they are not referenced by the SOAP message. These additional attachments must follow all of the attachments in the DIME message that are referenced by the SOAP message.

Example 2–21 illustrates a WSDL skeleton with the extensions for DIME encoding highlighted in bold.

Example 2–21 WSDL Skeleton, with the Extensions for DIME Encoding

```
<wsdl:definitions ...>
 <wsdl:binding ...>
  <soap:binding .../>
  <wsdl:operation ...>
    <soap:operation .../>
    <wsdl:input>
     <dime:message layout="uri" wsdl:required="true"/>?
     <-- extensibility elements -->
    </wsdl:input>
    <wsdl:output>
     <dime:message layout="uri" wsdl:required="true"/>?
     <-- extensibility elements -->
     </dime:message>
    </wsdl:output>
  </wsdl:operation>
 </wsdl:binding>
```

```
</wsdl:definitions>
```

Implementing Oracle-Proprietary DIME Encoding

The WebServicesAssembler supports the DIME-encoded format by providing a `useDimeEncoding` argument. This argument will apply DIME encoding to all streaming SOAP messages with attachments in a generated service or stub.

The messages generated with the `useDimeEncoding` argument are not interoperable and are incompatible with non-Oracle Web services or earlier versions of Oracle Application Server. Generating a Web service with this argument is useful for in-house projects where performance is key.

See "useDimeEncoding" in "General Web Services Assembly Arguments" in the *Oracle Application Server Web Services Developer's Guide* for more information on this argument.

Working with Attachments in WSIF

For information on enabling WSIF clients to handle message attachments, see "Adding Message Attachments in WSIF" on page 9-23.

Limitations

See "Working with Message Attachments" on page E-9.

Additional Information

For more information on:

- enabling WSIF clients to handle attachments, see Web Services Invocation Framework, see Chapter 9, "Using Web Services Invocation Framework".
- assembling a Web service top down, see "Assembling a Web Service from a WSDL" in the *Oracle Application Server Web Services Developer's Guide*.
- using Java classes to assemble a Web service, see "Assembling a Web Service with Java Classes" in the *Oracle Application Server Web Services Developer's Guide*.
- using EJBs to assemble a Web service, see "Assembling a Web Service with EJBs" in the *Oracle Application Server Web Services Developer's Guide*.
- using JMS topics and destinations to assemble a Web service, see "Assembling Web Services with JMS Destinations" in the *Oracle Application Server Web Services Developer's Guide*.
- using database resources, such as PL/SQL packages, SQL queries, DML statements, Oracle Streams AQ, or server-side Java classes, to assemble a Web service, see "Assembling Database Web Services" in the *Oracle Application Server Web Services Developer's Guide*.
- assembling a J2EE Web Service client, see "Assembling a J2EE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.
- assembling a J2SE Web service client, see "Assembling a J2SE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.
- using WebServicesAssembler commands to assemble Web service artifacts, see "Using WebServicesAssembler" in the *Oracle Application Server Web Services Developer's Guide*.

3 Managing Web Services

This chapter provides an overview of Web service management for Oracle Application Server Web Services.

- Understanding Web Service Management
- Configuring Server-Side Management Information
- Data Flow for Management Information in a J2SE Client
- Data Flow for Management Information in a J2EE Client
- Application Server Control Support for Web Service Management
- Working with Capability Assertions

Understanding Web Service Management

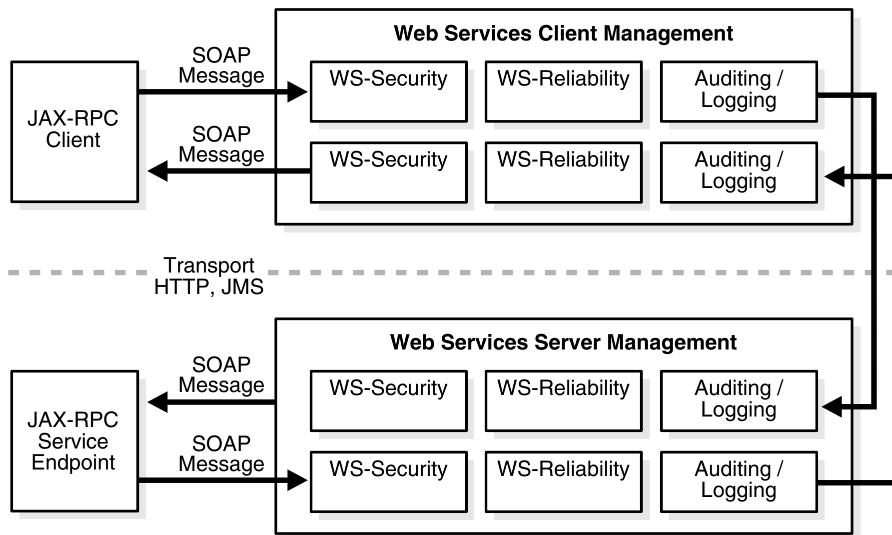
Web service management is a set of policies applied against a message en-route to a target. In the case of OracleAS Web Services, the messages are SOAP requests, responses and faults, and the targets are the client and the business logic on the server. OracleAS Web Services supports policies for the following management features.

- security—configures authentication, integrity with digital signatures, and confidentiality with encryption based on the WS-Security standard. For more information on the security features supported by OracleAS Web Services, see the *Oracle Application Server Web Services Security Guide*.
- reliability—configures guaranteed message delivery and ordering, and eliminates duplicate messages based on the WS-Reliability standard. For more information on the reliability features supported by OracleAS Web Services, see Chapter 5, "Ensuring Web Service Reliability".
- auditing and logging—auditing keeps a complete, persistent record of SOAP requests and faults. Logging enables content-based logging by using XPath to query the inbound and outbound SOAP messages. For more information on the auditing and logging features supported by OracleAS Web Services, see Chapter 6, "Auditing and Logging Messages".
- life cycle management—enables and disables services and their managed capabilities. For more information, see "Web Service Management Life Cycle" on page 3-4.
- deployment—configures the OracleAS Web Services proprietary deployment descriptors based on J2EE 1.4 standards. For more information, see Chapter 18, "Packaging and Deploying Web Services", in the *Oracle Application Server Web Services Developer's Guide*.

Figure 3–1 illustrates the passage of a SOAP request from the client to the business logic on the server. The SOAP request passes from the client through a layer of management for outbound messages before it is sent over the wire. When it reaches the server, the request must pass through a layer of management configuration for inbound messages before it is used by the business logic. The values given to the server-side management must be coordinated with the values in the client; otherwise, the request might be rejected before reaching the service implementation.

Similarly, the response from the business logic on the server reverses the process. The response must pass through a layer of management for outbound messages before it is sent over the wire. When it reaches the client, the response must pass through a layer of management for inbound messages before the client can use it.

Figure 3–1 Web Services Message Flow with Management Enabled



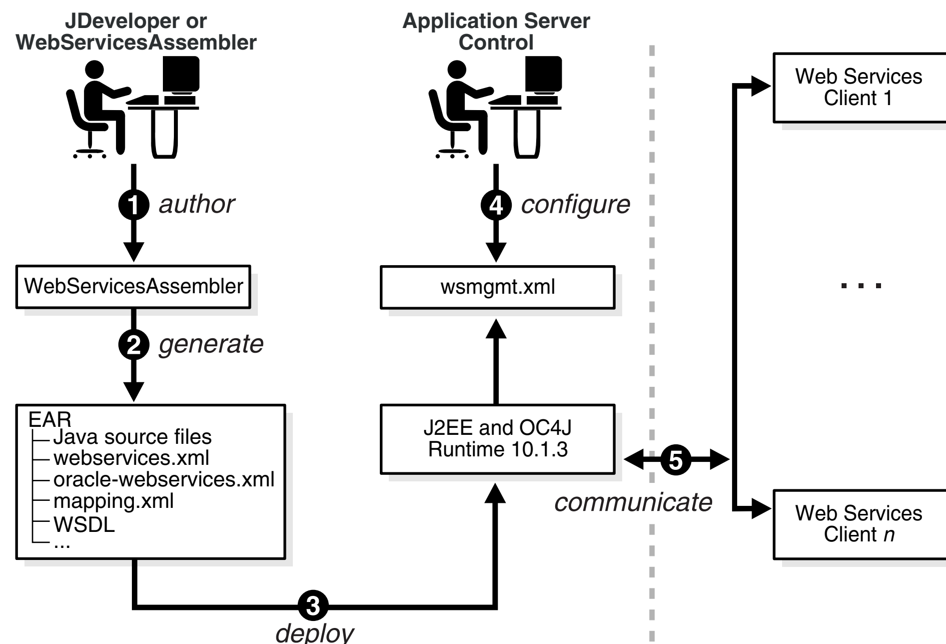
As Figure 3–1 suggests, Web service management can consist of as many as four separate configurations. The server can have separate configurations for inbound messages (requests) and outbound messages (responses). The client can have separate configurations for outbound messages (requests) and inbound messages (responses).

These management policies can be configured pre-deployment within Oracle JDeveloper. Post-deployment you can configure the policies within the Oracle Application Server Control management environment. For more information see the JDeveloper on-line help and "Overview of Managing Web Services" and "Enabling and Disabling Web Services Management Features" in the Application Server Control on-line help.

Web Services Management Environment

Figure 3–2 illustrates the management environment. Three major product components make up this solution: the design time with Oracle JDeveloper or WebServicesAssembler, the runtime with OracleAS Web Services Release 10.1.3, and the management environment with Oracle Application Server Control.

Figure 3–2 Web Services Management Data Flow in the Server



The following steps correspond to the numbers in the figure:

1. A developer uses Oracle JDeveloper or its command-line counterpart, WebServicesAssembler, to author a Web service and to configure the Web service management information that will be applied at runtime.
2. JDeveloper (or WebServicesAssembler) can then be used to assemble the Web service and package it in an enterprise archive (EAR). Among other Web service artifacts, the EAR also contains the `oracle-webservices.xml` file that describes management policies. This file is the application server-specific binding to the Oracle infrastructure. The `oracle-webservices.xml` file can be considered to be a extension of the standard JAX-RPC `webservices.xml` file, which defines platform-independent Web service behavior.
3. Upon deployment to OC4J, the policies in the `oracle-webservices.xml` file are automatically copied to the runtime Web services management policy file, `wsmgmt.xml`. This file resides at *`ORACLE_HOME`*`\j2ee\home\config\wsmgmt.xml` where *`ORACLE_HOME`* is the installation directory for OC4J.
4. Application Server Control can be used to further manipulate the management policies in `wsmgmt.xml`.
5. The Web service clients can communicate with the Application Server through the OC4J runtime.

As this figure illustrates, the creation and administration of Web services management is completely separate from the Web service, business logic, and client implementation. The management configuration can be changed independently of the implemented business logic and does not require redeployment.

If you redeploy the EAR, any changes that Application Server Control has made to the policy in `wsmgmt.xml` will be overwritten. In effect, redeploying is the same as undeploying and then deploying. When a Web service is undeployed, all of the Web service management configuration is removed from `wsmgmt.xml`.

In addition to the server-side policy configuration, a symmetric policy configuration is often required on the client side. OC4J provides the option to have the server-side policy expressed in the WSDL in the form of capability assertions. By providing this information in the WSDL, client-generation tools can inspect this publicly available contract. The tools can use the capability assertions to know what questions to ask in order to create a configuration that will allow the client to communicate with the server. For more information on how information about the Web service management configuration can be added to the WSDL, see "Working with Capability Assertions" on page 3-18.

Web Service Management Life Cycle

An important part of managing a Web service is managing its life cycle. This includes the ability to enable or disable the service's management configuration. Application Server Control provides the ability to enable and disable security, reliability, auditing, and logging. Application Sever Control can also be used to enable or disable the entire Web service.

Other aspects of life cycle management, specifically deployment and undeployment, are covered in the *Oracle Containers for J2EE Deployment Guide*.

Configuring Server-Side Management Information

There are a number of techniques that you can use to configure server-side management information for a Web service. For example, you can write the management information into an existing `oracle-webservices.xml` file by hand or you can allow JDeveloper or WebServicesAssembler to automate the process. If the Web service is already deployed, you can use Application Server Control to configure many of the management options. The following sections summarize the ways in which you can configure server-side management information:

- By Hand
- With JDeveloper
- With WebServicesAssembler
- With Application Server Control and WebServicesAssembler

By Hand

1. Examine the `oracle-webservices.xsd` schema and write the management information into an existing `oracle-webservices.xml` file by hand.
2. Run the appropriate WebServicesAssembler *`Assemble` command to assemble your Web service.
   - Use the `ddFileName` argument to specify the modified `oracle-webservices.xml` file.
   - Optionally, use the `genQosWsdl` argument to insert capability assertions into the WSDL if you are generating the Web service bottom up or use `genQosWsdl` if you are generating the Web service top down.
3. Deploy the Web service.

With JDeveloper

1. Use the wizards in JDeveloper to configure a Web service. This will produce an `oracle-webservices.xml` file.

2. Run the appropriate wizard in JDeveloper to configure management information in the Web service.
3. Select the "add capability assertions" option if you want to insert capability assertions into the WSDL.
4. Deploy the Web service.

With WebServicesAssembler

1. Run WebServicesAssembler with the `genDDs` command to create a generic `oracle-webservices.xml` file.

   This step will create a "skeleton" deployment descriptor file. This file will not contain any management configuration information. It will, however, provide the basic structure of the file and indicate where the management configuration should be placed.
2. Examine the `oracle-webservices.xsd` schema and write the management information into an existing `oracle-webservices.xml` file by hand.
3. Run the appropriate `*Assemble` command to assemble your Web service.
   - Use the `ddFileName` argument to specify the modified `oracle-webservices.xml` file.
   - Optionally, use the `genQosWsdl` argument to insert capability assertions into the WSDL.
4. Deploy the Web service.

With Application Server Control and WebServicesAssembler

This scenario enables you to establish a default server-side configuration without having to use JDeveloper or author the configuration by hand. The following steps describe how to use Application Server Control to define and populate a default management configuration for the server side. You then copy this configuration from the runtime configuration into the deployable EAR file.

1. Run WebServicesAssembler with the appropriate `*Assemble` command to create a Web service. Note that a Web service created by WebServicesAssembler will not have management information unless you explicitly build it in.
2. Deploy the Web service without management information. The deployment will cause a `wsmgmt.xml` file to be created in OC4J.
3. Use the Web services management screens in Application Server Control to configure server-side Web service management options. These values will be reflected in the `wsmgmt.xml`. file.
4. Copy Web service management information in the `wsmgmt.xml` file into the `oracle-webservices.xml` file that was created in Step 1.
   a. Locate the `<runtime>` element and the `<operations>` element that appear under the `<port>` element in the `wsmgmt.xml` file. Note the value of the `port` attribute in the `<port>` element.
   b. Locate the `<port-component>` element in the `oracle-websrevices.xml` file, whose `name` attribute has the same value as the `port` attribute of the `<port>` element in the `wsmgmt.xml` file.
   c. Copy the `<runtime>` element, the `<operations>` element, and all of their child elements in the `wsmgmt.xml` file.

   d. Paste these elements into the `oracle-webservices.xml` file, as a children of the `<port-component>` element located in Step **4b**.

   e. Repeat Steps **4a** to **4d** for each port you want to expose as a Web service.

5. Re-run the appropriate *`Assemble` command for your Web service.

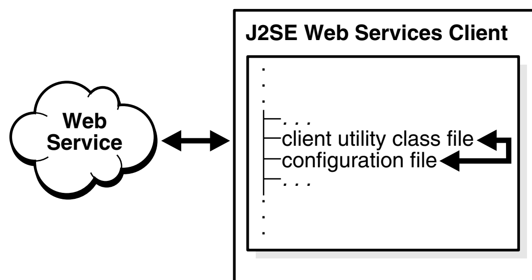
   - Use the `ddFileName` argument to specify the modified `oracle-webservices.xml` file.
   - Optionally, use the `genQosWsdl` option to insert capability assertions into the WSDL.

6. Re-deploy the Web service.

Data Flow for Management Information in a J2SE Client

Figure 3–3 presents an overview of how Web service management information is developed, transmitted, and manipulated in the J2SE client environment.

Figure 3–3 Web Services Management Data Flow in a J2SE Client



The data flow illustrated in Figure 3–3 is summarized in the following steps.

1. A developer creates a client-side configuration file. This can be done by using the wizards in JDeveloper, or by hand, using the `oracle-webservices-client-10_0.xsd` schema. The client-side configuration file contains the client-side configuration for Web services management features such as security, reliability, and auditing. (Note: logging is not available on the client.)

2. Use either JDeveloper or WebServicesAssembler to generate a J2SE client JAR file. If WebServicesAssembler is used, the `ddFileName` argument to the `genProxy` command specifies the configuration file.

   Among the files contained in the J2SE client JAR are:

   - proxy class file—this file can send invocations to the server. This file typically has a <*generated_name*>`_Stub.java` extension, where *generated_name* is derived from the target namespace and port name in the WSDL.
   - client-side configuration file—this file typically resides in the same directory as the proxy class file. It has the same generated name as the proxy class file, but with a `_Stub.xml` extension.

   For example, given a service endpoint interface, `test\proxy\Test`, you can use WebServicesAssembler to generate a proxy class file, `test\proxy\runtime\Test_Stub.class`. The client-side configuration file, specified with the `ddFileName` argument, will be copied to the same directory and will be called `test\proxy\runtime\Test_Stub.xml`.

"Assembling a J2SE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide* provides more information on using WebServicesAssembler to generate J2SE Web service clients.

3. At runtime, the generated proxy class file will read the XML file and apply the management configuration settings to the messages.
4. Requests and responses can be passed between the client and server.

Configuring Management Information for a J2SE Client

You can use JDeveloper or WebServicesAssembler to add a management configuration to the J2SE client.

- With JDeveloper
- With WebServicesAssembler

With JDeveloper

1. Given a WSDL belonging to a Web service, use the wizards in JDeveloper to create the proxy class file and the client-side configuration file.
2. Run the appropriate wizards in JDeveloper to configure management information for the client.
3. Run the client.

With WebServicesAssembler

"Assembling a J2SE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide* provides more detailed information on creating J2SE Web service clients.

1. Create a client-side configuration file. You can refer to the `oracle-webservices-client-10_0.xsd` schema and write the file by hand.
2. Given a WSDL belonging to a Web service, run WebServicesAssembler with the `genProxy` command to create a proxy class file. Use the `ddFileName` argument to specify the client-side configuration file.

```
<oracle:genProxy
    wsdl="HelloService.wsdl"
    output="src"
    packageName="oracle.demo.hello"
    ddFileName="clientConfig.xml"/>
```

   This results in the generation of several Java source files beneath the directory specified by the `output` parameter. In this case, the Java source files are generated into the `src` directory. The generated Java source file that implements the port is `oracle/demo/hello/runtime/HttpSoap11Binding_Stub.java`. The name of this file is based on the way that Java class names are derived from the `targetNamespaces` and `port` names in the WSDL. In this example the WSDL targetNamespace is `http://hello.demo.oracle/` and the name of the port is `HttpSoap11Binding`.

   You can modify this default behavior using additional inputs to WebServicesAssembler. The generated `_Stub.xml` file will always have the same root name as the generated Java source file which implements the port.

   Note that since the `_Stub.xml` file is generated into the source directory, you must ensure that this file is also copied to the directory into which you compile the

Java source files. The following example shows how you might do this in an Ant task.

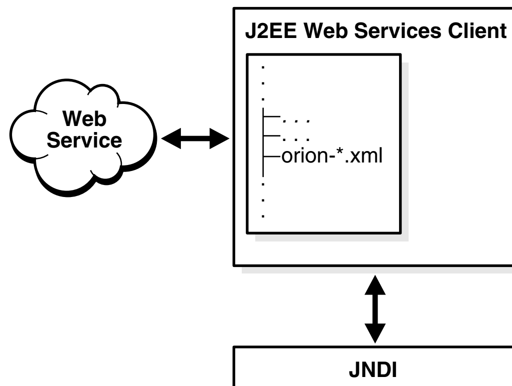
```
<copy todir="classes" >
   <fileset dir="src">
      <include name="**/*_Stub.xml"/>
   </fileset>
</copy>
```

3. Use the client utility class file created by `genProxy` as your application client, or use it as a template to write your own client code. The client utility class file is one of a number of files created by `genProxy`.
4. Run the client.

Data Flow for Management Information in a J2EE Client

Figure 3–4 illustrates how Web service management data is transmitted in a J2EE client.

Figure 3–4 Web Service Management Data Flow in a J2EE Client



1. To access the Web service, the J2EE client performs a JNDI lookup for a reference to the service endpoint interface.
2. At runtime, the object implementing the interface will use the management configuration used in either `orion-ejb-jar.xml`, `orion-web.xml`, or `orion-application-client.xml`, depending on the type of client.

   The `orion-*.xml` file which is contained in the deployment archive has a `<service-ref-mapping>` element which contains the information needed to generate the service endpoint interface at runtime. The Web service management information is contained within the `<service-ref-mapping>` element.

   "Adding OC4J-Specific Platform Information" in the *Oracle Application Server Web Services Developer's Guide* provides more information on the `<service-ref-mapping>` element.

Configuring Management Information for a J2EE Client

You can use JDeveloper or WebServicesAssembler to add a management configuration to the J2EE client.

- With JDeveloper
- With WebServicesAssembler

With JDeveloper

1. Given a WSDL belonging to a Web service, use JDeveloper to create the Java service endpoint interface file and the client component descriptor file. The configuration file will be either `orion-ejb-jar.xml`, `orion-web.xml`, or `orion-application-client.xml`, depending on the type of client.

2. Edit the `<service-ref-mapping>` tag in the `orion-*.xml` generated for your particular client. Enter the client-side Web service management configuration within this tag. For more information on the `<service-ref-mapping>` tag, see "Adding OC4J-Specific Platform Information" in the *Oracle Application Server Web Services Developer's Guide*.

3. Assemble and deploy the client module. "How to Assemble a J2EE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide* provides more information on assembling and deploying J2EE Web service clients.

With WebServicesAssembler

1. Run WebServicesAssembler with the `genInterface` command to create a Java service endpoint interface file.

2. Edit the `<service-ref-mapping>` tag in the `orion-*.xml` generated for your particular Web service. Enter the client-side Web service management configuration within this tag. "Adding OC4J-Specific Platform Information" in the *Oracle Application Server Web Services Developer's Guide* provides more information on the `<service-ref-mapping>` tag.

3. Assemble the client deployment module:

   a. Compile all of the client files.

   b. Copy deployment descriptor files to their appropriate locations. "Packaging Web Service Applications" in the *Oracle Application Server Web Services Developer's Guide* describes where files should reside for servlet, EJB, or JSP Web service clients.

   c. Package the client deployment module.

4. Deploy the client deployment module.

5. Run the client.

Dynamic Client-Side Configuration

In all of the client-side dynamic configuration scenarios described in the following sections, the client provides a configuration for each invocation. The client does this by setting the value for the following property on either the call or the stub/port.

```
oracle.webservices.ClientConstants.WSM_INTERCEPTOR_PIPELINE_CONFIG
```

The value provided can be either an XML DOM element reference to the root node, or a `java.io.File` object of a document that conforms to the Web service management client-side configuration schema. If an XML DOM element reference is provided, the client must load and parse the configuration. If a `java.io.File` object is provided, the runtime will perform the loading and parsing.

The configuration provided by the `ClientConstants.WSM_INTERCEPTOR_PIPELINE_CONFIG` attribute will override any static configuration. The following example sets the configuration for an XML DOM element reference.

```
...setProperty(
        "oracle.webservices.ClientConstants.WSM_INTERCEPTOR_PIPELINE_CONFIG",
        parseAndReturnElement(config));
```

This example sets the configuration for a `java.io.File` object.

```
...setProperty(
         "oracle.webservices.ClientConstants.WSM_INTERCEPTOR_PIPELINE_CONFIG",
         new java.io.File("config file path"));
```

The override is all or nothing: if a dynamic configuration is provided, then any static configuration is completely ignored.

The following sections provide code examples of passing a management configuration to DII, dynamic proxy, static proxy, and J2EE Web service clients. In all of these cases, the configuration is defined in the following lines.

```
String config =
             "<port-info>" +
               ...
             "</port-info>";
```

This can be entered into the configuration file for J2SE clients or the deployment descriptors for J2EE clients.

- Providing Dynamic Configuration for a DII Web Service Client
- Providing Dynamic Configuration for a Dynamic Proxy Web Service Client
- Providing Dynamic Configuration for a Static Proxy Web Service Client
- Providing Dynamic Configuration for a J2EE Web Service Client

Providing Dynamic Configuration for a DII Web Service Client Example 3–1 displays management configuration being passed to a DII client. The `setProperty` statement is highlighted in bold. Note the presence of the `<call-property>` element in the example. You can use this element to pass information such as an endpoint address, a user name and password, and any other standard and proprietary properties.

Example 3–1 Properties and Management Configuration Dynamically Passed to a DII Client

```
...
String config =
             "<port-info>" +
                "<call-property>" +
                    "<name>...</name>" +
                    "<value>...</value>" +
                "</call-property>" +
               "<runtime>" +
                  "<security>" + ... + "<security>" +
                "</runtime>" +
                "<operations>" +
                   "<operation name='echo'>" +
                      "<runtime>" +
                         "<security>" + ... + "<security>" +
                      "</runtime>" +
                   "</operation>" +
                "</operations>" +
             "</port-info>";
       call.setOperationName(
             new QName("http://oracle.com/test/wsdl", "echo"));
       call.setProperty(
             "oracle.webservices.ClientConstants.WSM_INTERCEPTOR_PIPELINE_CONFIG",
             parseAndReturnElement(config))
```

```
        call.invoke(params);
...
```

Providing Dynamic Configuration for a Dynamic Proxy Web Service Client Example 3–2 displays a management configuration being passed to a dynamic proxy. The `setProperty` statement is highlighted in bold. Note the presence of the `<stub-property>` element in the example. You can use this element to pass information such as an endpoint address, a user name and password, and any other standard and proprietary properties.

Example 3–2 Management Configuration Dynamically Passed to a Dynamic Proxy

```
...
String config =
      "<port-info>" +

         "<stub-property>" +
                 "<name>...</name>" +
                 "<value>...</value>" +
         "</stub-property>" +
         "<runtime>" +
            "<security>" + ... + "<security>" +
            "<reliability>" +...+"</reliability> +
         "</runtime>" +
         "<operations>" +
            "<operation name='echo'>" +
               "<runtime>" +
                  "<security>" + ... + "<security>" +
                  "<reliability>" +...+"</reliability>" +
                  "<auditing>" +...+ "</auditing>" +
               "</runtime>" +
            "</operation>" +
         "</operations>" +
      "</port-info>";
((Stub)port)._setProperty(
        "oracle.webservices.ClientConstants.WSM_INTERCEPTOR_PIPELINE_CONFIG",
        parseAndReturnElement(config));
port.echo("Hello");
...
```

Providing Dynamic Configuration for a Static Proxy Web Service Client Example 3–3 displays a management configuration being passed to a static proxy. The `setProperty` statement is highlighted in bold. Note the presence of the `<stub-property>` element in the example. You can use this element to pass information such as an endpoint address, a user name and password, and any other standard and proprietary properties.

Example 3–3 Properties and Management Configuration Dynamically Passed to a Static Proxy

```
...
String config =
      "<port-info>" +
         "<stub-property>" +
                 "<name>...</name>" +
                 "<value>...</value>" +
         "</stub-property>" +
         "<runtime>" +
              "<security>" + ... + "<security>" +
```

```
                "<reliability>" +...+"</reliability> +
            "</runtime>" +
            "<operations>" +
               "<operation name='echo'>" +
                  "<runtime>" +
                     "<security>" + ... + "<security>" +
                     "<reliability>"+...+ "</reliability>" +
                     "<auditing>" +...+ "</auditing>" +
                  "</runtime>" +
               "</operation>" +
            "</operations>" +
         "</port-info>";
port = service.getTestServicePort();
((Stub)port)._setProperty(
     "oracle.webservices.ClientConstants.WSM_INTERCEPTOR_PIPELINE_CONFIG",
     parseAndReturnElement(config));
port.echo("Hello");
..."
```

Providing Dynamic Configuration for a J2EE Web Service Client Example 3–4 displays a management configuration being passed to a J2EE Web service client. The `setProperty` statement is highlighted in bold.

Example 3–4 Management Configuration Dynamically Passed to a J2EE Web Service Client

```
...
String config =
      "<port-info>" +
         "<runtime>" +
            "<security>" + ... + "</security>" +
            "<reliability>" +...+"</reliability>" +
         "</runtime>" +
         "<operations>" +
            "<operation name='echo'>" +
               "<runtime>" +
                  "<security>" + ... + "<security>" +
                  "<reliability>"+...+ "</reliability>" +
                  "<auditing>" +...+ "</auditing>" +
               "</runtime>" +
            "</operation>" +
         "</operations>" +
      "</port-info>";
Context ic = new InitialContext();
Service service = (Service)ic.lookup(
      "java:comp/env/service/MyTestServiceRef");
TestInterface port = (TestInterface)service.getPort(
      portQName, j2ee.client.TestInterface.class );
((Stub)port)._setProperty(
  "oracle.webservices.ClientConstants.WSM_INTERCEPTOR_PIPELINE_CONFIG",
      parseAndReturnElement(config));
port.echo("Hello");
...
```

Static Client-Side Configuration

A static client-side configuration is supported only for J2SE and J2EE Web service clients. It is not supported for DII or dynamic proxies.

Static proxies are supported and are described in "Providing Dynamic Configuration for a Static Proxy Web Service Client" on page 3-11.

The Enterprise Web Services 1.1 specification defines a deployment descriptor for each J2EE component type. The deployment descriptor contains information that allows the component to access a Web service endpoint.

Each standard J2EE Web services deployment descriptor has a corresponding Oracle-proprietary deployment descriptor. The purpose of an Oracle-proprietary deployment descriptor is to provide deployment-specific configuration information. While the application developer might specify suitable defaults, the Oracle-proprietary deployment descriptors allow the deployer to change deployment settings without affecting the application. The names of the Oracle-proprietary deployment descriptors are identical to the standard descriptors, except they have an `orion-` prefix. Table 3–1 lists the standard deployment descriptor and Oracle-proprietary deployment descriptor for each J2EE component.

Table 3–1 J2EE Components and their Corresponding Deployment Descriptors

J2EE Component	Standard J2EE Web Services Deployment Descriptor	Oracle-Proprietary Deployment Descriptor
JSP or Servlet	web.xml	orion-web.xml
EJB	ejb-jar.xml	orion-ejb-jar.xml
application client	application-client.xml	orion-application-client.xml

The standard deployment descriptors contain a `<service-ref>` element. This element captures all of the Web service access information, such as the location of the WSDL and mapping file, the service interface, the service ports, and their related service endpoint interfaces. The `<service-ref>` element is described in "Adding J2EE Web Service Client Information" to Deployment Descriptors in the *Oracle Application Server Web Services Developer's Guide*.

One of the elements within `<service-ref>` is `<service-ref-name>`. The value of the `<service-ref-name>` element in the standard descriptor maps to the value of the `name` attribute in the `<service-ref-mapping>` element in the proprietary (that is, the `orion-*`) descriptor. This reference provides the JNDI path and service name assigned by the client.

The Oracle-proprietary deployment descriptors contain structures that use the value of the `<service-ref-mapping>` element to map to its corresponding standard deployment descriptor. Within this element is a `<port-info>` element. This element provides all of the information, including Web service management information, for a port within a service reference. The structure and content of the `<port-info>` element is the same for every type of client. For more information on the `<service-ref-mapping>` element, see "Adding OC4J-Specific Platform Information" in the *Oracle Application Server Web Services Developer's Guide*.

The following sections describe how the standard and Oracle-proprietary deployment descriptors are used to provide static configuration information to J2EE Web service clients.

- Providing Static Configuration for a Servlet or JSP Web Service Client
- Providing Static Configuration for an EJB Web Service Client
- Providing Static Configuration for an Application Client Web Service Client

Providing Static Configuration for a Servlet or JSP Web Service Client Example 3–5 displays a fragment of a `web.xml` deployment descriptor for a Servlet or JSP Web service client. The `<service-ref-name>` element contains the reference to the static configuration for the client.

Example 3–6 displays the corresponding `orion-web.xml` file. The `<service-ref-mapping name="...">` element also contains the reference. The `<port-info>` element within `<service-ref-mapping>` contains the Web service management information.

The values for the `<service-ref-name>` and `<service-ref-mapping>` elements must match. These elements are highlighted in bold.

Example 3–5 web.xml—Static Configuration for a Servlet or JSP Web Service Client

```
<web-app>
   ...
   <service-ref>
      <service-ref-name>service/MyTestServiceRef</service-ref-name>
      ...
   </service-ref>
   ...
</web-app>
```

Example 3–6 orion-web.xml—Static Configuration and Management Information for a Servlet or JSP Web Service Client

```
<orion-web-app ...>
    <service-ref-mapping name="service/MyTestServiceRef">
        <port-info>
            ...
            <runtime>
                ...
                <security/>
                <reliability/>
                ...
            </runtime>
            <operations>
                ...
                  <operation name="echo">
                    <runtime>
                        ...
                        <security/>
                        <reliability/>
                        <auditing/>
                        ...
                    </runtime>
                  </operation>
                ...
            </operations>
        </port-info>
    </service-ref-mapping>
</orion-web-app>
```

Providing Static Configuration for an EJB Web Service Client Example 3–7 displays a fragment of an `ejb-jar.xml` deployment descriptor for an EJB Web service client. The `<service-ref-name>` element contains the reference to the static configuration for the client.

Example 3–8 displays the corresponding `orion-ejb-jar.xml` file. The `<service-ref-mapping name="...">` element also contains the reference. The `<port-info>` element within `<service-ref-mapping>` contains the Web service management information.

The values for the `<service-ref-name>` and `<service-ref-mapping>` elements must match. These elements are highlighted in bold.

Example 3–7 ejb-jar.xml—Static Configuration for an EJB Web Service Client

```
<ejb-jar ...>
   ...
   <enterprise-beans>
      <session>
       <ejb-name>EjbServiceConsumer</ejb-name>
          ...
          <service-ref>
             <service-ref-name>service/MyTestServiceRef</service-ref-name>
             ...
          </service-ref>
      </session>
   </enterprise-beans>
</ejb-jar>
```

Example 3–8 orion-ejb-jar.xml—Static Configuration and Management Information for an EJB Web Service Client

```
<orion-ejb-jar ...>
    <enterprise-beans>
        ...
        <session-deployment name="EjbServiceConsumer">
            <service-ref-mapping name="service/MyTestServiceRef">
                <port-info>
                    ...
                    <runtime>
                        <security/>
                    </runtime>
                        <operations>
                            ...
                            <operation name="echo">
                                <runtime>
                                    ...
                                    <security/>
                                    ...
                                </runtime>
                            </operation>
                        </operations>
                </port-info>
            </service-ref-mapping>
        </session-deployment>
    </enterprise-beans>
</orion-ejb-jar>
```

Providing Static Configuration for an Application Client Web Service Client Example 3–9 displays a fragment of an `application-client.xml` deployment descriptor for an application client Web service client. The `<service-ref-name>` element contains the reference to the static configuration for the client.

Example 3–10 displays the corresponding `orion-application-client.xml` file. The `<service-ref-mapping name="...">` element also contains the reference.

The <port-info> element within <service-ref-mapping> contains the Web service management information.

The values for the <service-ref-name> and <service-ref-mapping> elements must match. These elements are highlighted in bold.

Example 3–9 application-client.xml—Static Configuration for an Application Client Web Service Client

```
<application-client>
   ...
   <service-ref>
      <service-ref-name>service/MyTestServiceRef</service-ref-name>
      ...
   </service-ref>
</application-client>
```

Example 3–10 orion-application-client.xml—Static Configuration and Management Information for an Application Client Web Service Client

```
<orion-application-client>
    <service-ref-mapping name="service/MyTestServiceRef">
        <port-info>
            ...
            <runtime>
                <security/>
            </runtime>
            <operations>
               <operation name="echo">
                   <runtime>
                       <security/>
                   </runtime>
               </operation>
            </operations>
        </port-info>
    </service-ref-mapping>
</orion-application-client>
```

Application Server Control Support for Web Service Management

The following sections briefly describe the functionality for managing a Web service that is available by using Application Server Control.

Configuring, Enabling, and Disabling Web Service Management Features

You can use the Application Server Control to manage the Web services you deploy. For example, you can deploy your Web service applications, perform configuration tasks, and monitor your Web service ports.

In addition, you can perform administration tasks associated with standard Web Service Management features, which include auditing, logging, reliability, and security. Web Service Management is a set of policies applied against a message en-route to a target. In the case of OracleAS Web Services, the messages are SOAP requests, responses, and faults, and the targets are the client and the business logic on the server.

For instructions on how to configure, enable, and disable Web service management features, see the topics *Overview of Web Services Management* and *Enabling and Disabling Web Service Management Features* in the Application Server Control on-line help.

Enabling and Disabling a Web Service

You can use Application Server Control to enable a disabled Web service or disable an enabled Web service for an OC4J instance.

For instructions on how to enable and disable a Web service, see the topic *Enabling and Disabling a Web Service* in the Application Server Control on-line help.

Configuring Auditing for a Web Service

The Auditing feature of Web Service Management enables you to keep a complete, persistent record of SOAP requests, responses, and faults. During development it is often very convenient to be able to inspect the contents of SOAP requests and responses to diagnose problems.

Auditing captures request, response, and fault messages and stores them in the following persistent file.

```
ORACLE_HOME\log\wsmgmt\audit\log.xml
```

For instructions on how to configure auditing, see the topic *Configuring Auditing for a Web Service* in the Application Server Control on-line help.

Configuring Logging for a Web Service

When you configure logging for a Web service, you can identify the attributes you want logged from incoming, outgoing, and fault SOAP messages.

For instructions on how to configure logging for a Web service, see the topic *Configuring Logging for a Web Service* in the Application Server Control on-line help.

Configuring Reliability for a Web Service

You can use Application Server Control to configure your Web service with reliable messaging. Reliable messaging allows users of the Web services stack to exchange SOAP messages without duplicates and with guaranteed delivery and message ordering.

For instructions on how to configure reliability for a Web service, see the topic *Configuring Reliability for a Web Service* in the Application Server Control on-line help.

Configuring Security for a Web Service

You can use Application Server Control to configure your Web service with authentication, integrity with digital signatures, and confidentiality with encryption based on the WS-Security standard.

For instructions on how to configure security for a Web service, see the topic *Configuring Security for a Web Service* in the Application Server Control on-line help.

Viewing the WSDL for a Web Service

You can use Application Server Control to view the WSDL for a Web service. You cannot edit the contents of the `.wsdl` file, but you can review the contents to verify the attributes and characteristics of the Web service.

For instructions on how to view the WSDL, see the topic *Specifying an XPath When Configuring Web Services Logging* in the Application Server Control on-line help

Testing a Web Service

You can use Application Server Control to test a Web service. When you test the Web service, you display its Home Page. The Web Services Home Page URL is also the service endpoint that the Web service exposes to its clients. From this URL you can

invoke the operations for values that you enter and verify that the Web service is responding appropriately.

For instructions on how to use Application Server Control to test a Web service, see the topic *Testing a Web Service* in the Application Server Control on-line help.

Viewing Web Service Operations

You can use Application Server Control to view Web service operations and operation metrics.

For instructions on how to view Web service operations and operation metrics, see the topic *Viewing Web Service Operations* in the Application Server Control on-line help.

Working with Capability Assertions

Capability assertions are descriptions of Web service management policies, such as security and reliability. They allow consumers of Web services to discover which management policies are enabled for the Web service.

For example, assume that you created a Web service that requires a security token, such as a user name and password. The client will not be able to access the service unless it is aware of this requirement. Capability assertions provide hints to the client that a user name and password must be placed in the message SOAP headers at runtime.

Capability assertions are derived from the server-side Web service management configuration in the `oracle-webservices.xml` deployment descriptor and generated into the WSDL. The generation can be performed either by JDeveloper wizards or WebServicesAssembler commands. In JDeveloper, the Web service management configuration wizard contains a "Capability Assertions" option. In WebServicesAssembler, you provide a WSDL, set the `genQos` argument to `true`, and set the `ddFileName` argument to an `oracle-webservices.xml` deployment descriptor that contains the server-side management configuration.

For the generation of an OracleAS Web Services client, a WSDL and configuration file are used as input. The configuration file describes some of the built-in behavior of the generated client and is based on the `oracle-webservices-client-10_0.xsd` schema. The configuration file can be written by hand or created by a tool, such as JDeveloper. The capability assertions from the WSDL are used as hints to help in the creation of the client-side configuration.

For a J2SE client, the client-side configuration file is automatically packaged with the proxy classes when they are generated. For a J2EE client, the client-side configuration is set up at deployment time.

How to Assemble Capability Assertions into a Web Service

Continuing the example described earlier in this section, the capability assertions generated into the WSDL will state that a user name and password are required to access the service. If you use JDeveloper to generate the client, it can take this information and prompt you for a user name and password. JDeveloper will create a client-side configuration file that will be used at runtime to place this user name and password into the message SOAP headers.

If you are using WebServicesAssembler instead of JDeveloper, you will have to understand the capability assertions in the WSDL and author the client-side configuration accordingly. You must then either pass the configuration to

WebServicesAssembler with the `ddFileName` argument for packaging with the client proxy or pass it to the call/proxy at runtime.

The following generalized steps summarize the process of including capability assertions in your Web service:

1. Create the server-side Web service management configuration for security, reliability, auditing, and logging by hand or by using JDeveloper.
   - **By Hand:** enter the management configuration into the `oracle-webservices.xml` deployment descriptor by hand.
   - **JDeveloper:** use the JDeveloper wizards to choose the Web service management options. The wizards will enter this information into the `oracle-webservices.xml` deployment descriptor.
2. Generate the capability assertions into the WSDL by using either WebServicesAssembler or JDeveloper.
   - **WebServicesAssembler:** use the `genQosWsdl` command to produce a WSDL with capability assertions.
   - **JDeveloper:** select the appropriate options in the wizards to generate capability assertions into the WSDL.
3. Generate the client code by using either WebServicesAssembler or JDeveloper.
   - **WebServicesAssembler:** supply a WSDL and client-side configuration file to the `genProxy` command (for a J2SE client) or `genInterface` command (for a J2EE client); use the `ddFileName` argument to specify the client-side configuration file.
   - **JDeveloper:** use the appropriate JDeveloper wizard to generate either J2SE or J2EE client code.
4. Run the client code:
   - for a J2SE client, compile and run the client
   - for a J2EE client, deploy, then run the client

Additional Information

For more information on:

- adding security features to a Web service, see Chapter 4, "Ensuring Web Services Security" and the *Oracle Application Server Web Services Security Guide*.
- adding reliable messaging to a Web service, see Chapter 5, "Ensuring Web Service Reliability".
- adding logging and auditing to a Web service, see Chapter 6, "Auditing and Logging Messages".
- assembling a Web service top down, see "Assembling a Web Service from a WSDL" in the *Oracle Application Server Web Services Developer's Guide*.
- using Java classes to assemble a Web service, see "Assembling a Web Service with Java Classes" in the *Oracle Application Server Web Services Developer's Guide*.
- using EJBs to assemble a Web service, see "Assembling a Web Service with EJBs" in the *Oracle Application Server Web Services Developer's Guide*.

- using JMS topics and destinations to assemble a Web service, see "Assembling Web Services with JMS Destinations" in the *Oracle Application Server Web Services Developer's Guide*.
- using database resources, such as PL/SQL packages, SQL queries, DML statements, Oracle Streams AQ, or server-side Java classes, to assemble a Web service, see "Assembling Database Web Services" in the *Oracle Application Server Web Services Developer's Guide*.
- assembling a J2EE Web Service client, see "Assembling a J2EE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.
- assembling a J2SE Web service client, see "Assembling a J2SE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.
- packaging and deploying of Web services, see "Packaging and Deploying Web Services" in the *Oracle Application Server Web Services Developer's Guide*.
- using WebServicesAssembler commands to assemble Web service artifacts, see "Using WebServicesAssembler" in the *Oracle Application Server Web Services Developer's Guide*.
- using JDeveloper to add management information to a Web service, see the JDeveloper on-line help.
- using Application Server Control to add management information to a Web service, see the Application Server Control on-line help.

4 Ensuring Web Services Security

Web services security is described in the *Oracle Application Server Web Services Security Guide* The *Security Guide* contains the following chapters:

- Chapter 1, "Introduction"

  This chapter introduces essential Web service security concepts, standards, and specifications. It is divided into the following sections:

  – Web Service Security Concepts

  – Web Services Security Support in OC4J

  – Tool Support for Web Services Security

- Chapter 2, "Configuring Web Services Security"

  This chapter describes the Web service security configuration elements that can be used to secure a Web service on the client and the server. It is divided into the following sections:

  – Keystore Elements

  – Signature and Encryption Key Elements

  – Nonce Configuration Elements

  – Security Elements for Inbound Messages

  – Security Elements for Outbound Messages

- Chapter 3, "Administering Web Services Security"

  This chapter describes administration tasks for Web services security. It is divided into the following sections:

  – Using Keystores

  – Integrating Security Tokens with Security Providers

  – Using a Username Token

  – Using an X.509 Token

  – Using a SAML Token

  – Configuring XML Encryption

  – Configuring XML Signature

  – Combining Tokens, Encryption. and Signature

- Chapter 4, "Building Secure Web Services"

This chapter provides the generalized steps for assembling a secure Web service. Oracle Application Server Web Services provides the WebServicesAssembler tool which enables you to assemble the service top down (from a WSDL) or bottom up (from Java classes, EJBs, or database resources).

  – Assembling a Secure Web Service
  – Creating a Server-Side Security Configuration File
  – Creating a Client-Side Security Configuration File
  – Client JAR Files
  – Adding Transport-Level Security to a Web Service
  – Ant Tasks and WebServicesAssembler
  – Getting an Authenticated User Identity in a Web Service Application
  – Performing JAAS Provider Authorization on a Web Service
  – WS-Security and XML APIs
  – Development Decisions

- Chapter 5, "Secure Web Service Usage Scenarios"

  This chapter discusses common scenarios for using Web service security. It begins with the simplest use case, then proceeds through increasingly more complex use cases. The first section of the chapter discusses use cases with no security implications; these are then modified to add security features. It contains the following sections:

  – Non-Secured Web Services
  – HTTP-Based Security
  – WS-Security
  – XML Signature
  – XML Encryption
  – Gateways
  – Identity Management
  – Interoperability

- Chapter 6, "Troubleshooting"

  This chapter describes solutions to some of the errors you might encounter when working with OracleAS Web Services Security. The errors are divided into these categories.

  – General Errors
  – Keystore-Related Errors
  – Message Integrity Errors
  – Message Confidentiality Errors
  – Authentication Errors

- Appendix A, "Schemas"

  This appendix lists the contents of the OracleAS Web Services Security schema file, `oracle-webservices-security-10_0.xsd`.

- Appendix B, "Security Threats and Solutions"

  This appendix describes how the functionality in OracleAS Web Services can be used to address the threats to security that are present in today's Web environment.

Additional Information

For more information on:

- securing a Web service, see the *Oracle Application Server Web Services Security Guide*
- adding Web service management information, including security, to a Web service, see Chapter 3, "Managing Web Services".
- adding reliability information to a Web service, see Chapter 5, "Ensuring Web Service Reliability".
- adding auditing and logging information to a Web service, see Chapter 6, "Auditing and Logging Messages".
- assembling a Web service top down, see "Assembling a Web Service from a WSDL" in the *Oracle Application Server Web Services Developer's Guide*.
- using Java classes to assemble a Web service, see "Assembling a Web Service with Java Classes" in the *Oracle Application Server Web Services Developer's Guide*.
- using EJBs to assemble a Web service, see "Assembling a Web Service with EJBs" in the *Oracle Application Server Web Services Developer's Guide*.
- using JMS topics and destinations to assemble a Web service, see "Assembling Web Services with JMS Destinations" in the *Oracle Application Server Web Services Developer's Guide*.
- using database resources, such as PL/SQL packages, SQL queries, DML statements, Oracle Streams AQ, or server-side Java classes, to assemble a Web service, see "Assembling Database Web Services" in the *Oracle Application Server Web Services Developer's Guide*.
- assembling a J2EE Web Service client, see "Assembling a J2EE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.
- assembling a J2SE Web service client, see "Assembling a J2SE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.
- using WebServicesAssembler commands to assemble Web service artifacts, see "Using WebServicesAssembler" in the *Oracle Application Server Web Services Developer's Guide*.
- the contents of the `oracle-webservices.xml` deployment descriptor which contains the Web services management configuration, see "Packaging and Deploying Web Services", in the *Oracle Application Server Web Services Developer's Guide*.
- the contents of the `wsmgmt.xml` file which contains the security configuration, see Appendix A, "Understanding the Web Services Management Schema".

5

Ensuring Web Service Reliability

Message exchange in Web services can be inherently unreliable. It is not well-suited for long-running conversations in a reliable fashion without additional infrastructure. To use Web services for mission-critical, more complex business processes, you must be able to exchange messages reliably. To this end, Oracle Application Server Web Services provides the ability to add quality of service (QOS) guarantees around reliable messaging. The facility provided is that of the OASIS standard. You can find the specification at the following Web site:

http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm

WS-Reliability is a specification for open, reliable Web service messaging including guaranteed delivery, duplicate message elimination and message ordering, enabling reliable communication between Web services. The reliability features are based on extensions to the Simple Object Access Protocol (SOAP), rather than being tied to the underlying transport protocol. The specification allows a variety of systems to interoperate reliably in a platform- and vendor-neutral manner.

Reliability allows users of the Web services stack to exchange SOAP messages without duplicates and with guaranteed delivery and message ordering. The information that allows messages to be delivered in a reliable fashion is contained in extensions to the SOAP message headers. The SOAP header extensions, the details of the messaging model, and the semantics associated with it are defined by the WS-Reliability specification.

Reliability encompasses the following features.

- guaranteed delivery—ensures that a sent message will be received.
- duplication elimination—ensures that the receiver will not see any duplicate messages (messages with the same message ID are treated as duplicates).
- guaranteed message delivery and duplicate message elimination—ensures that all messages sent will be received, and any duplicate messages will be removed.
- message ordering—ensures that a sequence of messages that are sent will be received in the same order.

Enabling reliability on the server enables all of these features. The exercising of the features is driven by the client either through a configuration file or an API.

This chapter contains the following sections.

- Setting Up Reliability
- Managing Reliability on the Server
- Managing Reliability on the Client
- Configuring Client-Side Database Support

- Dynamically Configuring Client-Side Reliability
- Tool Support for Web Services Reliability

Setting Up Reliability

The following sections describe the requirements you must satisfy before you can use reliable messaging between the server and client.

- Providing a Running Database
- Installing SQL Tables for the Client and Server

Providing a Running Database

You can configure reliable messaging such that messages will be stored in the server-side memory store or in a database store. If you choose to use a database store, then a database must be running on the server before the reliable service is deployed.

Installing SQL Tables for the Client and Server

To enable reliable messaging, you must install certain SQL tables on both the client and the server. The SQL scripts that install these tables are stored in the `j2ee/home/database/webservices/reliability/oracle` directory.

- `clear-tables.sql`—can be used to delete all records from the server-side reliability tables. This script is provided as a convenience and should be used only by the database administrator if the tables are in an irreversibly inconsistent state.
- `drop-tables.sql`—can be used to drop all of the client and server reliability tables.
- `reliability-tables.sql`—creates reliability tables on both the client and server. The tables required on the server are `RM_SERVER_MESSAGE_INFO` and `RM_SERVER_GROUP_INFO`. The tables required on the client are `RM_CLIENT_MESSAGE_INFO` and `RM_CLIENT_GROUP_INFO`. If you are developing only a client or a server, then only its respective table has to be installed.

The scripts can be executed on the command line with the following SQL*Plus command.

`Sqlplus` *username*/*password* @*sql_script*

In this command, *username* and *password* are your user name and password for executing SQL scripts; *sql_script* is the name of the SQL script.

Changing the Widths of Database Columns

By default, database column widths for fields such as `GROUP_ID`, `REPLY_TO_URL`, `URL_PATTERN`, and `ENDPOINT_ADDRESS` are set to 150. However, some applications might generate longer names. This is likely to happen if you are working with unusually long URLs or host names. If a generated name is longer than the specified column width, then a database error can be returned.

To avoid this problem, you can modify the `reliability-tables.sql` script to increase the column widths of the server-side and client-side tables. Example 5–1 illustrates a modified `reliability-tables.sql` script where the column width of the `GROUP_ID`, `REPLY_TO_URL`, `URL_PATTERN` and `ENDPOINT_ADDRESS` fields have been increased from 150 to 250.

Example 5–1 reliability-tables.sql with Modified Column Widths

```
CREATE TABLE RM_SERVER_MESSAGE_INFO (
GROUP_ID varchar(250),
SEQUENCE_NUM number,
TIMESTAMP number,
EXPIRY_TIME number,
RETRY_COUNT number,
ACK_STATUS varchar(50),
PROCESS_STATUS varchar(50),
REPLY_TO_URL varchar(250),
ENDPOINT_ADDRESS varchar(250),
URL_PATTERN varchar(250),
REPLY_PATTERN varchar(150),
FAULT_CODE varchar(50),
LAST_UPDATE number,
CONTENT_TYPE varchar(150),
SOAP_ACTION varchar(150),
MESSAGE BLOB,
PIPELINE_CONFIG BLOB,
CONSTRAINT RM_SERVER_MESSAGE_INFO_PK PRIMARY KEY (GROUP_ID, SEQUENCE_NUM));
CREATE TABLE RM_SERVER_GROUP_INFO (
GROUP_ID varchar(250) PRIMARY KEY,
SEQUENCE_NUM number ,
LAST_UPDATE number ,
TIMESTAMP number ,
GROUP_EXPIRY_TIME number ,
GROUP_MAX_IDLE number ,
REPLY_TO_URL varchar(250),
ENDPOINT_ADDRESS varchar(250));
```

Adding Reliable Messaging to a Web Service

To enable reliable messaging, you have to configure it into the Web service on the server side and into the client. The following steps provide a general outline and a "quick start" of how you can enable reliable messaging. For more detailed description, see "WebServicesAssembler Support for Web Service Reliability" on page 5-16.

Server side:

On the server-side there are two ways to enable reliable messaging in a Web service. One way is to provide a reliability configuration in a deployment descriptor and assemble it into the Web service. The sections "Assembling Reliability into a Web Service Bottom Up" on page 5-16 and "Assembling Reliability into a Web Service Top Down" on page 5-17 describes this technique.

The second technique is to use the Application Server Control tool to configure reliable messaging for the Web service after it is deployed. This technique is described in the following steps.

1. Deploy the Web service.

   Web services are deployed in the standard manner into a running instance of OC4J. For more information on deploying Web service EAR files, see the *Oracle Containers for J2EE Deployment Guide*.

2. If you have access to the Application Server Control tool, you can use it to configure reliable messaging on the server-side. For more information, see the topic "Configuring Reliability for a Web Service" in the Application Server Control on-line help.

a. Log in to the Application Server Control tool.

b. Click the Web services tab.

c. Select the name of the Web service that was deployed.

d. Click the Administration tab.

e. Click Enable/Disable features.

f. Move "Reliability" to the enabled column and click "OK".

The Web service will have reliable messaging enabled.

Client Side:

To enable reliable messaging on the client, assemble a reliability configuration into the client proxy. The settings that you choose for the reliability elements in the client side must agree with the reliability settings for the server.

1. Create a client deployment descriptor file that includes the configuration elements for reliability.

   "Client-Side Reliability Configuration Elements" on page 5-9 describes the reliability elements that can be included in the client-side configuration file. The configuration file describes some of the built-in behavior of the generated client and is based on the `oracle-webservices-client-10_0.xsd` schema. If there are certain quality of service properties marked as "required" (such as duplicate elimination or guaranteed delivery), then the client must also specify these properties.

   The configuration file can be written by hand or created by a tool, such as JDeveloper.

2. Assemble the client proxy using the WebServicesAssembler command line or ant-tasks.

   Use the `genProxy` command to generate the Web service client. Specify the name of the WSDL with the `wsdl` argument and the name of the file that contains the reliability configuration with the `ddFileName` argument. "Assembling Reliability into a J2SE Web Service Client Proxy" on page 5-18 provides more information on assembling reliability into Web service clients.

The client stubs will have reliability enabled and are ready to be used by client code.

Managing Reliability on the Server

This section describes the configuration elements for reliability that can be found in the server-side `oracle-webservices.xml` file. For a description of how you can add management information, including reliability, to `oracle-webservices.xml`, see "Configuring Server-Side Management Information" on page 3-4.

As with other files that are to be deployed, `oracle-webservices.xml` is typically stored in an EAR file. Once the EAR is deployed to OracleAS Web Services, the Web services management information in `oracle-webservices.xml` is extracted and stored in `wsmgmt.xml`. You can modify the management configuration post-deployment by using the Application Server Control tool.

Server-Side Reliability Configuration Elements

Reliability features can be assigned at the port level and at the operation level.

- Port-Level Reliability Elements on the Server

- Operation Level Reliability Elements on the Server

Port-Level Reliability Elements on the Server

This section describes the reliability elements that can be set at the port level. The port level configuration determines where messages are stored.

The value of the `type` attribute in the `<repository>` element determines whether the server will store messages in-memory or in a database store. If `type="memory"` then messages are stored in an in-memory store. If `type="jdbc"` then messages are stored in a database store.

Table 5–1 provides more information on `<repository>` and other port-level reliability elements that can appear in the `oracle-webservices.xml` file.

Table 5–1 Port-Level Reliability Elements in the Server-Side Configuration File

Element Name	Description
<ack-interval>	Specifies the interval, in seconds, after which an attempt is made to send acknowledgments or faults for messages that were processing asynchronously. The default value is 60 seconds.
<ack-limit>	The `value` attribute specifies a limit on the number of attempts that will be made to acknowledge a message. The default value is -1. This indicates no-limit, or an infinite number of acknowledgment attempts. A value of 0 indicates no acknowledgment attempts.
<cleanup-interval>	The `value` attribute specifies the interval, in seconds, after which an attempt is made to purge the store of expired or invalid messages. The default is 3600 seconds (60 minutes).
<max-age>	The `value` attribute specifies the maximum time the user would want the message to be stored. The default value is 86400 seconds.
<order-interval>	The `value` attribute specifies the interval, in seconds, after which the unordered messages are processed. The default is 60 seconds.
<repository>	Describes the kind of store that the server will use to store messages. The value of the `type` attribute will determine the type of store being configured. The allowed values for `type` are `memory` for a memory store and `jdbc` for a database store. The default is `memory`. ■ For a memory store (`memory`), `name` is a required attribute. If the store with that name has been configured for another application in the server, then that store will be used. Otherwise a new store will be configured. Example 5–2 illustrates a port-level reliability configuration that stores messages in an in-memory store. ■ For a database store (`jdbc`), a datasource must be configured on the server and its location specified by `jndiLocation`. Example 5–3 illustrates a port-level reliability configuration that stores messages in a database store.

Example 5–2 lists the reliability elements as they are used in `oracle-webservices.xml` to describe reliability for an in-memory store.

Example 5–2 Port-Level Reliability Set for an In-Memory Store

```
...
<port-component name="...">
   <runtime enabled="reliability">
      <reliability>
        <repository name="bank-server-store" type="memory"/>
        <ack-interval value="60"/>
```

```
            <order-interval value="60"/>
            <cleanup-interval value="3600"/>
            <ack-limit value="10"/>
            <max-age value="86400"/>
          </reliability>
       </runtime>
       <operations>
       ...
    </port-component>
```

Example 5–3 lists the reliability elements for a configured datasource on the server.

Example 5–3 Port Level Reliability Set for a Configured Datasource on the Server

```
...
<port-component name="...">
   <runtime enabled="reliability">
    <reliability>
      <repository type="jdbc" jndiLocation="jdbc/OracleManagedDS"/>
      <ack-interval value="60"/>
      <order-interval value="60"/>
      <cleanup-interval value="3600"/>
      <ack-limit value="10"/>
      <max-age value="86400"/>
    </reliability>
   </runtime>
   <operations>
   ...
</port-component>
```

Operation Level Reliability Elements on the Server

This section describes the reliability elements that can be set at the operation level. Example 5–4 lists the reliability elements as they are used in `oracle-webservices.xml`.

Note that since message ordering can take place across operations, it cannot be set at this level. It also cannot be set at the port level because you might not want all operations to be reliable or sensitive to message ordering.

Example 5–4 Operation-Level Reliability Elements in the Server-Side Configuration File

```
...
<port-component name="...">
  ...
         <operations>
            <operation name="deposit">
               <runtime>
                  <reliability>
                     <duplication-elimination-required value="true"/>
                     <guaranteed-delivery-required value="true"/>
                  </reliability>
               </runtime>
            </operation>
         </operations>
      </port-component>
...
```

Table 5–2 describes the reliability assertions that you can set at the operation level in the `oracle-webservices.xml` file. If the values of these assertions are `true`, but

the client does not send the message according to the requirement, then a SOAP fault is returned, describing the assertion that was violated.

Table 5–2 Operation-Level Reliability Elements in the Server-Side Configuration File

Element Name	Description
<duplication-elimination-required>	If set to `true`, it indicates that non-reliable clients will not be allowed to invoke this operation. Specifically messages that do not have the SOAP header for duplicate elimination will be rejected. The default is `false`.
<guaranteed-delivery-required>	If set to `true`, it indicates that non-reliable clients will not be allowed to invoke this operation. Specifically, messages that do not have the SOAP header for guaranteed delivery will be rejected. The default is `false`.

Capability Assertions and Reliability

On the server side, capability assertions can specify the reliable capabilities of the Web service. For reliability, capability assertions restrict the kind of messages sent to the endpoint.

Capability assertions for reliability can be added to the Web service either by JDeveloper wizards or WebServicesAssembler commands. In JDeveloper, the Web service management configuration wizard contains a "Capability Assertions" option.

In WebServicesAssembler, when you are assembling a Web service top down, use the `genQosWsdl` command to add capability assertions to the WSDL. Use the following arguments to the `genQosWsdl` command.

- use the `wsdl` argument to specify a WSDL
- set the `genQos` argument to `true`
- set the `ddFileName` argument to an `oracle-webservices.xml` deployment descriptor that contains the server-side reliability configuration elements

When you are assembling a Web service bottom up, use the appropriate `*Assemble` command to add capability assertions to the WSDL. Use the following arguments to the `*Assemble` command.

- set the `genQos` argument to `true` to indicate that capability assertions will be added to the WSDL
- set the `ddFileName` argument to an `oracle-webservices.xml` deployment descriptor that contains the server-side reliability configuration elements

Port-Level Reliability Elements on the Server on page 5-5 and "Operation Level Reliability Elements on the Server" on page 5-6 describe the elements that can be used in the `oracle-webservices.xml` deployment descriptor.

"Working with Capability Assertions" on page 3-18 provides an overview of working with capability assertions.

Example 5–3 describes the capability assertion elements defined for WS-Reliability.

Table 5–3 Capability Assertions Defined for WS-Reliability

Reliability Assertion	Description
`<guaranteed-delivery required="true">`	This assertion indicates that only messages that have a guaranteed delivery SOAP header element will be processed. Any messages that arrive at the reliable endpoint without a SOAP header indicating guaranteed delivery will be rejected with a SOAP fault.
`<reply-patterns callback="true" poll="true" response="true"/>`	This assertion indicates support for various reply patterns. These assertions are explicit advertisements of the capabilities of the endpoint. Clients can use these capabilities to determine how they want to interact with the endpoint.
`<duplicate-elimination required="true"/>`	This assertion indicates that only messages that have a duplicate elimination SOAP header element will be processed. Any messages that arrive at the reliable endpoint without the correct SOAP header in the message indicating duplicate elimination will be rejected with a SOAP fault.

Example 5–5 provides an excerpt from a generated WSDL with capability assertions tags generated into it. You can edit the generated values in the WSDL to change their affect. If you edit the WSDL, then you will have to regenerate the endpoint and any clients.

Example 5–5 WSDL Fragment, Containing Capability Assertions for Reliability

```
...
<binding name="HttpSoap11Binding" type="tns:Bank">
<soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="deposit">
<soap:operation soapAction="http://www.oracle.com/bank/deposit"/>
   <capability-assertions wsdl:required="true">
     <guaranteed-delivery required="true">
       <reply-patterns callback="true" poll="true" response="true"/>
     </guaranteed-delivery>
     <duplicate-elimination required="true"/>
   </capability-assertions>
<input>
<soap:body use="encoded" namespace="http://www.oracle.com/bank"
   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" parts="accountName
amount"/>
</input>
</operation>
</binding>
...
```

Managing Reliability on the Client

Clients can manage reliability by using either an API or a configuration file.

- Client-Side Reliability Configuration Elements
- Dynamically Configuring Client-Side Reliability

Client-Side Reliability Configuration Elements

Reliability must be configured for both the server side and the client side for reliable message communication to occur. This section describes only the client-side configuration elements. For information on the server-side elements, see "Server-Side Reliability Configuration Elements" on page 5-4.

On the client, reliability elements are stored in a configuration file. The name of the file and where it resides depends on whether the client is J2EE or J2SE.

- For J2EE servlet, EJB, and Web application clients, the configuration file is:
  - `META-INF/orion-ejb-jar.xml` for an EJB client
  - `WEB-INF/orion-web.xml` for a JSP or servlet client
  - `META-INF/orion-application-client.xml` for a Web application client
- For J2SE clients, the configuration file is: <*generated_name*>`_Stub.xml`, where *generated_name* is derived from the `targetNamespace` and `port` name in the WSDL.

Reliability can be assigned at the port level and at the operation level.

Port Level Reliability Elements on the Client

This section describes the reliability elements that can be set at the port level for the client. Example 5–6 lists the reliability elements as they are used in the client-side configuration file.

Example 5–6 Port-Level Reliability Elements in the Client-Side Configuration File

```
...
<port-info>
   <runtime enabled="reliability">
       <reliability>
           <repository name="standalone-client-store" type="memory"/>
           <retry-interval value="60"/>
           <cleanup-interval value="3600"/>
           <poll-interval value="60"/>
           <retry-limit value="10"/>
           <max-age value="86400"/>
           <standalone-listener-port value="8008"/>
       </reliability>
   </runtime>
...
</port-info>
```

Table 5–4 describes the reliability elements that can be set at the port level.

Table 5–4 Port-Level Reliability Elements in the Client-Side Configuration File

Element Name	Description
<async-poll-reply-to-url>	The `value` attribute specifies the URL to which acknowledgments and faults will be sent if asynchronous polling is desired. The URL for this is typically the host name of the client along with the port that the listener is on.
<cleanup-interval>	The `value` attribute specifies the interval, in seconds, after which an attempt is made to purge the store of expired or invalid messages. The default is 3600 seconds (60 minutes).

Table 5–4 (Cont.) Port-Level Reliability Elements in the Client-Side Configuration File

Element Name	Description
<max-age>	The `value` attribute specifies the maximum time the user would want the message to be stored. The default value is 86400 seconds.
<poll-interval>	The `value` attribute specifies the interval, in seconds, after which a poll message will be sent to the server for reliable messages that have not yet received any acknowledgment. Default value is 60 seconds.
<repository>	Specifies the kind of store that will be used to store messages. The `name` attribute is a `String` that specifies the name of the repository. The value of the `type` attribute will determine the type of store being configured. The allowed values for `type` are `memory` for a memory store and `jdbc` for a database store. The default type of store is a `memory` store. ■ If `memory` is specified, then `name` is a required attribute. If a store with the same name has been configured for another server application, then that store will be used. If it has not, then a new store will be configured. ■ If `jdbc` is specified, then you must provide database support on the client side. For a J2EE client, a data source must be configured and its location specified with `jndiLocation`. For information on configuring a client-side database, see "Configuring Client-Side Database Support" on page 5-12.
<retry-interval>	The `value` attribute specifies the interval after which an attempt will be made to retry sending of messages that have not yet received any acknowledgment or fault. The default value is 60 seconds.
<retry-limit>	The `value` attribute specifies a limit on the number of attempts that will be made to retry a message. The default value is 10. A value of -1 indicates no -limit, or infinite retries. **Note:** If the `value` attribute is set to 0, then the client will not attempt to re-send messages. This, in effect, negates the guaranteed delivery quality of service. As an optimization, the reliability stack does retain messages in the database if the `value` attribute is set to 0. No polling will be performed for these messages since no guarantee can be made that the message will reach the destination and an acknowledgment returned.
<standalone-listener-port>	The `value` attribute specifies the port at which a listener will be started for receiving acknowledgments or faults. This is used for J2SE clients where there is no infrastructure to which acknowledgments can be sent. If the client is in a J2EE application, then a supplied servlet can be used.

Operation Level Reliability Elements on the Client

This section describes the reliability elements that can be set at the operation level for the client. Example 5–7 lists the reliability elements as they are used in the client-side configuration file.

Example 5–7 Operation-Level Reliability Elements in the Client-Side Configuration File

```
<port-info>
...
   <operations>
      <operation name="...">
         <runtime>
```

```
            <reliability>
               <guaranteed-delivery enabled="true"/>
               <duplicate-elimination enabled="true"/>
               <group-expiry-time value="86400"/>
               <reply-pattern value="Callback"/>
               <reply-to-url value="http://localhost:8008"/>
               <expiry value="3600"/>
            </reliability>
         </runtime>
      </operation>
   </operations>
</port-info>
```

Table 5–5 describes the reliability elements that can be set on the operation level.

You should provide a value for either `<group-max-idle-time>` or `<group-expiry-time>`. If both elements are specified, then the value for `<group-expiry-time>` will take precedence.

Table 5–5 Operation-Level Reliability Elements in the Client-Side Configuration File

Element Name	Description
<duplicate-elimination>	If the `enabled` attribute is `true`, duplicate elimination of the message is enabled. Default value is `true`.
<expiry>	The `value` attribute specifies the expiry time for a single message in a group. For example, if a message, has not been acknowledged for a time duration longer than the expiration duration, then no further attempt will be made to deliver the message. The client will be notified of the failure. If a value is set for `<group-expiry-time>`, it should be greater than the value of `<expiry>`. Default value is 3600 seconds.
<group-expiry-time>	The `value` attribute specifies the expiration time, in seconds, for all messages in the group. The value for `<group-expiry-time>` should be greater than the value set for `<expiry>`. Default value is 84600 seconds.
<group-max-idle-time>	The `value` attribute specifies the maximum idle time, in seconds, for a group of messages. This element is not required to be present.
<guaranteed-delivery>	If the `enabled` attribute is `true`, guaranteed delivery of message is enabled. Default value is `true`.
<reply-pattern>	The `value` attribute can be either `Callback` (asynchronous acknowledgment/fault), `Response` (synchronous acknowledgment/fault) or `Polling` (the acknowledgment or fault will be polled for). Default value is `Polling`. The `<reply-pattern>` element must be set only if `<guaranteed-delivery>` is `true`.
<reply-to-url>	The `value` attribute specifies the URL to which acknowledgments and faults will be sent for messages that want asynchronous acknowledgments. The URL for this is typically the host name of the client along with the port that the listener is on. The `<reply-to-url>` element must be set only if `<guaranteed-delivery>` is `true`.

Configuring Client-Side Database Support

OracleAS Web Services lets you choose whether to store messages sent to the client in memory or in a database. Messages stored in memory are not persistent: if the server goes down, the messages are lost.

If you choose to store client messages in a database, you must provide the database and connection details to the client. This information cannot be passed to the client by using the client-side APIs. Instead, you must provide settings for the port-level `<repository>` element in the client-side management configuration file. This element is described in Table 5–4 on page 5-9.

Configuring Database Support for a J2SE Client

For a J2SE client, you must provide all of the parameters that are required to establish a database connection. This information, such as the name of the database, its URI, connection type, the connection driver, and the user name and password, are all attributes of the `repository` element. You enter this information into the <*generated_name*>`_Stub.xml` file, where *generated_name* is derived from the target namespace and port name in the WSDL.

Example 5–8 illustrates the configuration of a database connection for a J2SE client. Note that the J2SE client requires all of the `repository` attributes to connect to the database.

Example 5–8 Database Connection for a J2SE Client

```
<reliability>
    <repository
        name="bank-client-store"
        type="jdbc"
        connection-driver="org.hsqldb.jdbcDriver"
        username="SCOTT"
        password="TIGER"
        url="jdbc:hsqldb:testdb" />
    <retry-interval value="60" />
    <retry-limit value="10" />
    <poll-interval value="60" />
    <cleanup-interval value="3600" />
    <max-age value="86400" />
    <standalone-listener-port value="9999" />
</reliability>
```

Configuring Database Support for a J2EE Client

A J2EE client typically interacts with a datasource in an application server. The client connects to the data source by performing a JNDI lookup on its location.

You enter the information into one of the following files, based on your client.

- `META-INF/orion-ejb-jar.xml` for an EJB client
- `WEB-INF/orion-web.xml` for a JSP or servlet client
- `META-INF/orion-application-client.xml` for a Web application client

Datasources used for reliable clients should be configured to use local transactioning. To declare local transactioning, add the attribute `tx-level="local"` to the `<managed-data-source>` element. For example:

```
...
 <managed-data-source name="OracleDS" tx-level="local"
```

```
            jndi-name="jdbc/OracleManagedDS" ...
     />
 ...
```

In Example 5–9 the values for the `repository` attributes `type` and `jndiLocation` provide the datasource type and location details to the J2EE client. The repository elements are highlighted in bold.

Example 5–9   Database Connection for a J2EE Client

```
<reliability>
     <repository
          type="jdbc"
          jndiLocation="jdbc/OracleManagedDS"/>
     <retry-interval value="60" />
     <retry-limit value="10" />
     <poll-interval value="60" />
     <cleanup-interval value="3600" />
     <max-age value="86400" />
</reliability>
```

Configuring a Listener for a J2EE Client

For a J2EE client where the reliability client runs inside of an application, a Servlet class, `oracle.j2ee.ws.reliability.AckFaultServlet`, is provided for use in the application.

Example 5–10 illustrates the code that must be added to the application's `web.xml` file.

Example 5–10   Listener Configuration for a J2EE Client

```
<servlet>
    <servlet-name>wsrm</servlet-name>
    <servlet-class>oracle.j2ee.ws.reliability.AckFaultServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
    <servlet-mapping>
        <servlet-name>wsrm</servlet-name>
        <url-pattern>/wsrm/*</url-pattern>
    </servlet-mapping>
```

The replyTo URL must be set to the URL to which the `AckFaultServlet` is mapped. This is in contrast to the J2SE case, where the replyTo URL must correspond to the standalone listener's *host*:*port* value.

Dynamically Configuring Client-Side Reliability

OracleAS Web Services provides the ability to dynamically configure reliability features and receive notifications for acknowledgments and faults at the application level. This functionality is provided by the `oracle.webservices.reliability` package, a client-level API for managing reliability features.

The functionality contained in this package lets you override any default configuration, or the configuration that was initially provided to the client.

You must use the API if you want to implement message ordering for your Web service. Currently, using the API is the only way to start and end groups of ordered messages.

You can also use this API if you want to enforce reliability policies on a per-message basis.

Table 5–6 describes the classes in the `oracle.webservices.reliability` package. More detailed information about the classes, such as fields and method-level descriptions, can be found in the output of the Javadoc tool.

Table 5–6 Client APIs for Reliability

Class Name	Description
ReliabilityClientFactory	This is the factory class for creating a `ReliabilityClient`.
ReliabilityClient	The `ReliabilityClient` holds listeners for receiving asynchronous messages and configured stores for the client. It is a factory for `ReliabilityContext` instances.
ReliabilityContext	The class represents the context for reliable messaging that the client is in. The context can be used to override any default configuration for the client and to add listeners for reliability events. Ordered groups of messages can be started and ended only by using the context (and not by static configuration) as groups have to be determined dynamically
ReliabilityEvent	This is an event that is sent to a `ReliabilityListener`.
ReliabilityListener	Listener for reliability events (acks/faults). The client can implement a reliability listener and add it to its `ReliabilityContext` to receive notifications.

The APIs in the reliability package can be used to create the reliability context for both static stub and DII clients.

The `createContext()` method in the `ReliabilityClient` class creates a reliability context. The context is where you can set, among other features, the reliability listeners, message acknowledgment type, duplicate message elimination, guaranteed delivery, message expiration time, and the reply URL.

For a static stub, the reliability context is set on the current thread. All operations invoked on the stub within the scope of a reliability context inherit their reliability characteristics from that context.

For a DII client, the details about the service are passed to the client by URIs. "Providing Dynamic Configuration for a DII Web Service Client" on page 3-10 provides more information on how to pass management details to a DII Web service client.

Example 5–11 illustrates the code to make a stub that is generated by `genProxy` reliable. In the code sample, a reliability context is created that is used over the lifetime of the client (`client.createContext()`). The context is set on `ThreadContext` when it is created. This allows operations on the banking stub to inherit their reliability characteristics from the context.

Reliability is set to return asynchronous acknowledgments of messages received (`setAcknowledgmentType`). The code guarantees that messages will be delivered (`setGuaranteedDeliveryEnabled`), and that duplicate messages will be eliminated (`setDuplicateEliminationEnabled`).

Example 5–11 Reliability Code for a Static Stub Client

```
ReliabilityClient client= ReliabilityClientFactory.getClientFactory().getClient();
ReliabilityContext reliabilityContext = client.createContext();
```

```
reliabilityContext.setAcknowledgmentType(ReliabilityContext.ACKNOWLEDGMENT_TYPE_
CALLBACK);
reliabilityContext.setDuplicateEliminationEnabled(true);
reliabilityContext.setGuaranteedDeliveryEnabled(true);

Banking banking = new Banking_Impl();
Bank bank = banking.getBankPort();
bank.deposit(new Deposit("checking", 100));
```

Example 5–12 illustrates the same client code, but for a DII client. In this case, the service name and deposit details are obtained dynamically with URIs.

Example 5–12 Reliability Code for a DII Client

```
ReliabilityClient client = ReliabilityContextFactory.getDefault().getClient();
ReliabilityContext reliabilityContext = client.createContext();

reliabilityContext.setAcknowledgmentType(ReliabilityContext.ACKNOWLEDGMENT_TYPE_
CALLBACK);
reliabilityContext.setDuplicateEliminationEnabled(true);
reliabilityContext.setGuaranteedDeliveryEnabled(true);

Service service = ServiceFactory.newInstance().createService(new
QName("http://oracle.com/test/wsdl","TestService","test") );
Call call = service.createCall(new
QName("http://oracle.com/test/wsdl","TestServicePort"),
                   new QName("http://oracle.com/test/wsdl","deposit"));
call.setProperty(ClientConstants.WSM_INTERCEPTOR_PIPELINE_CONFIG,
                   new File("wsmClient.xml"));
```

Example 5–13 illustrates client code that performs message ordering. Like the previous examples, a reliability context is created that is used over the lifetime of the client (`reliabilityClient.createContext()`). To ensure that messages are received in the same order in which they are sent, embed the code which is capable of sending messages within the `startMessageOrdering` and `endMessageOrdering` methods. These methods are highlighted in bold.

Note that for message ordering to work, the methods `setDuplicateEliminationEnabled` and `setGuaranteedDeliveryEnabled` must also be present and set to `true`. These methods are also highlighted in bold.

Example 5–13 Message Ordering on the Client

```
reliabilityContext = reliabilityClient.createContext();

reliabilityContext.setAcknowledgmentType(ReliabilityContext.ACKNOWLEDGMENT_TYPE_
CALLBACK);
            reliabilityContext.setDuplicateEliminationEnabled(true);
            reliabilityContext.setGuaranteedDeliveryEnabled(true);
            reliabilityContext.startMessageOrdering();

      Banking banking = new Banking_Impl();
      Bank bank = banking.getBankPort();
            bank.deposit("Checking", 10.0);
            bank.withdraw("Checking", 5.0);

            reliabilityContext.endMessageOrdering();
```

Tool Support for Web Services Reliability

The following sections describe the support for reliability offered by Application Server Control and JDeveloper.

- WebServicesAssembler Support for Web Service Reliability
- Application Server Control Support for Web Service Reliability
- JDeveloper Support for Web Service Reliability

WebServicesAssembler Support for Web Service Reliability

This section describes how to use WebServicesAssembler commands to add a reliability configuration to a Web service. A configuration can be added in a top down or bottom up Web service assembly. A Web service client can also be assembled in this way. In each case, the configuration is specified in an XML file and passed to the WebServicesAssembler command with the `ddFileName` argument.

- Assembling Reliability into a Web Service Bottom Up
- Assembling Reliability into a Web Service Top Down
- Assembling Reliability into a J2SE Web Service Client Proxy

Assembling Reliability into a Web Service Bottom Up

The following general steps describe how to assemble a reliability configuration into a Web service bottom up.

1. Create the XML file that provides settings for the reliability features.

   "Server-Side Reliability Configuration Elements" on page 5-4 describe the port-level and operation-level reliability features that are available on the server.

2. Use the appropriate `*Assemble` command to assemble the Web service. Use the `ddFileName` argument to pass the name of the XML file that contains the reliability configuration to the command.

   "Web Service Assembly Commands" in the *Oracle Application Server Web Services Developer's Guide* provides more information on the commands that can assemble a Web service bottom up. See "`ddFileName`" in the "Deployment Descriptor Arguments" section of the *Oracle Application Server Web Services Developer's Guide* for more information on this argument.

Example 5–14 illustrates an `assemble` Ant task that uses the `ddFileName` argument to add the reliability configuration in `wsmBankServiceConfig.xml` to the Web service.

Example 5–14 Passing a Reliability Configuration in a Bottom Up Web Service Assembly

```
<oracle:assemble appName="bank"
            targetNamespace="http://www.oracle.com/bank"
            typeNamespace="http://www.oracle.com/bank"
            serviceName="Banking"
            interfaceName="oracle.ws.server.bank.Bank"
            className="oracle.ws.server.bank.BankImpl"
            input="./build/classes/service"
            output="build"
            ear="build/bank.ear"
            style="rpc"
            use="encoded"
            createOneWayOperations="true"
```

```
            ddFileName="wsmBankServiceConfig.xml"
            >
        </oracle:assemble>
```

Example 5–15 illustrates the contents of the `wsmBankServiceConfig.xml` for a server-side reliability configuration. Note that the contents of the file are enclosed in the `<oracle-webservices>` tag. The reliability configuration is highlighted in bold.

Example 5–15 Sample Server-Side Reliability Configuration

```
<oracle-webservices xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="oracle-webservices-management-10_0.xsd"
   schema-major-version="10" schema-minor-version="0">
   <webservice-description name="Banking">
      <port-component name="HttpSoap11">
         <runtime enabled="reliability">
            <reliability>
               <repository name="bank-server-store" type="memory"/>
               <ack-interval value="60"/>
               <order-interval value="60"/>
               <cleanup-interval value="10000"/>
               <ack-limit value="10"/>
               <max-age value="86000"/>
            </reliability>
         </runtime>
         <operations>
            <operation name="deposit">
               <runtime>
                  <reliability>
                     <duplication-elimination-required value="false"/>
                     <guaranteed-delivery-required value="false"/>
                  </reliability>
               </runtime>
            </operation>
         </operations>
      </port-component>
   </webservice-description>
</oracle-webservices>
```

Assembling Reliability into a Web Service Top Down

The following general steps describe how to assemble a reliability configuration into a Web service top down.

1. Provide settings for the reliability features that you want to enable in an XML file.

   "Server-Side Reliability Configuration Elements" on page 5-4 describe the port-level and operation-level reliability features that are available on the server.

2. Use the `topDownAssemble` command to assemble the Web service. Use the `ddFileName` argument to pass the name of the XML file that contains the reliability configuration to the command.

   "Assembling a Web Service from a WSDL" in the *Oracle Application Server Web Services Developer's Guide* provides more information about the `topDownAssemble` command and assembling a Web service top down. See "`ddFileName`" in the "Deployment Descriptor Arguments" section of the *Oracle Application Server Web Services Developer's Guide* for more information on the `ddFileName` argument.

Example 5–16 illustrates a `topDownAssemble` Ant task that uses the `ddFileName` argument to add the reliability configuration in `wsmLoggingServiceConfig.xml` to the Web service. The `ddFileName` argument is highlighted in bold.

Example 5–16 Passing a Reliability Configuration in a Top Down Web Service Assembly

```
<oracle:topDownAssemble
            wsdl=" ./wsdl/LoggingFacility.wsdl"
            unwrapParameters="false"
            className="oracle.demo.topdowndoclit.service.DocLitLoggerImpl"
            input="build/classes/service"
            output="build"
            ear="dist/doclit_topdown.ear"
            mappingFileName="type-mapping.xml"
            packageName="oracle.demo.topdowndoclit.service"
            fetchWsdlImports="true"
            classPath="./build/classes/client"
            ddFileName="wsmLoggingServiceConfig.xml"
</oracle:topDownAssemble>
```

Assembling Reliability into a J2SE Web Service Client Proxy

The following general steps describe how to assemble a reliability configuration into a J2SE Web service client proxy.

1. Provide settings for the reliability features that you want to enable in an XML file.

   "Client-Side Reliability Configuration Elements" on page 5-9 describe the port-level and operation-level reliability features that can be set for the client.

2. Use the `genProxy` command to assemble the Web service client. Use the `ddFileName` argument to pass the name of the XML file that contains the reliability configuration to the command.

   "Configuring Management Information for a J2SE Client" on page 3-7 provides more detailed information on adding a management configuration to a J2SE client.

   "Assembling a J2SE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide* provides more information on creating J2SE Web service clients.

Example 5–17 illustrates a `genProxy` Ant task that uses the `ddFileName` argument to add the reliability configuration in `wsmClientDD.xml` to the Web service. The `ddFileName` argument is highlighted in bold.

Example 5–17 Passing a Reliability Configuration in a Web Service Client Assembly

```
<oracle:genProxy
       wsdl="http://localhost:8888/bankdemo/bank?WSDL"
       output="test/src"
       debug="true"
       packageName="oracle.generated"
       ddFileName="wsmClientDD.xml"/>
</oracle:genProxy>
```

Example 5–18 illustrates the contents of the `wsmClientDD.xml` for a client-side reliability configuration. Note that the contents of the file are enclosed in the `<port-info>` tag.

Example 5–18 Sample Client-Side Reliability Configuration

```
...
```

```
<port-info>
   <runtime enabled="reliability">
      <reliability>
         <repository name="standalone-client-store" type="memory"/>
         <retry-interval value="60"/>
         <cleanup-interval value="10000"/>
         <poll-interval value="60"/>
         <retry-limit value="10"/>
         <max-age value="86000"/>
         <standalone-listener-port value="9876"/>
      </reliability>
   </runtime>
   <operations>
      <operation name="deposit">
         <runtime>
            <reliability>
               <guaranteed-delivery enabled="true"/>
               <duplicate-elimination enabled="true"/>
               <group-expiry-time value="860000"/>
               <reply-pattern value="Callback"/>
               <reply-to-url value="http://localhost:9876"/>
               <expiry value="860000"/>
            </reliability>
         </runtime>
      </operation>
   </operations>
</port-info>
...
```

Assembling Reliability into a J2EE Web Service Client

The following general steps describe how to assemble a reliability configuration into a J2EE Web service client.

1. Provide settings for the reliability features that you want to enable in an XML file.

   "Client-Side Reliability Configuration Elements" on page 5-9 describe the port-level and operation-level reliability features that can be set for the client.

2. Use the `genInterface` command to assemble the Web service client. Pass the reliability configuration to the command with the `ddFileName` argument. "How to Assemble a J2EE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide* provides detailed instructions for creating the J2EE Web service client.

   "Configuring Management Information for a J2EE Client" on page 3-8 provides more detailed information on adding a management configuration to a J2EE client.

Application Server Control Support for Web Service Reliability

Application Server Control can read and modify the reliability configuration of a deployed Web service. Once the configuration has been modified and applied, the Web service will run with the new configuration values. Application Server Control can be used to set Web service reliability options on the port and operation level.

On the server-side, Application Server Control only lets you view, enable, and disable reliability arguments. You cannot view or change any values for clients.

JDeveloper Support for Web Service Reliability

JDeveloper can be used to develop OracleAS Web Services and client Web service management configuration files. JDeveloper can aid you in the initial creation of these files or it can be used to add management configuration to existing files.

The "Configure Management and Reliability" wizard helps you configure port and operation level reliability for inbound and outbound SOAP messages. Options in the wizard allow you to configure all of the reliability elements described in "Port-Level Reliability Elements on the Server" on page 5-5 and "Operation Level Reliability Elements on the Server" on page 5-6.

Limitations

See "Ensuring Web Service Reliability" on page E-10.

Additional Information

For more information on:

- adding Web service management information, including security, to a Web service, see Chapter 3, "Managing Web Services".
- adding security information to a Web service, see Chapter 4, "Ensuring Web Services Security" and the *Oracle Application Server Web Services Security Guide*.
- adding auditing and logging information to a Web Service, see Chapter 6, "Auditing and Logging Messages".
- the contents of the `wsmgmt.xml` file which contains the security configuration, see Appendix A, "Understanding the Web Services Management Schema".
- assembling a Web service top down, see "Assembling a Web Service from a WSDL" in the *Oracle Application Server Web Services Developer's Guide*.
- using Java classes to assemble a Web service, see "Assembling a Web Service with Java Classes" in the *Oracle Application Server Web Services Developer's Guide*.
- using EJBs to assemble a Web service, see "Assembling a Web Service with EJBs" in the *Oracle Application Server Web Services Developer's Guide*.
- using JMS topics and destinations to assemble a Web service, see "Assembling Web Services with JMS Destinations" in the *Oracle Application Server Web Services Developer's Guide*.
- using database resources, such as PL/SQL packages, SQL queries, DML statements, Oracle Streams AQ, or server-side Java classes, to assemble a Web service, see "Assembling Database Web Services" in the *Oracle Application Server Web Services Developer's Guide*.
- assembling a J2EE Web Service client, see "Assembling a J2EE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.
- assembling a J2SE Web service client, see "Assembling a J2SE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.
- the contents of the `oracle-webservices.xml` deployment descriptor which contains the Web services management configuration, see "Packaging and Deploying Web Services" in the *Oracle Application Server Web Services Developer's Guide*.

- using WebServicesAssembler commands to assemble Web service artifacts, see "Using WebServicesAssembler" in the *Oracle Application Server Web Services Developer's Guide*.

6 Auditing and Logging Messages

This chapter describes the logging and auditing functionality available for Oracle Application Server Web Services. Auditing enables you to keep a complete, persistent record of request, response, and fault SOAP messages. It records the entire message. Logging enables you to extract and persistently store discrete portions of request, response, and fault SOAP messages.

- Understanding Auditing
- Managing Auditing on the Server
- Understanding Logging
- Managing Logging on the Server
- Tool Support for Web Services Auditing and Logging

Understanding Auditing

Auditing enables you to keep a complete, persistent record of SOAP requests, responses, and faults. During development it is often very convenient to be able to inspect the contents of SOAP requests and responses to diagnose problems.

In a production environment, specific SOAP messages can be stored to support non-repudiation efforts. For example, an on-line retailer could record all received purchase order requests.

Auditing also enables you to design a strategy that records and analyzes security-related events. A history of audit records can produce an audit trail enabling the reconstruction and examination of a sequence of events. The audit trail can be used to detect attacks, confirm compliance with policy, deter abuse, and so on.

Auditing and Performance

Auditing is initiated and performed by the server. It cannot be initiated or performed by the client. Adding auditing to your Web service does not require changes or extensions to the WSDL.

If you enable auditing, there will probably be a measurable impact upon the performance of the system. You should use it only during development and debugging, or when there is a clear need to persist the entire contents of a message during production. Auditing for non-repudiation is a good example. In this case, it is likely that you will have to audit only a small set of Web service operations, so the performance impact should be acceptable.

The size of the logged messages will effect how large the performance impact will be. Larger message will take more time and system resources to store.

Processing Audit Messages

Auditing can be applied at runtime to messages flowing into the platform (request messages), flowing out of the platform (response messages), and generated fault messages. The following sections describe how the runtime processes these messages.

- Auditing Request Messages
- Auditing Response Messages
- Auditing Fault Messages

Example 6–1 illustrates a sample logged SOAP message. The original SOAP message appears in the <MSG_TEXT> element and is highlighted in bold in this example.

Example 6–1 Sample SOAP Message in the Audit Log

```
<MESSAGE>
  <HEADER>
      <TSTZ_ORIGINATING>2003-12-22T16:44:09.455-05:00</TSTZ_ORIGINATING>
      <ORG_ID>wsm</ORG_ID>
      <COMPONENT_ID>auditing</COMPONENT_ID>
      <MSG_TYPE TYPE="NOTIFICATION"></MSG_TYPE>
      <MSG_LEVEL>1</MSG_LEVEL>
      <HOST_ID>jdoe-us</HOST_ID>
      <HOST_NWADDR>111.2.3.444</HOST_NWADDR>
      <PROCESS_ID>null-12</PROCESS_ID>
      <USER_ID>jdoe</USER_ID>
   </HEADER>
   <CORRELATION_DATA>

   <EXEC_CONTEXT_ID><UNIQUE_ID>138.2.8.162:82684:1072129449465</UNIQUE_
ID><SEQ>0</SEQ></EXEC_CONTEXT_ID>
      </CORRELATION_DATA>
      <PAYLOAD>
         <MSG_TEXT>&lt;env:Envelope
xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">&lt;env:Header/>
&lt;env:Body>&lt;operation:echo xmlns:operation="http://oracle.com/test/wsdl">
&lt;String_1>Hello&lt;/String1>&lt;/operation:echo>&lt;/env:Body>
&lt;/env:Envelope></MSG_TEXT>
     </PAYLOAD>
</MESSAGE>
```

Auditing Request Messages

When an inbound SOAP message is received, the runtime determines, based on the configuration, whether auditing is enabled for the port. The configuration also indicates whether request messages are to be audited. If auditing is enabled for both the port and operation, then the entire message is written to the audit log.

Auditing Response Messages

When the Web service implementation creates outbound response SOAP messages, the management features are applied in reverse order. If auditing is enabled for the port and the configuration also indicates that response messages are to be audited, then the entire message is written to the audit log.

Auditing Fault Messages

Fault messages can be generated by the Web service implementation or by any of the application handlers. If the auditing of fault messages is enabled for the port and operation by the configuration, then fault messages are written to the audit log.

Managing Auditing on the Server

The configuration information for auditing is located in the server-side Web services proprietary deployment descriptor `oracle-webservices.xml`. The contents of this file can be modified during packaging, deployment, or runtime. At runtime, changes must be made using either Application Server Control or by editing the *ORACLE_HOME*`\j2ee\home\config\wsmgmt.xml` file.

Auditing can be performed only at the operation level. Hence, there is no global or port-level configuration.

Auditing captures request, response, and fault messages and stores them in the persistent file *ORACLE_HOME*`\log\wsmgmt\audit\log.xml`. This is a rolling file: after it reaches a certain size, it is renamed `log-[`*number*`].xml` and a new `log.xml` file is started. Over time, older versions of the file may be deleted. Application Server Control can read this file. For information about this file see "Configuring Auditing for a Web Service" in the Application Server Control on-line help

Server-Side Auditing Configuration Elements

An auditing configuration can be defined for each operation in a Web service port. The `<auditing>` element and its three attributes, `request`, `response`, and `fault`, define the auditing configuration. Each attribute can have a `true` or `false` value to indicate whether Web service management will store these messages. The following is a sample auditing configuration.

```
<auditing request="true" response="false" fault="false"/>
```

Example 6–2 illustrates the `<auditing>` element, highlighted in bold, as it appears in the `oracle-webservices.xml` deployment descriptor. Note that even though you cannot specify auditing for a port, a `<runtime enabled="auditing">` element is still required at the port level. The schema for this file is:

`http://xmlns.oracle.com/oracleas/schema/oracle-webservices-10_0.xsd`

Example 6–2 Operation-Level Auditing Element in the Server-Side Configuration File

```
...
<port-component name="String">
  ...

   <runtime enabled="auditing">
     ...
   </runtime>
    <operations>
      <operation name "..." >
        <runtime>
        ...
        <auditing request="true" response="true" fault="false"/>
        </runtime>
      </operation>
    </operations>
</port-component>
```

```
...
```

Table 6–1 describes the attributes of the auditing element.

Table 6–1 Elements to Enable Message Auditing

Element Name	Description
<auditing>	Has attributes `request`, `response`, and `fault` that accept a Boolean value. If an attribute is `true`, then messages of that type will be audited. ■ `request`—Indicates whether request messages will be audited. Default is `true`. ■ `response`—Indicates whether response messages will be audited. Default is `false`. ■ `fault`—Indicates whether fault messages will be audited. Default is `false`.

Managing Auditing on the Client

The configuration information for auditing is located in the client-side Web services proprietary deployment descriptor. Depending on the client application, this can be either `orion-web.xml`, `orion-ejb-jar.xml`, or `orion-application-client.xml`. The contents of this file can be modified during packaging or deployment.

Auditing can be performed only at the operation level. Hence, there is no global or port-level configuration.

Similar to the server-side, a client-side auditing configuration can be defined for each operation in a Web service port. The `<auditing>` element and its three attributes, `request`, `response`, and `fault`, define the auditing configuration. Each attribute can have a `true` or `false` value to indicate whether Web service management will store these messages. Table 6–1 provides a description of the `<auditing>` element and its attributes. The following is a sample auditing configuration.

```
<auditing request="true" response="false" fault="false"/>
```

The auditing configuration appears within the `<operation>` clause of the `<service-ref-mapping>` element. Each operation can have its own auditing configuration. Example 6–3 illustrates the skeleton of an `orion-web.xml` Oracle-proprietary client-side deployment descriptor that indicates the position of the auditing configuration in the hierarchy. The auditing configuration is highlighted in bold. Note that the configuration occurs within the `<port-info>` element. The schema for this file is:

`http://xmlns.oracle.com/oracleas/schema/orion-web-10_0.xsd`

Example 6–3 Sample Client-Side orion-web.xml with an Auditing Configuration

```
<orion-web-app ...>
    <service-ref-mapping name="service/MyTestServiceRef">
        <port-info>
            ...
            <runtime>
                ...
                <security/>
                <reliability/>
                ...
            </runtime>
```

```
                <operations>
                    ...
                      <operation name="echo">
                        <runtime enabled = "security,reliability,auditing">
                            ...
                            <security/>
                            <reliability/>
                            <auditing request="true" response="true" fault="false"/>
                            ...
                        </runtime>
                      </operation>
                    ...
                </operations>
            </port-info>
        </service-ref-mapping>
</orion-web-app>
```

Understanding Logging

OracleAS Web Services includes basic SOAP message logging. This functionality lets you extract and persistently store discrete portions of request, response, and fault SOAP messages.

There are many situations where you might want to persistently capture information from SOAP messages. For example many large organizations deploying internal Web services may want to track which groups within the organization are using the service. Logging could enable this by extracting and collecting group information from messages.

Logging and Performance

Logging is likely to have a minor impact upon the performance of the system. Since only simple types can be extracted from messages, this impact should be limited. If the logging mechanism is asked to extract large amounts of information from individual messages (for example, large text blocks) a proportional performance impact may occur.

Processing Logging Messages

Logging can be applied at runtime to messages flowing into the platform (request messages), flowing out of the platform (response messages), and generated fault messages. See the following sections for information on how the runtime processes these messages.

- Logging Request Messages
- Logging Response Messages
- Logging Fault Messages

Example 6–4 provides a sample of a log record derived from an inbound SOAP request message. In the log, two messages are logged, `account` and `amount`.

Example 6–4 Sample Log File Entry

```
<MESSAGE>
  <HEADER>
     <TSTZ_ORIGINATING>2004-01-30T11:32:19.724-05:00</TSTZ_ORIGINATING>
     <ORG_ID>wsm</ORG_ID>
     <COMPONENT_ID>logging</COMPONENT_ID>
```

```
    <MSG_TYPE TYPE="NOTIFICATION"></MSG_TYPE>
    <MSG_LEVEL>1</MSG_LEVEL>
    <HOST_ID>xwen-us</HOST_ID>
    <HOST_NWADDR>138.2.8.175</HOST_NWADDR>
    <PROCESS_ID>null-10</PROCESS_ID>
    <USER_ID>xwen</USER_ID>
  </HEADER>
    <CORRELATION_DATA>

<EXEC_CONTEXT_ID><UNIQUE_ID>138.2.8.175:52155:1075480339694</UNIQUE_
ID><SEQ>0</SEQ></EXEC_CONTEXT_ID>
    </CORRELATION_DATA>
    <PAYLOAD>
      <MSG_TEXT>account='Checking',amount='100.0'</MSG_TEXT>
    </PAYLOAD>
</MESSAGE>
```

Logging Request Messages

When an inbound SOAP message is received, the runtime determines, based on the configuration, whether logging is enabled for the port. The logging mechanism then determines, based again on the configuration, what information, if any, is to be extracted from the message for the operation. This information is then stored.

Logging Response Messages

When the Web service implementation creates outbound response SOAP messages, the management features are applied reverse order. If logging is enabled for the port and the configuration also indicates that information from response messages should be logged, then the information is extracted and stored.

Logging Fault Messages

Fault messages can be generated by the Web service implementation or any of the application handlers. If logging fault messages is enabled by the configuration, then the specified information is extracted from the message and stored.

Managing Logging on the Server

The configuration information for logging is stored in the server-side Web services proprietary deployment descriptor `oracle-webservices.xml`. The contents of this file can be modified during packaging, deployment, or runtime. At runtime, changes must be made either by using Application Server Control or by editing the *ORACLE_HOME*`\j2ee\home\config\wsmgmt.xml` file.

Logging must be configured at the port and operation levels. There is no global-level configuration.

Server-Side Logging Configuration Elements

If logging features are employed, they must be configured at both the port and the operation level. The port-level configuration contains information that is referenced by the operation level. Unlike security, operation-level configuration does not override port-level configuration.

Logging information is stored in *ORACLE_HOME*`/log/wsmgmt/logging/log.xml`. This is a rolling file: after it reaches a certain size, it is renamed `log-[`*number*`].xml` and a new `log.xml` file is started.

Port-Level Logging Elements on the Server

Example 6–5 displays the port-level logging elements as they appear in the server-side Web services proprietary deployment descriptor, `oracle-webservices.xml`. These elements are used to create a set of namespace prefix and URI pairs that are referenced by the operation level configuration. The schema for this file is:

http://xmlns.oracle.com/oracleas/schema/oracle-webservices-10_0.xsd

Example 6–5 Port-Level Logging Elements in the Server-Side Configuration File

```
...
<port-component name="...">
  ...
 <runtime enabled="logging">
  <logging>
     <namespaces>
       <namespace prefix='env' uri='http://schemas.xmlsoap.org/soap/envelope/'/>
       <namespace prefix='ns0' uri='http://oracle.com/test/wsdl'/>
     </namespaces>
  </logging>
  ...
 </runtime>
</port-component>
...
```

Table 6–2 describes the port-level namespace elements that must be set for logging information to be reported. The namespace prefix-URI pairs are used for processing `xpath` expressions.

Table 6–2 Namespace Elements for Logging

Element Name	Description
<namespace>	Specifies a name-value pair of namespace prefixes and namespace URIs. The namespace URIs are implicitly referenced by the use of the prefixes in `xpath` expressions defined for logging at the operation level.

Operation Level Logging Elements on the Server

Example 6–6 illustrates the operation-level logging elements as they appear in the server-side Web services proprietary deployment descriptor, `oracle-webservices.xml`. At the operation level, you can enable or disable logging for request, response, and fault messages. The schema for this file is:

`http://xmlns.oracle.com/oracleas/schema/oracle-webservices-10_0.xsd`

Example 6–6 Operation-Level Logging Elements in the Server-Side Configuration File

```
...
<port-component name="String">
  ...
    <operations>
      <operation name="...">
       <runtime>
        ...
        <logging>
          <request enabled='true'>
            <attributes>
```

```
              <attribute name='input'
                 ...
                 xpath='/env:Envelope/env:Body/ns0:echo/String_1/text()'/>
            </attributes>
          </request>
          <response enabled='true'>
            <attributes>
              <attribute name='output'
                 xpath='/env:Envelope/env:Body/ns0:echoResponse/result/text()'/>
            </attributes>
          </response>
          <fault enabled='true'>
            <attributes>
              <attribute name='output'
                 xpath='/env:Envelope/env:Body/ns0:echoResponse/result/text()'/>
            </attributes>
          </fault>
        </logging>
      </runtime>
     </operation>
    </operations>
</port-component>
...
```

Table 6–3 describes the message logging elements: `request`, `response`, and `fault`. Setting an element to `true` enables logging for that message type. If logging is enabled, you can specify one or more attributes you want to log for each message type. These attributes are indicated by its `name` and `xpath`.

If logging for a message type is enabled, but no attributes are configured, then no logging will occur. Similarly, if the `xpath` query returns no results, then no logging will occur.

Table 6–3 Elements to Enable Message Logging

Element Name	Description
<request>	If `enabled=true`, specify name-value pairs of message attributes. Each attribute is specified by an attribute `name` and `xpath` value. Default is `true`.
<response>	If `enabled=true`, specify name-value pairs of message attributes. Each attribute is specified by an attribute `name` and `xpath` value. Default is `true`.
<fault>	If `enabled=true`, specify name-value pairs of message attributes. Each attribute is specified by an attribute `name` and `xpath` value. Default is `true`.

Tool Support for Web Services Auditing and Logging

This section provides an overview of the parts of the Web service auditing and logging configuration that can be set by the JDeveloper and Application Server Control tools. For detailed information of the individual auditing and logging options that can be controlled by these tools, see the on-line help for Application Server Control and JDeveloper.

- Application Server Control Support for Auditing and Logging
- JDeveloper Support for Auditing and Logging

WebServicesAssembler Support for Web Service Auditing and Logging

This section describes how to use WebServicesAssembler commands to add an auditing and logging configuration to a Web service. A configuration can be added in a top down or bottom up Web service assembly. Auditing can also be assembled into a Web service client (logging is not available on the client). In each case, the configuration is specified in an XML file and passed to the WebServicesAssembler command with the `ddFileName` argument.

- Assembling Auditing and Logging into a Web Service Bottom Up
- Assembling Auditing and Logging into a Web Service Top Down
- Assembling Auditing into a J2SE Web Service Client Proxy
- Assembling Auditing into a J2EE Web Service Client

Assembling Auditing and Logging into a Web Service Bottom Up

The following general steps describe how to assemble an auditing and logging configuration into a Web service bottom up.

1. Create the XML file that provides settings for the auditing and logging features you want to enable. There are two different ways in which you can do this.

   - Manually write the XML file that contains the auditing and logging settings. Example 6–8 illustrates a sample XML file that contains these settings. Save the file.
   - Run the WebServicesAssembler tool with the appropriate `*Assemble` command for your Web service. One of the files it creates will be an `oracle-webservices.xml` file that contains a skeleton of the ports and operations. For example, you could run the Ant task in Example 6–7 *without* the `ddFileName` argument to obtain this file. Edit `oracle-webservices.xml` file to enter the appropriate settings for the auditing and logging features. Save the file under a different name.

   "Server-Side Auditing Configuration Elements" on page 6-3 and "Server-Side Logging Configuration Elements" on page 6-6 describe the port-level and operation-level auditing and logging features that are available on the server.

2. Use the appropriate `*Assemble` command to assemble the Web service. Use the `ddFileName` argument to pass the file with the auditing and logging configuration you created in Step 1 to the WebServicesAssembler tool.

   "Web Service Assembly Commands" in the *Oracle Application Server Web Services Developer's Guide* provides more information on the commands that can assemble a Web service bottom up. See "`ddFileName`" in the "Deployment Descriptor Arguments" section of the *Oracle Application Server Web Services Developer's Guide* for more information on this argument.

Example 6–7 illustrates an `assemble` Ant task that uses the `ddFileName` argument to add the auditing and logging configuration in `wsmBankServiceConfig.xml` to the Web service.

Example 6–7 Passing an Auditing and Logging Configuration in a Bottom Up Web Service Assembly

```
<oracle:assemble appName="bank"
            targetNamespace="http://www.oracle.com/bank"
            typeNamespace="http://www.oracle.com/bank"
            serviceName="Banking"
            interfaceName="oracle.ws.server.bank.Bank"
```

```
            className="oracle.ws.server.bank.BankImpl"
            input="./build/classes/service"
            output="build"
            ear="build/bank.ear"
            style="rpc"
            use="encoded"
            ddFileName="wsmBankServiceConfig.xml"
            >
        </oracle:assemble>
```

Example 6–8 illustrates the contents of the wsmBankServiceConfig.xml for a server-side auditing and logging configuration. While both auditing and logging are enabled at the port level, only a logging configuration is provided. At the operations level, an auditing configuration is provided for the withdraw operation and a logging configuration is provided for the deposit operation. Note that the contents of the file are enclosed in the <oracle-webservices> element.

Example 6–8 Sample Server-Side Auditing and Logging Configuration

```
<?xml version='1.0' encoding='UTF-8'?>
<oracle-webservices xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/
                                  oracle-webservices-10_0.xsd">
    <webservice-description name="Bank">
        <port-component name="HttpSoap11">
            <runtime enabled="auditing, logging">
                <auditing/>
                <logging>
                    <namespaces>
                        <namespace prefix="target_ns"
uri="http://oracle.com/bank/wsdl"/>
                        <namespace prefix="soap"
uri="http://schemas.xmlsoap.org/soap/envelope/"/>
                    </namespaces>
                </logging>
            </runtime>
            <operations>
                <operation name="withdraw">
                    <runtime>
                        <auditing request="true" response="true" fault="false"/>
                    </runtime>
                </operation>
                <operation name="deposit">
                    <runtime>
                        <logging>
                            <request enabled="true">
                                <attributes>
                                    <attribute name="request_string"
             xpath="/soap:Envelope/soap:Body/target_ns:deposit/String_1/text()"/>
                                </attributes>
                            </request>
                            <response enabled="true">
                                <attributes>
                                    <attribute name="response_result"
        xpath="/soap:Envelope/soap:Body/target_ns:depositResponse/result/text()"/>
                                </attributes>
                            </response>
                        </logging>
                    </runtime>
                </operation>
```

```
            </operations>
        </port-component>
    </webservice-description>
</oracle-webservices>
```

Assembling Auditing and Logging into a Web Service Top Down

The following general steps describe how to assemble an auditing and logging configuration into a Web service top down.

1. Create the XML file that provides settings for the auditing and logging features that you want to enable. There are two different ways in which you can do this.

   - Manually write the XML file that contains the auditing and logging settings. Example 6–8 illustrates an XML file that contains sample auditing and logging settings.
   - Run the WebServicesAssembler tool with the `topDownAssemble` command. One of the files it creates is an `oracle-webservices.xml` file that contains a skeleton of the ports and operations. For example, you could run the Ant task in Example 6–9 *without* the `ddFileName` argument to obtain this file. Edit `oracle-webservices.xml` file to enter the appropriate settings for the auditing and logging features. Save the file under a different name.

   "Server-Side Auditing Configuration Elements" on page 6-3 and "Server-Side Logging Configuration Elements" on page 6-6 describe the port-level and operation-level auditing and logging features that are available on the server.

2. Use the `topDownAssemble` command to assemble the Web service. Use the `ddFileName` argument to pass the file with the auditing and logging configuration you created in Step 1 to the WebServicesAssembler tool.

   "Assembling a Web Service from a WSDL" in the *Oracle Application Server Web Services Developer's Guide*, provides more information about the `topDownAssemble` command and assembling a Web service top down. See "`ddFileName`" in the "Deployment Descriptor Arguments" section of the *Oracle Application Server Web Services Developer's Guide* for more information on this argument.

Example 6–9 illustrates a `topDownAssemble` Ant task that uses the `ddFileName` argument to add the auditing and logging configuration in `wsmLoggingServiceConfig.xml` to the Web service.

Example 6–9 Passing an Auditing and Logging Configuration in a Top Down Web Service Assembly

```
<oracle:topDownAssemble
            wsdl="BankService.wsdl"
            input="build/classes/service"
            output="build"
            className="oracle.ws.server.bank.BankImpl"
            ear="dist/bank.ear"
            packageName="oracle.ws.server.bank"
            ddFileName="BankServiceWsmConfig.xml"/>
```

Assembling Auditing into a J2SE Web Service Client Proxy

The following general steps describe how to assemble an auditing configuration into a J2SE Web service client proxy. Logging is not available for a Web service client.

1. Provide settings for the auditing features that you want to enable in an XML file.

"Managing Auditing on the Client" on page 6-4 describe the operation-level auditing features that can be set for the client.

2. Use the `genProxy` command to assemble the Web service proxy. Pass the auditing configuration to the command with the `ddFileName` argument.

   This will result in the generation of an XML file. For the `genProxy` command illustrated in Example 6–10, this would result in the file `oracle\generated\runtime\BankPortBinding_Stub.xml` being generated into the `test\src` directory.

   "Configuring Management Information for a J2SE Client" on page 3-7 provides more detailed information on adding a management configuration to a J2SE client.

Example 6–10 illustrates a `genProxy` Ant task that uses the `ddFileName` argument to add the auditing configuration in `wsmClientDD.xml` to the Web service.

Example 6–10 Passing an Auditing Configuration in a Web Service Client Assembly

```
<oracle:genProxy
       wsdl="http://localhost:8888/bankdemo/bank?WSDL"
       output="test/src"
       packageName="oracle.generated"
       ddFileName="wsmClientDD.xml"/>
</oracle:genProxy>
```

Example 6–11 illustrates the contents of the `wsmClientDD.xml` for a client-side auditing configuration. The auditing configuration is highlighted in bold. Note that the contents of the file are enclosed in the `<port-info>` tag.

Example 6–11 Sample Client-Side Auditing Configuration

```
<port-info>
   <runtime>
   </runtime>
   <operations>
      <operation name="deposit">
         <runtime>
            <auditing request="true" response="true" fault="false"/>
         </runtime>
      </operation>
   </operations>
</port-info>
```

Assembling Auditing into a J2EE Web Service Client

J2EE Web service clients can be configured to perform auditing. Logging is not available for Web service clients. The following general steps describe how you can add an auditing configuration to a J2EE Web service client. In addition to generating the client code, you must also edit both the standard and the Oracle-proprietary Web service deployment descriptor.

1. Generate the J2EE client code by providing the WSDL and the service endpoint interface as input to the `genInterface` command. See "How to Assemble a J2EE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide* for more detailed information on using WebServicesAssembler to assemble J2EE client code.

2. Edit the deployment descriptor for the J2EE component (either `web.xml`, `ejb-jar.xml`, or `application-client.xml`) to add the code to the `<service-ref>` element that will allow the component to access the Web service

endpoint. For examples of how auditing and other Web service management information can be added to a standard J2EE Web service client deployment descriptor, see the following examples.

- Example 3–5, "web.xml—Static Configuration for a Servlet or JSP Web Service Client"
- Example 3–7, "ejb-jar.xml—Static Configuration for an EJB Web Service Client"
- Example 3–9, "application-client.xml—Static Configuration for an Application Client Web Service Client"

See "Adding J2EE Web Service Client Information to Deployment Descriptors" in the *Oracle Application Server Web Services Developer's Guide* for information on the `<service-ref>` element and its contents.

3. Edit the Oracle-proprietary deployment descriptor for the J2EE component (either `orion-web.xml`, `orion-ejb-jar.xml`, or `orion-application-client.xml`) to add OC4J platform-specific information and the auditing configuration to the `<service-ref-mapping>` element.

   For examples of how auditing and other Web service management information can be added to a J2EE Web service client, see the following examples.

   - Example 3–6, "orion-web.xml—Static Configuration and Management Information for a Servlet or JSP Web Service Client"
   - Example 3–8, "orion-ejb-jar.xml—Static Configuration and Management Information for an EJB Web Service Client"
   - Example 3–10, "orion-application-client.xml—Static Configuration and Management Information for an Application Client Web Service Client"

   See "Adding OC4J-Specific Platform Information" in the *Oracle Application Server Web Services Developer's Guide* for information on the `<service-ref-mapping>` element and its contents.

   Example 6–12 illustrates a sample `orion-web.xml` file that has been edited to contain an auditing configuration. The configuration appears within the `<service-ref-mapping>` element, which is used to provide platform-specific information.

Example 6–12 orion-web.xml File Edited to Contain an Auditing Configuration

```
<?xml version="1.0"?>
<orion-web-app
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/orion-web-1
0_0.xsd">
    <service-ref-mapping name="service/BankTeller">
        <port-info>
            <wsdl-port namespaceURI="http://oracle.com/bank"
localpart="BankPort"/>
<service-endpoint-interface>oracle.ws.client.bank.BankService</service-endpoint-in
terface>
            <runtime enabled="auditing"/>
            <operations>
                <operation name="deposit">
                    <runtime>
                        <auditing request="true" response="true" fault="false"/>
                    </runtime>
                </operation>
            </operations>
```

```
        </port-info>
    </service-ref-mapping>
</orion-web-app>
```

Application Server Control Support for Auditing and Logging

Application Server Control lets you enable and configure different auditing options at deployment and at runtime. Application Server Control lets you perform the following tasks.

- Modify at deployment time which Web service operations and messages to audit.
- Control at runtime which Web service operations and messages to audit.
- Browse the messages contained in the audit logs.

Similarly, you can enable and configure different logging options at deployment and at runtime. Application Server Control lets you perform the following tasks.

- Modify at deployment time the type of information to extract and log.
- Control at runtime the type of information to extract and log.
- Browse the messages contained within the logs.

For more information, see the topics *Configuring Auditing for a Web Service* and *Configuring Logging for a Web Service* in the Application Server Control on-line help.

JDeveloper Support for Auditing and Logging

Wizards in JDeveloper let you configure auditing and logging for a Web service in a deployment module. The auditing and logging configuration is stored in the Oracle-proprietary deployment descriptor, `oracle-webservices.xml`.

When JDeveloper performs the final packaging of a Web service module, it includes the deployment descriptor in the location required by the Web services runtime.

When JDeveloper is used as the deployment tool for deploying Web services, you have the option to override any auditing and logging configuration that was set during packaging.

This configuration created by JDeveloper can also be overridden at deployment by Application Server Control. See the JDeveloper on-line help for more information on enabling logging and auditing. For more information on overriding auditing and logging features after deployment, see "Edit Auditing Configuration Page" and "Edit Logging Configuration Page" in the Application Server Control on-line help.

Limitations

See "Auditing and Logging Messages" on page E-10.

Additional Information

For more information on:

- the contents of the `oracle-webservices.xml` deployment descriptor which contains the Web services management configuration, see Chapter 18, *Packaging and Deploying Web Services*, in the *Oracle Application Server Web Services Developer's Guide*.

- adding Web service management information, including security, to a Web service, see Chapter 3, "Managing Web Services".
- adding security information to a Web service, see Chapter 4, "Ensuring Web Services Security" and the *Oracle Application Server Web Services Security Guide*.
- adding reliability information to a Web service, see Chapter 5, "Ensuring Web Service Reliability".
- the contents of the `wsmgmt.xml` file which contains the security configuration, see Appendix A, "Understanding the Web Services Management Schema".
- assembling a Web service top down, see "Assembling a Web Service from a WSDL" in the *Oracle Application Server Web Services Developer's Guide*.
- using Java classes to assemble a Web service, see "Assembling a Web Service with Java Classes" in the *Oracle Application Server Web Services Developer's Guide*.
- using EJBs to assemble a Web service, see "Assembling a Web Service with EJBs" in the *Oracle Application Server Web Services Developer's Guide*.
- using JMS topics and destinations to assemble a Web service, see "Assembling Web Services with JMS Destinations" in the *Oracle Application Server Web Services Developer's Guide*.
- using database resources, such as PL/SQL packages, SQL queries, DML statements, Oracle Streams AQ, or server-side Java classes, to assemble a Web service, see "Assembling Database Web Services" in the *Oracle Application Server Web Services Developer's Guide*.
- assembling a J2EE Web Service client, see "Assembling a J2EE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.
- assembling a J2SE Web service client, see "Assembling a J2SE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.
- using WebServicesAssembler commands to assemble Web service artifacts, see "Using WebServicesAssembler" in the *Oracle Application Server Web Services Developer's Guide*.
- the contents of the `oracle-webservices.xml` deployment descriptor which contains the Web services management configuration, see "Packaging and Deploying Web Services" in the *Oracle Application Server Web Services Developer's Guide*.

7

Custom Serialization of Java Value Types

This chapter describes the Custom Serialization Framework API. This API enables you to map and serialize Java value types between schema-defined XML types and Java value type classes that do not conform to the JAX-RPC value type requirements and cannot be handled by the default JAX-RPC serialization mechanism. This framework can also be used to implement custom mapping for the schema types that have default mappings. For example, you can use the framework to map `xsd:dateTime` to the custom class `com.hello.MyDate` instead of its default mapping `java.util.Calendar`.

The Custom Serialization Framework API can be used to serialize Java value types for publishing a Web service top down, bottom up, or for generating Java value types from a schema to publish a Web service endpoint.

The built-in data type mappings that OracleAS Web Services supports can be found in Appendix D, "Mapping Java Types to XML and WSDL Types" and in the JAX-RPC 1.1 specification.

`http://java.sun.com/webservices/jaxrpc/index.jsp`

You do not need to create custom mappings for these Java value types. "OC4J Support for Java Value Types" on page D-3 and Chapter 5.4 of the JAX-RPC 1.1 specification lists the requirements for Java value types.

- The Custom Serialization Framework API
- Using Custom Serialization in Web Service Development

The Custom Serialization Framework API

If you want to expose a custom or non-standard Java value type in your Web service, OracleAS Web Services provides an API that lets you serialize them into an XML representation. The `SOAPElementSerializer` interface in the `oracle.webservices.databinding` package defines the interface for pluggable custom serializers for the OracleAS Web Services JAX-RPC implementation.

```
public interface SOAPElementSerializer
```

The serializer converts a Java object to an XML representation of SAAJ `SOAPElement`. An instance of `SOAPElementSerializer` serializes a Java object to a `SOAPElement` and deserializes a `SOAPElement` to a Java object.

The interface has the following methods.

- `public SOAPElement serialize(QName tagName, Object object)`

Implements the `serialize` method to marshall a Java object to an XML fragment of a SAAJ `SOAPElement` instance. For this serializer implementation to be reusable, the implementation of `serialize` should use the value of the `tagName` parameter as the top level element tag name of the returned `SOAPElement` instance. This is because an XML schema `complexType` may be used for different element definitions.

- `public Object deserialize(SOAPElement element)`

  Implements the `deserialize` method to unmarshal an XML fragment of a SAAJ `SOAPElement` instance to a Java object.

- `public void init(Map prop)`

  Called by the Oracle Application Server Web Services runtime to indicate to a `SOAPElementSerializer` that the `SOAPElementSerializer` implementation instance is being placed into service. The `init` method is called with an instance of `Map` created from the `init-param` elements specified under the corresponding serializer element in the Custom Type Mapping File. Both the `key` type and `value` type of the `Map` instance are `java.lang.String`.

An implementation of the interface requires a default no-argument constructor. For more information on the `SOAPElementSerializer` interface, see the output of the Javadoc tool.

Using Custom Serialization in Web Service Development

To use the custom serialization in your Web service, you provide an implementation of the `SOAPElementSerializer` for your custom Java type value class. You must also provide the custom Java value type class and an XML configuration file that defines the mapping between the Java type value class and the XML schema type it should represent. You can find more information about these files in "Developing a Custom Serializer Implementation for a Java Type Value Class" on page 7-3, "Defining a Custom Java Type Value Class" on page 7-3, and "Creating an oracle-webservices Type Mapping Configuration" on page 7-6.

The following sections describe how you can use custom serialization in top down, bottom up and schema-driven Web service development.

- Implementing a Custom Serializer
- Using Custom Serialization in Top Down Web Service Development
- Using Custom Serialization in Bottom Up Web Service Development
- Using Custom Serialization in Schema-Driven Web Service Development
- Implementing a Serializer with Custom Marshalling Logic

Implementing a Custom Serializer

This section describes the steps in developing a custom serializer for a Java value type class. Typically, you start with the custom Java value type class that you want to express as an XML schema type. From there, you develop a custom serializer implementation of the class, the service endpoint interface, and its implementation. The oracle-webservices type mapping configuration XML file identifies the schema type that will be used to represent the custom Java type value class.

For top down Web service assembly, the service endpoint interface is generated from a WSDL. For bottom up Web service assembly, develop a Java service endpoint interface

- Defining a Custom Java Type Value Class
- Developing a Custom Serializer Implementation for a Java Type Value Class
- Developing a Service Endpoint Interface that Uses a Java Type Value Class
- Creating an oracle-webservices Type Mapping Configuration
- Using Custom Types in Client-Side Proxy Code

Defining a Custom Java Type Value Class

This section illustrates a custom Java type value class that can be used to map to an XML schema type.

Note that the custom Java type class does not have to be `Serializable` if the Web service is not going to be used with any J2EE features that require Serializable (such as EJB or JMS).

An empty constructor is not a requirement. The serializer implementation that you write programmatically constructs the instance of the corresponding Java type and populates the content of the Java object. This is why you want to use the custom serializer. It provides full control over the XML-to- Java and Java-to-XML mappings.

In JAX-RPC, `xsd:dateTime` can be represented by either `java.util.Calendar` or `java.util.Date`. If you do not want to use either of these classes to represent a date, you can implement your own date class. Example 7–1 illustrates a custom Java value class, `oracle.demo.custom_type.MyDate`, that can be used to represent a date.

Example 7–1 Sample Custom Java Type Value Class

```
package oracle.demo.custom_type;

public class MyDate implements java.io.Serializable{

    private String string;

    public MyDate(String s) {
        string = s;
    }

    public String getString() {
        return string;
    }

    public void setString(String string) {
        this.string = string;
    }

}
```

Developing a Custom Serializer Implementation for a Java Type Value Class

To use a Java type value class in a Web service implementation and to allow an XML schema type to represent the class in a WSDL, you must provide an implementation of `SOAPElementSerializer` for the Java type value class. The implementation will handle the serialization and deserialization between the Java type value class and the XML schema type.

Example 7–2 illustrates an implementation of `SOAPElementSerializer` for the `MyDate` value type class described in "Defining a Custom Java Type Value Class" on page 7-3. The implementation will allow `MyDate` to be used in the Web service

endpoint interface and to handle serialization and deserialization between `MyDate` and `xsd:dateTime`. It will also allow the `xsd:dateTime` schema type to represent the `MyDate` class in the generated WSDL.

Example 7–2 Sample SOAPElementSerializer Implementation

```
package oracle.demo.custom_type;
import oracle.webservices.databinding.SOAPElementSerializer;
import javax.xml.soap.SOAPFactory;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.Text;
import javax.xml.namespace.QName;
import java.util.Iterator;
import java.util.Map;
public class MyDateSerializer implements SOAPElementSerializer {
    private Map initParams;
    protected SOAPFactory soapFactory;
    public MyDateSerializer() {
    }
    public void init(Map prop) {
        initParams = prop;
        try {
            soapFactory = SOAPFactory.newInstance();
        } catch (SOAPException e) {
            e.printStackTrace();
        }
    }
    public Object deserialize(SOAPElement ele) {
        String str = "";
        for( Iterator i = ele.getChildElements(); i.hasNext();) {
            Object obj = i.next();
            if( obj instanceof Text ) {
                str += ( ((Text)obj).getValue() );
            }
        }
    return new MyDate(str);
    }
    /**
    * The qname parameter must be used to create the top level element.
    */
    public SOAPElement serialize(QName qname, Object obj) {
        SOAPElement xml = null;
        try {
            xml = soapFactory.createElement(
            qname.getLocalPart(),
            qname.getPrefix(),
            qname.getNamespaceURI());
            xml.addTextNode( ((MyDate)obj).getString() );
        } catch (SOAPException e) {
            e.printStackTrace();
        }
        return xml;
    }
}
```

Developing a Service Endpoint Interface that Uses a Java Type Value Class

Develop a Java service endpoint interface that will be exposed as the Web service. Example 7–3 and Example 7–4 illustrate a Java service endpoint interface and implementation that use the `oracle.demo.custom_type.MyDate` class.

Note that to satisfy JAX-RPC requirements, the interface extends `java.rmi.Remote` and the methods throw `java.rmi.RemoteException`.

The `MyDateServiceEndpoint` example also employs a user-defined bean type, `MyDateBean`, as a parameter. `MyDateBean` has two properties of `MyDate`: `beginDate` and `endDate`. WebServicesAssembler will generate the schema for `MyDateBean` because `MyDate` is configured to use the custom serializer and `MyDateBean` conforms to the JAX-RPC Value Type (Bean) patterns. Example 7–5 illustrates `MyDateBean`.

Example 7–3 Sample Java Service Endpoint Interface

```
package oracle.demo.custom_type;

import oracle.demo.custom_type.MyDate;

public interface MyDateServiceEndpoint extends java.rmi.Remote {

    public MyDate echoMyDate(MyDate date)
            throws java.rmi.RemoteException;


    public MyDateBean echoMyDateBean(MyDateBean bean)
            throws java.rmi.RemoteException;

}
```

Similarly, note that new methods defined in the implementation illustrated in Example 7–4 throw a `java.rmi.RemoteException`.

Example 7–4 Sample Implementation of a Java Service Endpoint Interface

```
package oracle.demo.custom_type;

public class MyDateServiceEndpointImpl implements MyDateServiceEndpoint {

    public MyDate echoMyDate(MyDate date) throws java.rmi.RemoteException {
        return date;
    }

    public MyDateBean echoMyDateBean(MyDateBean bean) throws
               java.rmi.RemoteException {
        return bean;
    }

}
```

Example 7–5 illustrates a user-defined bean, `MyDateBean`.

Example 7–5 Sample User-Defined Bean Type

```
package oracle.demo.custom_type;

public class MyDateBean implements java.io.Serializable{
```

```
    private MyDate beginDate;
    private MyDate endDate;

    public MyDate getBeginDate() {
        return beginDate;
    }

    public void setBeginDate(MyDate beginDate) {
        this.beginDate = beginDate;
    }

    public MyDate getEndDate() {
        return endDate;
    }

    public void setEndDate(MyDate endDate) {
        this.endDate = endDate;
    }
}
```

Creating an oracle-webservices Type Mapping Configuration

Create an oracle-webservices type mapping configuration XML file to specify the XML schema type that will be used to represent the custom Java type value class. The contents of the `<type-mapping>` element specify the XML schema type, the custom Java type value class, and the class that performs the custom serialization. Table 7–1 describes the contents of the `<type-mapping>` element.

Table 7–1 Contents of the <type-mapping> Element

Attribute or Element Name	Description
java-class	Attribute that identifies the Java type value class
xml-type	Attribute that identifies the XML schema type
serializer-class	Element that identifies the class that performs the custom serialization

Use the `ddFileName` argument to specify the oracle-webservices type mapping configuration to WebServicesAssembler.

The schema files `oracle-webservices-10_0.xsd` and `oracle-webservices-types-10_0.xsd` can be found in the `oc4j-schemas.jar` file in the *`OC4J_HOME`*`/j2ee/home/lib/` directory.

Example 7–6 illustrates an oracle-webservices type mapping configuration that tells WebServicesAssembler to map the `oracle.demo.custom_type.MyDate` class to `xsd:dateTime`. The `<serializer-class...>` element identifies the `MyDateSerializer` class developed in Example 7–2.

Example 7–6 Sample oracle-webservices Type Mapping Configuration

```
<oracle-webservices>
    <webservice-description name="MyDateService">
        <type-mappings xmlns:xsd="http://www.w3.org/2001/XMLSchema">
            <type-mapping
                java-class="oracle.demo.custom_type.MyDate"
                xml-type="xsd:dateTime">
                <serializer class="oracle.demo.custom_type.MyDateSerializer"/>
            </type-mapping>
        </type-mappings>
```

```
    </webservice-description>
</oracle-webservices>
```

Using Custom Types in Client-Side Proxy Code

By default, WebServicesAssembler does not configure custom types when generating the Web service client-side proxy. The generated proxy will use the default JAX-RPC Java value type-to-XML schema type mappings. For the `MyDate` example presented in this section, this means that the default Java value type, `java.util.Calendar`, will be used to represent `xsd:dateTime`.

To configure custom types into your generated client proxy code, use the `ddFileName` and `classpath` arguments to `genProxy` to specify the oracle-webservices type mapping configuration XML file. The classpath must include the value types and custom serializer. This will allow you to use the custom Java classes in your client-side proxy code. For the example in this section, providing the oracle-webservices type mapping configuration and its classpath when you generate the client-side proxy code enables you to use `MyDate` instances for `xsd:dateTime` in the Web service client-side proxy.

Example 7–7 assumes that the client-side proxy was generated with an oracle-webservices type mapping configuration XML file. Note the use of instances of `MyDate` instead of the default mapping, `Calendar`.

Example 7–7 Client-Side Proxy Code, Using Custom Java Type Values

```
ServiceFactory factory = ServiceFactory.newInstance();
        MyDateService service =  (MyDateService)factory.loadService(MyDateService.
class);
        MyDateServiceEndpoint proxy = service.getMyDateServiceEndpointPort();
        ((Stub)proxy)._setProperty( Stub.ENDPOINT_ADDRESS_PROPERTY, address );
        MyDate req = new MyDate("Year:2004, Month:5, Date:24");
        MyDate res = proxy.echoMyDate( req );

        MyDate begin = new java.util.GregorianCalendar(1977, 5, 24);
        MyDate end = new java.util.GregorianCalendar(2004, 5, 24);
        ...
```

Using Custom Serialization in Top Down Web Service Development

You can use the custom serializer to provide special Java value type-to-XML schema type mappings for a custom Java value type that you want to use in the service endpoint interface. This means that you can substitute a custom Java value type for a type that has a defined, default JAX-RPC mapping in the WSDL. WebServicesAssembler will process the custom Java value type and generate the correct code for the interface.

For example, you can change the default mapping of the schema-defined XML type `xsd:dateTime` from `java.util.Calendar`, to a custom Java value type class, `hello.MyDate`. The custom serializer will allow you to generate the correct code for the service endpoint interface.

Other than providing the custom Java value type class, the `SOAPElementSerializer` implementation, and the type mapping configuration file, this is very similar to the generic top down Web service development.

Developing a Web service that uses a custom serializer follows these general steps.

1. Run WebServicesAssembler with the `genInterface` command and the custom serializer to generate the service endpoint interface with the custom type mappings.
2. Implement the service endpoint interface.
3. Run WebServicesAssembler again with the `topdownAssemble` command to assemble the Web service and package it in an EAR.

The following sections describe these steps in greater detail.

Prerequisites

Before you begin, provide the following files and information.

- the custom Java value type class(es). For more information on custom Java value type classes, see, "Defining a Custom Java Type Value Class" on page 7-3.
- the implementation of the `SOAPElementSerializer` interface for the custom Java value type class(es). For more information on implementing the `SOAPElementSerializer` interface, see "Developing a Custom Serializer Implementation for a Java Type Value Class" on page 7-3.
- the WSDL that employs the custom Java value class(es).
- the oracle-webservices type mapping configuration XML file

  For more information on creating the oracle-webservices type mapping configuration, see "Creating an oracle-webservices Type Mapping Configuration" on page 7-6.

How to Use Custom Serialization in Top Down Web Service Development

The following steps describe how to use custom serialization in top down Web service development.

1. Provide the files described in the Prerequisites section as input to the WebServicesAssembler genInterface command. For example:

   ```
   java -jar wsa.jar -genInterface
                    -ddFileName myOracleWSDescriptor.xml
                    -output build/src/service
                    -wsdl wsdl/MyWSDL.wsdl
                    -unwrapParameters false
                    -packageName oracle.demo.customdoclit.service
   ```

   The major difference between using a custom serializer and standard top down Web services development, is the presence of the `ddFileName` argument. WebServicesAssembler will use the mappings specified in `myOracleWSDescriptor.xml` to generate the service endpoint interface.

   The WebServicesAssembler tool generates a Java interface for every port type specified in the WSDL and a Java Bean for each complex type. The `myOracleWSDescriptor.xml` provides the custom mappings that are specified.

   The name of the directory that stores the generated interface is based on the values of the `output` and `packageName` arguments. For this example, the generated interface is stored in `build/src/service/oracle/demo/customdoclit/service`.

2. Compile the generated interface and type classes.

   For example:

```
javac -destdir ${service.classes.dir}
      -excludes {javac.excludes}
      -path build/src/service
      -classpath ${wsdemo.common.class.path}
```

If you have set your CLASSPATH to include the referenced libraries described in "Web Service Client APIs and JARs" in the *Oracle Application Server Web Services Developer's Guide*, you can omit the -classpath argument. Otherwise, ensure that all of these libraries are contained in the path represented by ${wsdemo.common.class.path}. If you are running an Ant example, the appropriate settings are made in the Ant build script.

3. Implement the service endpoint interface.

   The implementation must have a method signature that matches every method in the generated Java interface.

4. Compile the service implementation.

   For example, you can use the same command as in Step 2 if the interface source was generated in the same directory where the Impl class was saved. If it was not, you must change the value of the path argument.

5. Provide the service endpoint implementation, the WSDL, the custom Java value type class(es), the SOAPElementSerializer implementation(s), and the type mapping configuration XML file as input to the WebServicesAssembler topdownAssemble command.

```
java -jar wsa.jar - topDownAssemble
                  -ddFileName myOracleWSDescriptor.xml
                  -output ./build
                  -wsdl wsdl/MyWSDL.wsdl
                  -unwrapParameters false
                  -packageName oracle.demo.customdoclit.service
                  -className oracle.demo.customdoclit.service.MyEndpointImpl
                  -appName MyWebService
                  -ear ./build/MyWebService.ear
```

   WebServicesAssembler generates and packages a deployable EAR file.

6. Use WebServicesAssembler to generate the Web services client-side proxy.

   - generate stubs (client proxies) for a J2SE Web service client by running the WebServicesAssembler tool with the genProxy command, or
   - generate a service endpoint interface and a JAX-RPC mapping file for a J2EE Web service client by running the WebServicesAssembler tool with the genInterface command.

   For more information on generating and assembling client-side code, see "Assembling a J2EE Web Service Client" and "Assembling a J2SE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.

   To use MyDate to represent xsd:dateTime in your client-side proxy code, you must use ddFileName to specify the oracle-webservices type mapping configuration XML file and classpath to provide the path to the custom Java class MyDate.

   For example, the following command generates client proxies (stubs) that can be used for a J2SE client.

```
java -jar wsa.jar -genProxy
    -wsdl http://localhost:8888/custom_type_mydate/custom_type_mydate?WSDL
```

```
    -packageName oracle.demo.custom_type.mydate_with_ser
    -output ./build/src/client
    -ddFileName MyClientSideDD.xml
    -classpath ./build/classes/service
```

For information on coding a client-side proxy to use custom types, see "Using Custom Types in Client-Side Proxy Code" on page 7-7.

Using Custom Serialization in Bottom Up Web Service Development

You can use the custom serializer to provide special Java value type-to-XML schema type mappings of some specific Java value types used in the service endpoint interface. This means that you can use your own custom schema document which describes special Java value types-to-XML schema type mappings. When you provide special Java value type-to-XML schema type mappings, you do not want WebServicesAssembler to generate schema definitions for these types.

Because this is bottom up Web service development, WebServicesAssembler will generate a WSDL. In the WSDL, instead of generating the schema for the value types, WebServicesAssembler will import one or more schema documents you specify.

Prerequisites

Before you begin, provide the following files and information.

- the custom Java value type class(es). These classes must reside in the classpath. For more information on custom Java value type classes, see, "Defining a Custom Java Type Value Class" on page 7-3.
- the schema document that describes the custom Java value type(s)
- the implementation of the `SOAPElementSerializer` interface for the custom Java value type class(es). For more information on implementing the `SOAPElementSerializer` interface, see "Developing a Custom Serializer Implementation for a Java Type Value Class" on page 7-3.
- the Java service endpoint interface that uses special custom Java value type class(es). For more information on Service Implementation Endpoints, see "Developing a Service Endpoint Interface that Uses a Java Type Value Class" on page 7-5.
- the oracle-webservices type mapping configuration XML file.

  For more information on creating the oracle-webservices type mapping configuration file, see "Creating an oracle-webservices Type Mapping Configuration" on page 7-6.

How to Use Custom Serialization in Bottom Up Web Service Development

The following instructions use sample code to develop a Web service bottom up, that requires custom type mapping.

1. Provide the files described in the Prerequisites section as input to WebServicesAssembler `assemble` command. For example:

```
java -jar wsa.jar -assemble
    -targetNamespace http://oracle.com/ws_demo/custom_type_mydate
    -typeNamespace http://oracle.com/ws_demo/custom_type_mydate
    -input ./build/classes/service
    -classpath ./build/classes/service
    -output ./dist
    -ddFileName ./MyOracleWSDescriptor.xml
```

```
    -interfaceName oracle.demo.custom_type.MyDateServiceEndpoint
    -className oracle.demo.custom_type.MyDateServiceEndpointImpl
    -serviceName MyDateService
    -appName custom_type_mydate
```

This command is similar to the standard bottom up Web services development except that the `ddFileName` argument is used to specify the oracle-webservices type mapping configuration XML file. The `input` argument contains the custom serializer implementation.

WebServicesAssembler generates the WSDL and packages a deployable EAR file.

2. Use WebServicesAssembler to generate the Web service client-side proxy. To use `MyDate` to represent `xsd:dateTime` in your client-side proxy code, you must use `ddFileName` to specify the oracle-webservices type mapping configuration file and `classpath` to provide the path to the custom Java class `MyDate`.

```
java -jar wsa.jar -genProxy
    -wsdl http://localhost:8888/custom_type_mydate/custom_type_mydate?WSDL
    -packageName oracle.demo.custom_type.mydate_with_ser
    -output ./build/src/client
    -ddFileName ./MyClientSideDD.xml
    -classpath ./build/classes/service
```

For information on coding a client-side proxy to use custom types, see "Using Custom Types in Client-Side Proxy Code" on page 7-7.

Ant Tasks for Generating a Web Service

This release provides Ant tasks for Web service development. The following examples show how the WebServicesAssembler commands in the preceding examples can be rewritten as Ant tasks.

For the `assemble` command, here is an example Ant task.

```
<oracle:assemble targetNamespace="http://oracle.com/ws_demo/custom_type_mydate"
    typeNamespace="http://oracle.com/ws_demo/custom_type_mydate"
    input="./build/classes/service"
    classpath="./build/classes/service"
    output="./dist"
    ddFileName="./custom_type_mydate_pdd.xml"
    serviceName="MyDateService"
    appName="custom_type_mydate"
    >
        <oracle:porttype
            interfaceName="oracle.demo.custom_type.MyDateServiceEndpoint"
            className="oracle.demo.custom_type.MyDateServiceEndpointImpl">
        </oracle:porttype>
</oracle:assemble>
```

For the `genProxy` command, here is an example Ant task.

```
<oracle:genProxy
          wsdl="http://localhost:8888/custom_type_mydate/custom_type_mydate?WSDL"
          packageName="oracle.demo.custom_type.mydate_with_ser"
          output="./build/src/client"
          ddFileName="./MyOracleClientDD.xml"
          classpath="./build/classes/client"
           />
```

Using Custom Serialization in Schema-Driven Web Service Development

In schema-driven Web service development, you invoke WebServicesAssembler with the schema document to generate the Java value type classes (beans) before you assemble the Web service.

Because the Java value classes generated from a schema complex type conform to the JAX-RPC value type (bean) pattern requirement, a custom serializer is not necessary. If you are satisfied with the default mapping provided by JAX-RPC, you do not need to implement the a custom serializer.

However, if you want to change the default marshaling between XML and Java, then you must implement the custom serializer for the generated Java value type classes.

Assembling a Web service from a schema follows these general steps.

1. Run WebServicesAssembler with the schema document to generate the Java Value Type classes, the standard JAX-RPC mapping file, and the oracle-webservices type mapping configuration XML file (`custom-type-mappings.xml`).
2. Develop your own service endpoint interface and its implementation.
3. If you are not happy with the default marshalling for one or more classes, write a custom serializer to implement your own marshalling logic.
4. Edit the oracle-webservices type mapping configuration XML file to include the custom serializer. You can also substitute an alternative Java class for any of the generated Java value type classes.
5. Run WebServicesAssembler again, this time, to generate a WSDL document. Input the JAX-RPC mapping file, the edited oracle-webservices type mapping configuration XML file, the schema document, the generated Java value types, and the service endpoint interface and its implementation. Also include the custom serializer, if one was created.

   The generated WSDL will import the specified schema instead of generating one from the Java value types.

WebServicesAssembler will assemble a deployable EAR with a generated WSDL and all of the implementation files and deployment descriptors to enable a Web service.

Prerequisites

Before you begin, provide the following files and information.

- the schema document. "Sample Schema Document" on page 7-15 lists a schema document that defines types can be exposed through a service endpoint interface.
- the definition of a custom Java type that will represent the complex type in the schema.
- if you are providing custom default marshalling logic, you must also provide a custom serializer to implement your own marshalling logic.

Schema-Driven Web Services Assembly with Custom Serialization

This section describes how to develop a Web service from a XML schema that contains complex types. It also describes how to provide a custom serializer to handle serialization and deserialization between the complex type and a Java custom type.

"Sample Schema Document" on page 7-15 describes the schema used in this example. The schema contains two complex types: `InitItem` and `StringItem`.

The section assumes that you want to use the custom Java type, `oracle.webservices.examples.customtypes.MyInitItem`, described in "Java Custom Type Implementation" on page 7-15, to represent the `InitItem` complex type in the schema. To do this, you must provide a custom serializer to handle serialization and deserialization between the Java instance of `MyInitItem` and XML instance of `InitItem`.

For the complex schema type `StringItem`, the section assumes that you want to use the default JAX-RPC value type data binding mechanism to generate a value type class to represent the complex type. It also assumes that you want to do custom formatting on the XML instances of `xsd:string`. To do this, a custom serializer will be created to handle serialization and deserialization between Java instance of `java.lang.String` and the XML instance of `xsd:string`.

The following steps describe how to develop this Web service.

1. Run WebServicesAssembler with the schema document to generate the Java value types and the standard JAX-RPC mapping file.

   ```
   java -jar wsa.jar -genValueTypes
                     -packageName oracle.webservices.examples.customtypes
                     -schema ./MySchema.xsd
                     -output ./build/src/types
   ```

   The schema document used in this command is described in "Sample Schema Document" on page 7-15. The `genValueTypes` command generates two classes from this schema `InitItem` and `StringItem`. The `InitItem` class is not needed since you will be replacing it with your custom type class.

   This command also generates a JAX-RPC mapping file, `jaxrpc-mappings.xml`, which describes the binding of the generated value type classes, and a `custom-type-mappings.xml` file to record which value type classes are generated from an existing schema file. The `custom-type-mappings.xml` file is a generated server-side instance of the OracleAS Web Services type mapping configuration XML file.

2. Compile the generated value types to the `/build/classes/service` directory.

   The `genValueTypes` command does not compile any classes. You must perform this step yourself. Note that you can either delete the `IntItem` class or exclude it from your compilation. This is because you will be providing a custom serializer for it.

3. Implement the custom serializers. In this example, two custom serializers are required: a serializer to map between `oracle.webservices.examples.customtypes.MyIntItem` and `InitItem` and a serializer to map between `java.lang.String` and `xsd:string`.

   "Defining a Serializer Implementation with Marshalling Logic" on page 7-16 contains a sample implementation of these serializers.

4. Develop the Service Endpoint interface and its implementation.

   "Developing a Service Endpoint Interface and Implementation" on page 7-19 provides a sample implementation of a service endpoint interface that uses `oracle.webservices.examples.customtypes.MyInitItem` and the generated Java value type classes.

5. Edit the `custom-type-mappings.xml` file to include an alternative Java class and to include the custom serializer. In Step 1, the `custom-type-mappings.xml` file was generated into the output directory `/build/src/types`.

Edit the file to use `oracle.webservices.examples.customtypes.MyInitItem` and its custom serializer, and to use the `oracle.webservices.examples.customtypes.MyStringSerializer` custom serializer. For details about adding the alternative Java class and custom serializers, see "Editing the Generated oracle-webservices Type Mapping Configuration XML File" on page 7-21.

6. Run the WebServicesAssembler tool again, this time with the `assemble` command, to generate the WSDL and a deployable EAR file. Use the following items as input.

   - the XML schema document (`MySchema.xsd`)
   - the value type classes generated in Step 1 (contained in the `build/classes.service` directory)
   - the JAX-RPC mapping file (`jaxrpc-mappings.xml`) generated in Step 1
   - the compiled custom serializer classes created in Step 2 (contained in the `build/classes.service` directory)
   - the service endpoint interface and its implementation (`MyEndpointIntf` and `MyEndpointImpl`) created in Step 3
   - the modified custom type mapping file `MyCustomTypeMappings.xml` created in Step 4

   Following is a sample `assemble` command.

```
java -jar wsa.jar -assemble
          -schema ./MySchema.xsd
          -mappingFileName ./build/src/types/jaxrpc-mappings.xml
          -input ./build/classes/service
          -classpath ./build/classes/service
          -output ./dist
          -ddFileName ./MyCustomTypeMappings.xml
          -interfaceName oracle.webservices.examples.customtypes.MyEndpointIntf
          -className oracle.webservices.examples.customtypes.MyEndpointImpl
          -appName MyCustomTypes
```

   The input directory, `build/classes/service`, contains the compiled value type classes generated in Step 1 and the compiled custom serializer files created in Step 2. The class name of each custom serializer is listed in the oracle-webservices type mapping file `MyCustomTypeMappings.xml`. Use the `ddFileName` argument to specify this file.

   The WebServicesAssembler tool assembles a deployable EAR file in the `/dist` directory. The generated WSDL in the EAR file imports the schema document.

   Note that this command is similar to normal Java-to-WSDL, or bottom up Web services assembly.

7. Deploy the service and bind the application.

   Deploy EAR files in the standard manner into a running instance of OC4J. For more information on deploying EAR files, see the *Oracle Containers for J2EE Deployment Guide*. The following is a sample deployment command:

```
java -jar <OC4J_HOME>/j2ee/home/admin_client.jar deployer:oc4j:localhost:port
<user> <password>
             -deploy
             -file ./dist/MyCustomTypes.ear
             -deploymentName MyCustomTypes
             -bindWebApp default-web-site
```

8. Develop the client.

   - generate stubs (client proxies) for a J2SE Web service client by running the WebServicesAssembler tool with the `genProxy` command, or
   - generate a service endpoint interface and a JAX-RPC mapping file for a J2EE Web service client by running the WebServicesAssembler tool with the `genInterface` command.

   For an example of developing a J2SE client-side proxy that uses `oracle.webservices.examples.customtypes.MyInitItem` to represent `InitItem` and the custom serializer to handle the marshalling and unmarshaling of data between the client and server, see "Developing a Client for Custom Type Mapping and a Custom Serializer" on page 7-23.

Sample Schema Document

Example 7–8 illustrates the `MySchema.xsd` schema document that provides types that will be exposed as a service endpoint interface. The `InitItem` and `StringItem` are complex types.

Example 7–8 Sample Schema Document

```
<schema
   targetNamespace="http://examples.webservices.oracle/customtypes"
   xmlns:tns="http://ws.oracle.com/types"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns="http://www.w3.org/2001/XMLSchema">

   <complexType name="IntItem">
      <sequence>
         <element name="name" type="xsd:string"/>
         <element name="value" type="xsd:int"/>
      </sequence>
   </complexType>

   <complexType name="StringItem">
      <sequence>
         <element name="name" type="xsd:string"/>
         <element name="value" type="xsd:string"/>
      </sequence>
   </complexType>
</schema>
```

Java Custom Type Implementation

Example 7–9 illustrates an implementation of the Java custom type `oracle.webservices.examples.customtypes.MyIntItem`. This type will be used to represent the `InitItem` complexType illustrated in Example 7–8.

Example 7–9 Implementation of the Custom Java Type InitItem

```
/**
 * MyIntItem does not have the JAXRPC Value type (Bean) pattern.
 */
package oracle.webservices.examples.customtypes;
public class MyIntItem {
    private String name;
    private int value;
    public MyIntItem(String name, int value) {
```

```
        this.name = name;
        this.value = value;
    }
    public String itemName() {
        return name;
    }
    public int itemValue() {
        return value;
    }
}
```

Implementing a Serializer with Custom Marshalling Logic

This section provides examples of the files you must provide and the tasks you must perform if you want to implement a custom serializer that contains custom marshalling logic.

- Defining a Serializer Implementation with Marshalling Logic
- Developing a Service Endpoint Interface and Implementation
- Editing the Generated oracle-webservices Type Mapping Configuration XML File

Defining a Serializer Implementation with Marshalling Logic

This section provides an example of a serializer implementation that contains custom marshalling and unmarshalling logic. It assumes that you want to use the `oracle.webservices.examples.customtypes.MyInitItem` custom Java data type in your Web service to represent the `InitItem` complex type in the schema document. It also assumes that you want to provide custom marshalling and unmarshaling logic for the class. To provide the custom logic, you must implement a custom serializer.

Example 7–10 provides a sample custom serializer implementation for `MyInitItem`. To provide serialization capabilities, the custom serializer, `MyInitItemSerializer`, implements the `SOAPElementSerializer` class and implements its `serialize` and `deserialize` methods.

Example 7–10 Custom Serializer Implementation to Map MyInitItem to InitItem

```
package oracle.webservices.examples.customtypes;

import java.util.Iterator;
import java.util.Map;

import javax.xml.namespace.QName;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPFactory;
import javax.xml.soap.Text;

import oracle.webservices.databinding.SOAPElementSerializer;

public class MyIntItemSerializer implements SOAPElementSerializer {

    private Map config;
    private SOAPFactory soapFactory;
    public void init(Map initParams) {
        config = initParams;
        try {
            soapFactory = SOAPFactory.newInstance();
        } catch (SOAPException e) {
```

```
            e.printStackTrace();
        }
    }

    public SOAPElement serialize(QName tagName, Object object) {
        MyIntItem myIntItem = (MyIntItem)object;
        SOAPElement xml = null;
        try {
            xml = soapFactory.createElement(
                  tagName.getLocalPart(), tagName.getPrefix(), tagName.
getNamespaceURI());

            SOAPElement name = soapFactory.createElement(
                  "name", tagName.getPrefix(), tagName.getNamespaceURI());
            name.addTextNode(myIntItem.itemName());

            SOAPElement value = soapFactory.createElement(
                  "value", tagName.getPrefix(), tagName.getNamespaceURI());
            value.addTextNode(String.valueOf(myIntItem.itemValue()));

            xml.addChildElement(name);
            xml.addChildElement(value);
        } catch (SOAPException e) {
            e.printStackTrace();
        }
        return xml;
    }

    private String getTextContent(SOAPElement element) {
        String str = "";
        for (Iterator i = element.getChildElements(); i.hasNext();) {
            Object obj = i.next();
            if (obj instanceof Text) {
                str += (((Text) obj).getValue());
            }
        }
        return str;
    }

    public Object deserialize(SOAPElement element) {
        String name = null;
        int value = -1;
        for (Iterator i = element.getChildElements(); i.hasNext();) {
            Object obj = i.next();
            if (obj instanceof SOAPElement) {
                SOAPElement child = (SOAPElement)obj;
                if (child.getLocalName().equals("name")) {
                    name = MyStringSerializer.
createMyString(getTextContent(child));
                }
                if (child.getLocalName().equals("value")) {
                    value = Integer.parseInt(getTextContent(child));
                }
            }
        }
        return new MyIntItem(name, value);
    }
}
```

Example 7–11 provides a sample custom serialization implementation to map `java.lang.String` to `xsd:string`. To provide serialization capabilities, the custom serializer, `MyStringSerializer`, implements the `SOAPElementSerializer` class and implements its `serialize` and `deserialize` methods. It also provides a placeholder where you can enter your own string formatting or validation code.

Example 7–11 Custom Serializer Implementation to Map java.lang.String to xsd:string

```
package oracle.webservices.examples.customtypes;

import java.util.Iterator;
import java.util.Map;

import javax.xml.namespace.QName;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPFactory;
import javax.xml.soap.Text;

import oracle.webservices.databinding.SOAPElementSerializer;

public class MyStringSerializer implements SOAPElementSerializer {

    private Map config;
    private SOAPFactory soapFactory;

    public void init(Map initParams) {
        config = initParams;
        try {
            soapFactory = SOAPFactory.newInstance();
        } catch (SOAPException e) {
            e.printStackTrace();
        }
    }

    public SOAPElement serialize(QName tagName, Object object) {
        String value = (object == null) ? "null" : object.toString();
        SOAPElement xml = null;
        try {
            xml = soapFactory.createElement( tagName.getLocalPart(),
                                             tagName.getPrefix(),
                                             tagName.getNamespaceURI());
            xml.addTextNode(value);
        } catch (SOAPException e) {
            e.printStackTrace();
        }
        return xml;
    }

    public Object deserialize(SOAPElement element) {
        String value = "";
        for (Iterator i = element.getChildElements(); i.hasNext();) {
            Object obj = i.next();
            if (obj instanceof Text) {
                value += (((Text) obj).getValue());
            }
        }
        return createMyString( value );
    }
```

```
    static public String createMyString(String input) {
        // You can add your own formating or validation logic here
        return input.trim().toUpperCase();
    }
}
```

Developing a Service Endpoint Interface and Implementation

This section illustrates a service endpoint implementation that uses a different Java data type than the one in the generated schema document.

Example 7–14 provides a sample service endpoint interface, `oracle.webservices.examples.customtypes.MyEndpointIntf`, and Example 7–15 provides its implementation, `oracle.webservices.examples.customtypes.MyEndpointImpl`. Note that the interface extends `java.rmi.Remote` and its methods throw `java.rmi.RemoteException`. Also note that the `getItems` methods in the interface and implementation get items of type `MyInitItem` instead of the `InitItem` value type class generated from the schema document.

Example 7–12 illustrates the generated schema document that defines the `Inititem` and `StringItem` types.

Example 7–12 Schema Definitions of InitItem and StringItem

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<schema
    targetNamespace="http://examples.webservices.oracle/customtypes"
    xmlns:tns="http://ws.oracle.com/types"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.w3.org/2001/XMLSchema">
    <!-- this complex type is mapped to a custom value type using a custom
serializer -->
    <complexType name="IntItem">
        <sequence>
            <element name="name" type="xsd:string"/>
            <element name="value" type="xsd:int"/>
        </sequence>
    </complexType>

    <!—xsd:string is mapped to a custom value type using a custom serializer -->
    <complexType name="StringItem">
        <sequence>
            <element name="name" type="xsd:string"/>
            <element name="value" type="xsd:string"/>
        </sequence>
    </complexType>
</schema>
```

To assemble a Web service that will expose the methods that use `oracle.webservices.examples.customtypes.MyInitItem` data types, you must also replace the `InitItem` entry in the oracle-webservices type mapping configuration XML file with `MyInitItem`. This is described in "Editing the Generated oracle-webservices Type Mapping Configuration XML File" on page 7-21.

Optionally, if you want to provide custom marshalling and unmarshaling for `MyInitItem` data, you must implement a custom serializer for this class. This is described in "Defining a Serializer Implementation with Marshalling Logic" on page 7-16.

Example 7–13 illustrates the definition of the `Items` custom type class. This class will be used in the service endpoint implementation. the items class uses two value types, `MyIntItem` and `StringItem`. The `MyInitItem` class is a custom class that you need to provide a custom serializer. The `StringItem` class is generated from the schema with custom mapping of `String`. Both `MyIntItem` and `StringItem` are described by an existing schema document.

Example 7–13 Definition of the Items Class

```
package oracle.webservices.examples.customtypes;

public class Items {
    private MyIntItem[]  myIntItem;
    private StringItem[] stringItem;
    public Items() {
    }
    public Items(MyIntItem[] intItem, StringItem[] strItem) {
        myIntItem  = intItem;
        stringItem = strItem;
    }
    public MyIntItem[] getMyIntItem() {
        return myIntItem;
    }
    public void setMyIntItem(MyIntItem[] intItem) {
        myIntItem = intItem;
    }
    public StringItem[] getStringItem() {
        return stringItem;
    }
    public void setStringItem(StringItem[] strItem) {
        stringItem = strItem;
    }
}
```

Example 7–14 illustrates the service endpoint interface for `MyEndpointIntf`. Notice that the `getItems` method uses the `MyInitItem` data type instead of the `InitItem` type defined by the schema. As required by JAX-RPC, the interface extends `java.rmi.Remote` and the methods throw `java.rmi.RemoteException`.

Example 7–14 Service Endpoint Interface for MyEndpointIntf

```
package oracle.webservices.examples.customtypes;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface MyEndpointIntf extends Remote {
    public Items getItems( MyIntItem[] myInts, StringItem[] myStrings )
        throws RemoteException;
}
```

Example 7–15 illustrates the implementation of the `MyEndpointIntf` interface.

Example 7–15 Service Endpoint Implementation

```
package oracle.webservices.examples.customtypes;

public class MyEndpointImpl {

    public Items getItems( MyIntItem[] myInts, StringItem[] myStrings ) {
```

```
        Items items = new Items();
        items.setMyIntItem(myInts);
        items.setStringItem(myStrings);
        return items;
    }
}
```

Editing the Generated oracle-webservices Type Mapping Configuration XML File

The `genValueTypes` command creates a generated instance of the oracle-webservices type mapping configuration XML file, `custom-type-mappings.xml`. This file records the Java value type classes that are generated from an existing schema document. When you generate a WSDL from a service endpoint interface, you can use this file as input to WebServicesAssembler to avoid regenerating the schema definition for these pre-generated classes.

If you want to use an alternative Java class to represent a complex type in the schema instead of the class generated from the schema document, then you must edit the contents of the appropriate `<type-mapping>` element in the custom type mapping file. You can also edit the element to include a custom serializer for the alternative class.

The sample custom type mapping file in Example 7–16 assumes that the `genValueTypes` command used the schema in Example 7–8 on page 7-15 to generate the Java value types.

Example 7–16 Custom Type Mapping File, with Pre-Generated Value Type Classes

```
<?xml version="1.0" encoding="UTF-8"?>
    <oracle-webservices>
        <webservice-description name="unknown">
            <type-mappings
                xmlns:typens0="http://examples.webservices.oracle/customtypes"
                xmlns:typens2="http://www.w3.org/2001/XMLSchema"
                >
                <type-mapping
                    java-class="oracle.webservices.examples.customtypes.
StringItem"
                    xml-type="typens0:StringItem">
                </type-mapping>
                <type-mapping
                    java-class="oracle.webservices.examples.customtypes.IntItem"
                    xml-type="typens0:IntItem">
                </type-mapping>
                <type-mapping
                    java-class="int"
                    xml-type="typens2:int">
                </type-mapping>
                <type-mapping
                    java-class="java.lang.String"
                    xml-type="typens2:string">
                </type-mapping>
            </type-mappings>
        </webservice-description>
    </oracle-webservices>
```

To use `oracle.webservices.examples.customtypes.MyInitItem` and its custom serializer `oracle.webservices.examples.customtypes.MyInitItemSerializer` instead of the generated `InitItem` class, and to use the custom serializer `oracle.webservices.examples.customtypes.`

MyStringSerializer for java.lang.String, you must edit the custom-type-mappings.xml file.

1. Replace the instance of InitItem.

   a. Replace the instance of InitItem with oracle.webservices.examples.customtypes.MyInitItem in the appropriate <type-mapping> element.

   b. Add a <serializer> element to <type-mapping> to specify the custom serializer class, oracle.webservices.examples.customtypes.MyInitItemSerializer for MyInitItem.

   For example, the original <type-mapping> element in custom-type-mappings.xml:

```
<type-mapping
    java-class="oracle.webservices.examples.customtypes.IntItem"
    xml-type="typens0:IntItem">
</type-mapping>
```

   should now look like this:

```
 <type-mapping
    java-class="oracle.webservices.examples.customtypes.MyIntItem"
    xml-type="typens0:IntItem">
    <serializer class="oracle.webservices.examples.customtypes.
MyInitItemSerializer"/>
</type-mapping>
```

2. Edit the <type-mapping> element for the mapping of java.lang.String to xsd:string. The entry should look like this:

```
<type-mapping>
   java-class="java.lang.String"
   xml-type="xsd:string">
   <serializer class="oracle.webservices.examples.customtypes.
MyStringSerializer"/>
</type-mapping>
```

3. Rename the file MyCustomTypeMappings.xml and copy it to the root directory of the project.

Example 7–17 illustrates the edited version of the type mapping file. The type-mapping element for StringItem is retained in the file in order to avoid regenerating the schema definition for this pre-generated class.

The namespace prefix declaration (xmlns:xsd="http://www3w.org/2001/XMLSchema") is added so that it can be used in the type mapping configuration file. In the edited type mapping configuration file, you must refer to the string schema type defined in the schema namespace (http://www3w.org/2001/XMLSchema). Defining the prefix, xsd, enables you to use xsd:string as the "qualified name" of the schema type string.

Example 7–17 Custom Type Mapping File, with Edited Value Type Classes

```
<?xml version="1.0" encoding="UTF-8"?>
<oracle-webservices>
    <webservice-description name="unknown">
        <type-mappings
            xmlns:typens0="http://examples.webservices.oracle/customtypes"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
            <type-mapping
```

```
                java-class="java.lang.String"
                xml-type="xsd:string">
                <serializer class="oracle.webservices.examples.customtypes.
MyStringSerializer"/>
            </type-mapping>
            <type-mapping
                java-class="oracle.webservices.examples.customtypes.MyIntItem"
                xml-type="typens0:IntItem">
                <serializer class="oracle.webservices.examples.customtypes.
MyIntItemSerializer"/>
            </type-mapping>
            <type-mapping
                java-class="oracle.webservices.examples.customtypes.StringItem"
                xml-type="typens0:StringItem">
            </type-mapping>
        </type-mappings>
    </webservice-description>
</oracle-webservices>
```

Developing a Client for Custom Type Mapping and a Custom Serializer

To generate J2SE client-side proxy files, use the WebServicesAssembler `genProxy` command:

```
java -jar wsa.jar -genProxy
     -wsdl http://localhost:8888/MyCustomTypes/MyCustomTypes?WSDL
     -output ./build/src/client
```

Because WebServicesAssembler does not configure custom types when generating the Web services client-side proxy, the generated proxy will try to generate Value Type (Bean) classes by default to represent the schema types `InitItem` and `StringItem`. Then you can write your client program using the generated proxy code.

Example 7–18 Client Program Using Default Generated Value Types

```
package oracle.webservices.examples.customtypes;
    import javax.xml.rpc.ServiceFactory;
    import javax.xml.rpc.Stub;
    import examples.webservices.oracle.customtypes.*;
    public class MyCustomTypesClient {
        public static final String DEFAULT_URL
 ="http://localhost:8888/MyCustomTypes/MyCustomTypes";

        public static void main(String[] args) throws Exception {
            String address = DEFAULT_URL;
            ServiceFactory factory = ServiceFactory.newInstance();
            MyCustomTypes service = (MyCustomTypes) factory.
loadService(MyCustomTypes.class);
            MyEndpointIntf proxy = (MyEndpointIntf)service.getPort(MyEndpointIntf.
class);
            ((Stub) proxy)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY, address);
            IntItem int1 = new IntItem();
            int1.setName("int1");
            int1.setValue(123);
            IntItem int2 = new IntItem();
            int2.setName("int2");
            int2.setValue(456);
            IntItem[] ints = { int1, int2 };
            StringItem str1 = new StringItem();
            str1.setName("string1");
```

```
            str1.setValue("foo");
            StringItem str2 = new StringItem();
            str2.setName("string2");
            str2.setValue("coo");
            StringItem[] strings = { str1, str2 };
            Items items = proxy.getItems(ints, strings);
            System.out.println("items.getMyIntItem : " + items.getMyIntItem().
length);
            System.out.println("items.getStringItem: " + items.getStringItem().
            length);
        }
}
```

The `MyInitItem` class was used in developing the `Items` class aforementioned. The `Items` type was developed to be a return value of the service endpoint implementation. The `MyInitItemSerializer` and `MyStringTypeSerializer` are used in the OracleAS Web Services runtime. While WebServicesAssembler is assembling the deployable EAR file, the edited Type Mapping Configuration XML File is used to create a deployment descriptor inside the ear so that these custom serializes will be correctly used in the runtime.

How to Use Custom Serializers in Client Code

You can use a custom serializer in the Web service client-side proxy. This enables you to use the custom data types defined for the service in your client code. This section describes how to create a Web service client-side proxy that uses the `MyIntItemSerializer` (described in Example 7–10) to map `IntItem` to `MyIntItem` and `MyStringSerializer` (described in Example 7–11) to map the `string` type. To do this, you must use a client side version of the oracle-webservices type mapping configuration file, `MyCustomTypeMappings.xml` (described in Example 7–17) as input to `genProxy`.

The following steps summarize the process of adding a custom serializer to a client proxy:

1. Provide a client version of the server-side custom type mapping file.

   For more information on this step, see "Editing the Server Side Custom Type Mapping File".

2. Generate the Web service client proxy with the WebServicesAssembler `genProxy` command.

   For more information on this step, see "Generating the Web Service Client Side Proxy" on page 7-25.

3. Write the Web service client.

   For more information on this step, see "Writing a Web Service Client with Custom Datatypes" on page 7-25.

Editing the Server Side Custom Type Mapping File The custom type mapping file used on the client side must conform to the `oracle-webservices-client-10_0.xsd` schema. To provide a custom type mapping file for the client, you can use the service side custom type mapping file as a template. Edit the file to provide the appropriate client side root elements:

- Replace the `<oracle-webservices>` element with `<oracle-webservices-clients>`.

- Replace the `<webservice-description>` element and any attributes it might contain with `<webservice-client>`.

After making these edits, the client side custom type mapping file (in this case, `MyCustomTypeMappings.xml`) should look like the code in Example 7–19.

Example 7–19 Client Side Custom Type Mapping File

```
<?xml version="1.0" encoding="UTF-8"?>
<oracle-webservice-clients>
   <webservice-client>
      <type-mappings xmlns:typens0="http://examples.webservices.
oracle/customtypes"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema">
          <type-mapping
              java-class="java.lang.String"
              xml-type="xsd:string">
              <serializer class="oracle.webservices.examples.customtypes.
MyStringSerializer"/>
          </type-mapping>
          <type-mapping
              java-class="oracle.webservices.examples.customtypes.MyIntItem"
              xml-type="typens0:IntItem">
             <serializer class="oracle.webservices.examples.customtypes.
MyIntItemSerializer"/>
          </type-mapping>
          <type-mapping
              java-class="oracle.webservices.examples.customtypes.StringItem"
              xml-type="typens0:StringItem">
          </type-mapping>
       </type-mappings>
   </webservice-client>
</oracle-webservice-clients>
```

Generating the Web Service Client Side Proxy Generate the Web services client-side proxy with the WebServicesAssembler `genProxy` command. Use the `ddFileName` argument to provide the client side custom type mapping file as input. The `genProxy` command will create client proxy classes will use your serializers.

The following sample `genProxy` command uses the client side custom type mapping file `MyCustomTypeMappings.xml` as input.

```
java -jar wsa.jar -genProxy
     -wsdl http://localhost:8888/MyCustomTypes/MyCustomTypes?WSDL
     -ddFileName ./MyCustomTypeMappings.xml
     -output ./build/src/client
```

Writing a Web Service Client with Custom Datatypes When you write your client program using the generated proxy code, you can call `MyIntItem` instead of `IntItem`. This is illustrates in Example 7–20.

Example 7–20 Client Code Incorporating Custom Type Mapping and Custom Serializer

```
ServiceFactory factory = ServiceFactory.newInstance();
MyCustomTypes service = (MyCustomTypes) factory.loadService(MyCustomTypes.class);
MyEndpointIntf proxy = (MyEndpointIntf)service.getPort(MyEndpointIntf.class);
((Stub) proxy)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY, address);

    MyIntItem int1 = new MyIntItem("int1", 123);
    MyIntItem int2 = new MyIntItem("int2", 456);
    MyIntItem [] ints = { int1, int2 };
```

```
StringItem str1 = new StringItem();
str1.setName("string1");
str1.setValue("foo");
StringItem str2 = new StringItem();
str2.setName("string2");
str2.setValue("coo");
StringItem[] strings = { str1, str2 };

Items items = proxy.getItems(ints, strings);
```

Limitations

See "Custom Serialization of Java Value Types" on page E-10.

Additional Information

For more information on:

- assembling a Web service top down, see "Assembling a Web Service from a WSDL" in the *Oracle Application Server Web Services Developer's Guide*.
- using Java classes to assemble a Web service, see "Assembling a Web Service with Java Classes" in the *Oracle Application Server Web Services Developer's Guide*.
- using EJBs to assemble a Web service, see "Assembling a Web Service with EJBs" in the *Oracle Application Server Web Services Developer's Guide*.
- using JMS topics and destinations to assemble a Web service, see "Assembling Web Services with JMS Destinations" in the *Oracle Application Server Web Services Developer's Guide*.
- using database resources, such as PL/SQL packages, SQL queries, DML statements, Oracle Streams AQ, or server-side Java classes, to assemble a Web service, see "Assembling Database Web Services" in the *Oracle Application Server Web Services Developer's Guide*.
- assembling a J2EE Web Service client, see "Assembling a J2EE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.
- assembling a J2SE Web service client, see "Assembling a J2SE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.
- using WebServicesAssembler commands to assemble Web service artifacts, see "Using WebServicesAssembler" in the *Oracle Application Server Web Services Developer's Guide*.
- JARs that are necessary to compile client code, see "Web Service Client APIs and JARs" in the *Oracle Application Server Web Services Developer's Guide*,
- the contents of the `oracle-webservices.xml` deployment descriptor which contains the Web services management configuration, see "Packaging and Deploying Web Services" in the *Oracle Application Server Web Services Developer's Guide*.

8

Using JMS as a Web Service Transport

This chapter describes how to employ JMS as an alternative transport mechanism for Web Services. It describes these major topics:

- Understanding JMS as a Transport Mechanism
- Setting Up JMS Queues
- Assembling a Web Service Bottom Up that Uses JMS Transport
- Assembling a Web Service Top Down that Uses JMS Transport
- Assembling a Proxy that Uses JMS as a Transport
- Writing Client Code to Support JMS Transport

Understanding JMS as a Transport Mechanism

Building on top of JMS, Oracle Application Server Web Services support two messaging styles: one-way request messaging and two-way request and response messaging. One-way request messaging lets a Web service client unblock when the request message reaches a JMS queue. Two-way request and response messaging blocks a Web service client until the response message is received.

Using JMS as an Alternative to HTTP

Since SOAP is transport-independent and can be bound to any protocol, SOAP over JMS is an alternative messaging mechanism to the standard SOAP over HTTP messaging. Although both serve as a communication channel between a Web service provider and a Web service client, they are very different. When interoperability is the driving factor, use SOAP over HTTP because it is standardized by the Web Service Interoperability (WS-I) organization. When reliability, scalability, and asynchronous messaging are the key factors, consider SOAP over JMS.

JMS Supports Asynchronous and Reliable Messaging

SOAP over JMS ensures reliability because message delivery can be guaranteed. Messages sent by a Web service provider or client can be placed onto a queue and can be stored in persistent storage. In case of a communication failure, the failing message is retrieved from the persistent storage and re-sent until transmission is successful, which is extremely useful in transaction- and data-critical systems. Businesses that use Enterprise Application Integration (EAI) should find SOAP over JMS appealing because it boosts confidence when exchanging critical data between client and server.

Note: Reliability can also be enabled for SOAP messaging over HTTP. For more information, see Chapter 5, "Ensuring Web Service Reliability".

Asynchronous messaging lets a client invoke a service without waiting for the response. Asynchronous invocation can be implemented by both synchronous and asynchronous transport. JMS as an asynchronous transport can provide a correlation mechanism for associating response messages with request messages. It also lets a client query the status of its requests, and retrieve the responses independently. These features, which HTTP lacks, make the implementation of asynchronous invocation easier for both service requestor and service provider.

JMS Permits Scalable Services

Scalability is another advantage with SOAP over JMS. Unlike HTTP, JMS can support high-volume connections, even for services that get tens of thousands of connections per second.

General Steps for Implementing JMS as a Transport Mechanism

Enabling JMS as a transport mechanism follows these general steps.

1. Set up the queues that are to be used for sending and receiving SOAP messages. "Setting Up JMS Queues" provides more information on this step.
2. Generate the Web service, specifying the JMS queue information for the SOAP transport.
   - If generating bottom up, pass a descriptor containing the JMS transport details, or declare the JMS transport parameters on the command line or Ant task. "Assembling a Web Service Bottom Up that Uses JMS Transport" on page 8-4 provides more information on this step.
   - If generating top down, edit the WSDL to add the JMS address. "Assembling a Web Service Top Down that Uses JMS Transport" on page 8-8 provides more information on this step.
3. Generate the proxy (if necessary). When generating the proxy, you must identify the return queue. The return queue is the JMS queue on which you want to receive responses to Web service invocations. You can do this by declaring the return queue on the command line or in Ant tasks.

   "Assembling a Proxy that Uses JMS as a Transport" on page 8-9 provides more information on this step.

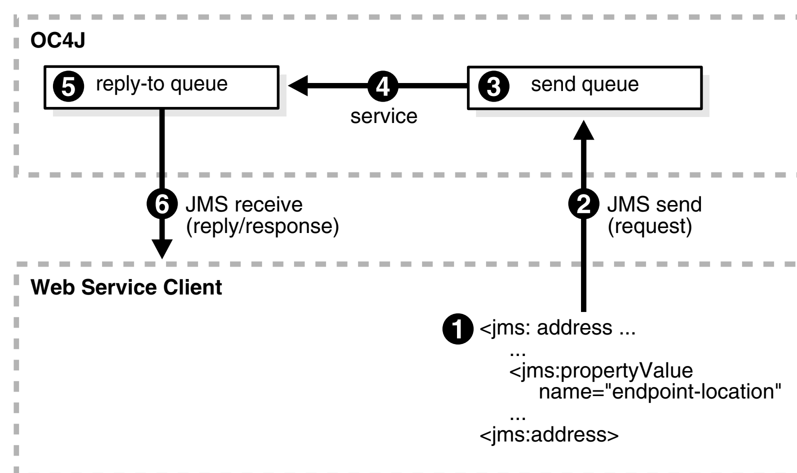
Data Flow for JMS Transport

The following steps describe how data is passed between the Web service client and OracleAS Web Services when JMS is the transport mechanism. The WSDL elements and WebServicesAssembler arguments mentioned in the steps are described later in this chapter. Figure 8–1 illustrates the sequence of steps.

1. In the Web service client, the message is sent as a request to a JMS queue in OC4J. The `<jms:address>` element in the WSDL provides the location of the target endpoint to which the message will be served from the JMS queue in OC4J.
2. The client performs a JMS `send` operation to send the request to the sender queue in OC4J.

3. The sender queue in OC4J accepts the request. If WebServicesAssembler was used to assemble the Web service, then the location of the sender queue is provided by the `sendQueueLocation` argument.
4. The server runtime dequeues the request and serves it to the endpoint identified by the client's request.
5. If the message has a reply (or response), the service puts the message on the reply-to queue. If WebServicesAssembler was used to assemble the Web service, then the location of the reply-to queue is provided by the `replyToQueueLocation` argument.
6. The client blocks on waiting to receive a response on the reply-to queue (in case of a request-response operation), thus simulating a synchronous request-response. If the message exchange pattern was one way, the client does not wait for a response.

Figure 8–1 Data Flow for JMS Transport



Setting Up JMS Queues

Queues must be set up in the OracleAS Web Services container to identify the sender queue and the receiver queue. Each queue is identified by a `<queue>` element and a `<queue-connection-factory>` element.

The `<queue>` element contains a `name` and a `location` attribute. The `name` attribute specifies the queue's JNDI name. The `location` attribute specifies the queue's JNDI location. The `queue` element can also contain an optional `description` element to provide a text description of the queue.

The `queue-connection-factory` element also contains a `name` and a `location` attribute. The `name` attribute specifies the JNDI name of the JMS connection factory used to produce connections for the `send` or `receive` operation. The `location` specifies the factory's JNDI location.

If the queues are set up in the OC4J/JMS `jms.xml` configuration file, you can add two clauses to that file: one for the sender queue and one for the reply-to queue. For more information on JMS in the OC4J environment, see *Oracle Containers for J2EE Services Guide*.

Example 8–1 illustrates XML code that identifies the `SOAP sender` queue at the JNDI location `jms/senderQueue`. This queue uses `jms/senderQueueConnectionFactory` as the JMS connection factory to produce connections for the `send` operation. Similarly, the second clause identifies the `SOAP`

receiver queue at the JNDI location jms/replyToQueue as the receiver queue. This queue uses jms/replyToQueueConnectionFactory as the JMS connection factory to produce connections for the response messages from the Web service endpoint.

Example 8–1 XML Code to Identify Sender Queue and Receiver Queue

```
...
<queue name="SOAP sender" location="jms/senderQueue">
    <description>A queue for SOAP messages</description>
</queue>
<queue-connection-factory
    name="jms/senderQueueConnectionFactory"
    location="jms/senderQueueConnectionFactory"/>

<queue name="SOAP receiver" location="jms/replyToQueue">
    <description>A queue for SOAP response messages</description>
</queue>
<queue-connection-factory
    name="jms/replyToQueueConnectionFactory"
    location="jms/replyToQueueConnectionFactory"/>
...
```

Assembling a Web Service Bottom Up that Uses JMS Transport

To configure JMS transport into a Web service that you are generating bottom up, extensions must be added to the WSDL that will identify a port that has JMS transport enabled. WebServicesAssembler provides two arguments that will add this information to the WSDL.

- sendQueueLocation—identifies the JNDI name of the JMS queue that messages will be sent to.
- sendConnectionFactoryLocation—identifies the JNDI location of the JMS factory used to produce connections for the JMS send operation.

The following steps describe how to generate a Web service bottom up that uses JMS as its transport mechanism. The steps follow the standard procedure for assembling a Web service bottom up. For examples of bottom up Web service assembly, see "Assembling a Web Service with Java Classes" and "Assembling a Web Service with EJBs" in the *Oracle Application Server Web Services Developer's Guide*.

It is assumed that you have already configured the JMS queues for sending and receiving SOAP messages. "Setting Up JMS Queues" on page 8-3 provides more information on queues.

1. Generate the service artifacts by running the WebServicesAssembler with the assemble command. Specify the queue location and queue connection factory with sendQueueLocation and sendConnectionFactoryLocation. The WSDL generated by this Ant task is illustrated in Example 8–2.

```
<oracle:assemble appName="echo"
        targetNamespace="http://www.oracle.com/echo"
        typeNamespace="http://www.oracle.com/echo"
        serviceName="EchoService"
        input="./build/classes/service"
        output="build"
        ear="build/echo.ear"
        style="rpc"
        use="encoded"
```

```
            createOneWayOperations="true"
            >
            <oracle:porttype
                interfaceName="oracle.j2ee.ws.jmstransport.Echo"
                className="oracle.j2ee.ws.jmstransport.EchoImpl"
                 >
                 <oracle:port
                   uri="/echo"
                   sendQueueLocation="jms/senderQueue"
                   name="EchoPort"
                   sendConnectionFactoryLocation="jms/senderQueueConnectionFactory"
                </oracle:port>
            </oracle:porttype>
     />
```

2. Deploy the service.

   Deploy the EAR file in the standard manner into a running instance of OC4J. For more information on deploying EAR files, see the *Oracle Containers for J2EE Deployment Guide*.

3. Generate the client code. This step is described in "Assembling a Proxy that Uses JMS as a Transport" on page 8-9.

WSDL Extensions for JMS Transport

The following WSDL extensions define JMS transport for a Web service.

- JMS Address Element
- JMS Property Value Element

These are Oracle-proprietary WSDL extensions and may not be interoperable. If you are using JDeveloper or WebServicesAssembler to develop and deploy the endpoint bottom up, then these extensions are generated into the WSDL. Otherwise, if you are manually writing the WSDL or developing and deploying the Web service top down, then you should add these extensions.

The following URI is the namespace for the extensions described in this section.

`xmlns:jms="http://www.oracle.com/technology/oracleas/wsdl/jms"`

JMS Address Element

Only one `<jms:address>` element should be used in a `port-component` definition. This element should be used in place of a `<soap:address>` declaration.

```
<jms:address
    jndiDestinationName="xxx"
    jndiConnectionFactoryName="xxx"/>
```

Table 8–1 describes the attributes of the `<jms:address>` element.

Table 8–1 Attributes of the <jms:address> Element

Attribute Name	Description
jndiDestinationName	The JNDI name of the JMS queue that messages will be sent to. If you are generating the Web service bottom up, then this attribute will be set by the `sendQueueLocation` argument.

Table 8–1 (Cont.) Attributes of the <jms:address> Element

Attribute Name	Description
jndiConnectionFactoryName	The JNDI name of the connection factory that will be used. If you are generating the Web service bottom up, then this attribute is set by the `sendConnectionFactoryLocation` argument.

JMS Property Value Element

The `<jms:propertyValue>` element must be added to the `<jms:address>` section of the `binding` operation to identify the endpoint location. The general format is the following.

```
<jms:propertyValue
    name="string"
    type="type"
    value="string"/>
```

Table 8–2 describes the attributes of the `<jms:propertyValue>` element.

Table 8–2 Attributes of the <jms:propertyValue> Element

Attribute Name	Description
name	The name of a property defined by JMS, by the JMS implementation, or by the user.
type	The datatype of the attribute.
value	A value that hard codes the value of this property in the WSDL.

The WebServicesAssembler `uri` argument can provide the value for the `propertyValue` element in the WSDL. When the `uri` argument is specified, the `name` attribute is set to `endpoint-location`, the `type` attribute is set to `string` and the `value` attribute is set to the value of the `uri` argument.

```
<jms:propertyValue name="endpoint-location" type="string" value="value of uri
argument"/>
```

Example 8–2 illustrates the WSDL created by the Ant task in "Assembling a Web Service Bottom Up that Uses JMS Transport" on page 8-4. The JMS transport configuration is highlighted in bold.

Example 8–2 WSDL with a JMS Transport Configuration

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
   xmlns:tns="http://www.oracle.com/echo"
   xmlns:jms="http://www.oracle.com/technology/oracleas/wsdl/jms"
name="Echo" targetNamespace="http://www.oracle.com/echo">
    <types>
        <schema xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:soap11-enc="http://schemas.xmlsoap.org/soap/encoding/"
targetNamespace="http://www.oracle.com/echo"/>
    </types>
```

```
    <message name="Echo_echo">
        <part name="String_1" type="xsd:string"/>
    </message>
    <message name="Echo_echoInt">
        <part name="int_1" type="xsd:int"/>
    </message>
    <message name="Echo_echoIntResponse">
        <part name="result" type="xsd:int"/>
    </message>
    <message name="Echo_echoResponse">
        <part name="result" type="xsd:string"/>
    </message>
    <portType name="Echo">
        <operation name="echo" parameterOrder="String_1">
            <input message="tns:Echo_echo"/>
            <output message="tns:Echo_echoResponse"/>
        </operation>
        <operation name="echoInt" parameterOrder="int_1">
            <input message="tns:Echo_echoInt"/>
            <output message="tns:Echo_echoIntResponse"/>
        </operation>
    </portType>
    <binding name="EchoBinding" type="tns:Echo">
        <operation name="echo">
            <input>
                <soap:body use="encoded" namespace="http://www.oracle.com/echo"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" parts="String_1"/>
            </input>
            <output>
                <soap:body use="encoded" namespace="http://www.oracle.com/echo"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" parts="result"/>
            </output>
            <soap:operation soapAction=""/>
        </operation>
        <operation name="echoInt">
            <input>
                <soap:body use="encoded" namespace="http://www.oracle.com/echo"
 encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" parts="int_1"/>
            </input>
            <output>
                <soap:body use="encoded" namespace="http://www.oracle.com/echo"
 encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" parts="result"/>
            </output>
            <soap:operation soapAction=""/>
        </operation>
        <soap:binding style="rpc"
 transport="http://www.oracle.com/technology/oracleas/wsdl/jms"/>
    </binding>
    <service name="Echo">
        <port name="EchoPort" binding="tns:EchoBinding">
            <jms:address
             jndiConnectionFactoryName="jms/OracleSyndicateQueueConnectionFactory"
    jndiDestinationName="jms/OracleSyndicateQueue" >
                <jms:propertyValue name="endpoint-location" type="string"
value="echo"/>
            </jms:address>
        </port>
    </service>
</definitions>
```

Adding JMS Transport Configuration with Deployment Descriptors

A JMS transport configuration can be passed to the Web service assembly by entering it into the `oracle-webservices.xml` deployment descriptor. This is an alternative to declaring the `sendQueueLocation` and `sendConnectionFactoryLocation`, arguments on the command line or Ant task.

When you enter the configuration into `oracle-webservices.xml`, use `<jms-address>` instead of `<jms:address>` and `<jms-propertyValue>` instead of `<jms:propertyValue>`. These elements should appear as subelements of the `<port-component>` element.

Example 8–3 illustrates an `oracle-webservices.xml` deployment descriptor that includes a JMS transport configuration. During assembly, a new WSDL will be created. The resulting WSDL will look like the one in Example 8–2. The JMS configuration is highlighted in bold. The `jndiDestinationName` and `jndiConnectionFactoryName` attributes are defined in "WSDL Extensions for JMS Transport" on page 8-5.

Example 8–3 oracle-webservices.xml with JMS Transport Configuration

```
<oracle-webservices
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="oracle-webservices-server-10_0.xsd"
    schema-major-version="10" schema-minor-version="0">
    <webservice-description name="Echo">
        <port-component name="EchoPort">
            <runtime enabled="true">
                <jms-address jndiDestinationName="jms/senderQueue"
jndiConnectionFactoryName="jms/senderQueueConnectionFactory" />
            </runtime>
        </port-component>
    </webservice-description>
</oracle-webservices>
```

Assembling a Web Service Top Down that Uses JMS Transport

For top down assembly, you must edit the WSDL to add the JMS address and property values for the endpoint.

The following steps describe how to generate a Web service that uses JMS as its transport mechanism. The steps follow the standard procedure for assembling a Web service described in "Generating the Web Service Top Down" in the *Oracle Application Server Web Services Developer's Guide*.

It is assumed that you have already configured the JMS queues for sending and receiving SOAP messages. "Setting Up JMS Queues" on page 8-3 provides more information on queues.

1. Provide a WSDL from which the Web service will be generated.
2. Edit the WSDL to add the elements that define JMS transport. "WSDL Extensions for JMS Transport" on page 8-5 describes the JMS transport elements.
3. Use the WSDL as input to the WebServicesAssembler `genInterface` command.
4. Compile the generated interface and type classes.
5. Write the Java service endpoint interface for the Web service you want to provide.
6. Compile the Java service endpoint interface.

7. Generate the service by running the WebServicesAssembler tool with the `topDownAssemble` command.

```
<oracle:topDownAssemble appName="bank"
            packageName="oracle.ws.server.bank"
            wsdl="./etc/Bank.wsdl"
            input="./build/classes/service}"
            output="build"
            ear="build/bank.ear"
            debug="true"
            fetchWsdlImports="true"
            >
            <oracle:classpath>
                <oracle:pathelement path="${build.impl.dir}"/>
                <oracle:pathelement location="${wsa.jar}"/>
            </oracle:classpath>
            <oracle:porttype
              className="oracle.ws.server.bank.BankImpl"
              classFileName="java/oracle/ws/server/bank/BankImpl.java"
              >
               <oracle:port name="BankPort" uri="bank2"/>
               <oracle:port name="BankPortWithJMS" uri="/bank"/>
               <oracle:port name="Soap12JmsEchoPort" uri="/soap12bank"/>
          </oracle:porttype>
        />
```

8. Deploy the service.

   Deploy the EAR file in the standard manner into a running instance of OC4J. For more information on deploying EAR files, see the *Oracle Containers for J2EE Deployment Guide*.

9. Generate the client code. This step is described in "Assembling a Proxy that Uses JMS as a Transport".

10. Add the following client JAR files to the classpath. These client JAR files are required when the Web service client supports JMS as a transport mechanism. The *J2EE_HOME* environment variable in the paths represents the location where the Oracle Application Server or the standalone OC4J is installed.

    - *J2EE_HOME*/j2ee/home/lib/oc4j-unsupported-apis.jar
    - *J2EE_HOME*/j2ee/home/lib/oc4j-internal.jar

    "Web Service Client APIs and JARs" in the *Oracle Application Server Web Services Developer's Guide*, provides a description of all of the available OracleAS Web Services client JARs.

Assembling a Proxy that Uses JMS as a Transport

When working with JMS as a transport mechanism you must ensure that the client code provides the JNDI names of the JMS queue and the connection factory for the reply-to queue which will be used for `send` operation JMS messages. The WebServicesAssembler arguments `replyToQueueLocation` and `replyToConnectionFactoryLocation` can provide these values. Used with the `genProxy` command, these arguments generate a proxy stub that can be used with JMS transport. The arguments will generate a stub configured with values for a queue that will receive response messages.

If you do not use the `replyTo*` arguments to generate a proxy stub, and the endpoint contains request/response operations, then you must set the `replyTo*` arguments

programmatically so that response messages can be received. "Setting the Send Queue Location and Connection Factory Programmatically" on page 8-11 describes how to provide this information directly in your code.

The following steps describe how to generate a proxy stub that uses JMS as its transport mechanism. The steps follow the standard procedure for assembling a proxy stub that are described in *How to Assemble a J2SE Web Service Client with a Static Stub* in the *Oracle Application Server Web Services Developer's Guide*,.

It is assumed that you have already configured the JMS queues for sending and receiving SOAP messages. "Setting Up JMS Queues" on page 8-3 provides more information on queues.

1. Provide the URI to the WSDL, the name of the output directory, the package name, and the locations of the JMS queue and connection factory as input to the WebServicesAssembler `genProxy` command.

   ```
   <oracle:genProxy
          wsdl="http://localhost:8888/bank/bank?WSDL"
          output="build/src/proxy"
          packageName="oracle.ws.client.bank"
          >
          <oracle:port
            replyToConnectionFactoryLocation="jms/receiverQueueConnectionFactory"
            replyToQueueLocation="jms/receiverQueue"
          </oracle:port>
      />
   ```

   This command generates the client proxy and stores it in the `build/src/proxy` directory. The client application uses the stub to invoke operations on a remote service.

2. Use the client utility class file created by `genProxy` as your application client, or use it as a template to write your own client code. The client utility class file is one of a number of files created by `genProxy`.

3. Compile the client files and put them in the classpath.

   List the appropriate JARs on the classpath before compiling the client. "Classpath Components for Clients Using a Client Side Proxy" in the *Oracle Application Server Web Services Developer's Guide* lists all of the JAR files that can possibly be used on the client classpath. As an alternative to listing individual JARs, you can include the client-side JAR, `wsclient_extended.jar` on the client classpath. This JAR file contains all the classes necessary to compile and run a Web service client. The classes are from the individual JAR files listed in "Setting the Web Service Proxy Client Classpath" in the *Oracle Application Server Web Services Developer's Guide*. This appendix provides information on `wsclient_extended.jar` and the client classpath.

   For JMS clients, you must also include the `oc4j-internal.jar` and `oc4j-unsupported-apis.jar` files on the classpath. "JMS Transport-Related Client JAR File" in the *Oracle Application Server Web Services Developer's Guide*. provides more information on the `oc4j-internal.jar` and `oc4j-unsupported-apis.jar` files.

   To provide a JNDI configuration for the client, add the `jndi.properties` file to the client classpath.

4. Run the J2SE JMS client from the command line.

Writing Client Code to Support JMS Transport

The following sections describe how to write Web service clients that use JMS transport.

- Writing Client Stub Code for JMS Transport
- Setting the Send Queue Location and Connection Factory Programmatically
- Writing DII Code for JMS Transport

Writing Client Stub Code for JMS Transport

Stub code for a client that uses JMS transport is very similar to code for clients that use HTTP transport. The difference is that the endpoint address must be set to the unique URI that identifies the JMS endpoint. "Writing Web Service Client Applications" in the *Oracle Application Server Web Services Developer's Guide* provides more information on writing client code for J2SE client applications.

Example 8–4 illustrates client stub code that uses JMS transport to send messages. The code that sets the endpoint address to the unique URI that identifies the JMS endpoint is highlighted in bold.

Example 8–4 Client Stub Code to Send Messages with JMS Transport

```
...
ServiceFactory serviceFactory = ServiceFactory.newInstance();
        Echo_Service echoService = (Echo_Service) serviceFactory.loadService(Echo_
Service.class);
        Echo_PortType echoPort = echoService.getEchoPort();
        ((Stub)echoPort)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY,
"/echo/echo");
         String echo = echoPort.echo("Test echo");
...
```

Setting the Send Queue Location and Connection Factory Programmatically

When you are working with JMS as a transport, client stub code requires to you to identify the JNDI names of the JMS queue and the connection factory for the reply-to queue for `send` operation JMS messages. If you do not provide these values when you generate the proxy stub code, you can add them programmatically. OracleAS Web Services provides two properties in the `oracle.webservices.transport.OracleStub` class that allow you to do this.

- `JMS_TRANSPORT_REPLY_TO_FACTORY_NAME`—specifies the JNDI name of the JMS connection factory to be used as the default reply-to of all `send` operation JMS messages. This property is comparable to the `replyToConnectionFactoryLocation` WebServicesAssembler argument.
- `JMS_TRANSPORT_REPLY_TO_QUEUE_NAME`—specifies the JNDI name of the JMS queue to be used as the default reply-to of all `send` operation JMS messages. This property is comparable to the `replyToQueueLocation` WebServicesAssembler argument.

Example 8–5 illustrates client stub code that uses these properties to set the JNDI name of the JMS queue and the connection factory for the reply-to queue for `send` operations. The code that sets the properties is highlighted in bold.

Example 8–5 Setting replyTo* Parameters Programmatically

```
ServiceFactory serviceFactory = ServiceFactory.newInstance();
```

```
   Echo_Service echoService = (Echo_service)serviceFactory.loadService(Echo_
Service.class;
   Echo_PortType echoPort = echoService.getEchoPort();
      ((Stub)echoPort)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY, "/echo/echo");
      ((Stub)echoPort)._setProperty(OracleStub.JMS_TRANSPORT_REPLY_TO_FACTORY_
NAME, "jms/receiverQueueConnectionFactory");
      ((Stub)echoPort)._setProperty(OracleStub.JMS.TRANSPORT_REPLY_TO_QUEUE_NAME,
"jms/receiverQueue");
   String echo = echoPort.echo("Test echo");
```

Writing DII Code for JMS Transport

If you are writing code for a Dynamic Invocation Interface (DII) client, you must create `JmsAddress` objects for the sender queue and the receiver queue. These objects are then used to declare an `JmsClientTransportFactory` object programmatically. This will allow the client to get response messages from a Web service invocation.

The `JmsAddress` and `JmsClientTransportFactory` classes belong to the `oracle.webservices.transport` package.

Example 8–6 illustrates DII client code for sending messages through JMS transport. The code which illustrates how `JmsClientTransportFactory` is created, set up, and used is highlighted in bold.

Example 8–6 DII Client Code to Send Messages Through JMS Transport

```
...
QName operation = new QName("http://www.oracle.com/echo", "echo");
            Call call = getCall(operation, SOAPVersion.SOAP_1_1);
            call.setTargetEndpointAddress("/echo/echo");
            JmsAddress jmsAddress = new JmsAddress("jms/senderQueue",
"jms/senderQueueConnectionFactory");
            JmsAddress replyToAddress = new JmsAddress("jms/receiverQueue",
"jms/receiverQueueConnectionFactory");
            JmsClientTransportFactory transportFactory = new
JmsClientTransportFactory(jmsAddress, replyToAddress);
            ((OracleCall) call).setClientTransportFactory(transportFactory);
...
```

Limitations

See "Using JMS as a Web Service Transport" on page E-11.

Additional Information

For more information on:

- assembling a Web service top down, see "Assembling a Web Service from a WSDL" in the *Oracle Application Server Web Services Developer's Guide*.
- using Java classes to assemble a Web service, see "Assembling a Web Service with Java Classes" in the *Oracle Application Server Web Services Developer's Guide*.
- using EJBs to assemble a Web service, see "Assembling a Web Service with EJBs" in the *Oracle Application Server Web Services Developer's Guide*.
- using JMS topics and destinations to assemble a Web service, see "Assembling Web Services with JMS Destinations" in the *Oracle Application Server Web Services Developer's Guide*.

- using database resources, such as PL/SQL packages, SQL queries, DML statements, Oracle Streams AQ, or server-side Java classes, to assemble a Web service, see "Assembling Database Web Services" in the *Oracle Application Server Web Services Developer's Guide*.
- assembling a J2EE Web Service client, see "Assembling a J2EE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.
- assembling a J2SE Web service client, see "Assembling a J2SE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.
- using WebServicesAssembler commands to assemble Web service artifacts, see "Using WebServicesAssembler" in the *Oracle Application Server Web Services Developer's Guide*.
- JARs that are necessary to compile client code, see "Web Service Client APIs and JARs" in the *Oracle Application Server Web Services Developer's Guide*,
- the contents of the `oracle-webservices.xml` deployment descriptor which contains the Web services management configuration, see "Packaging and Deploying Web Services" in the *Oracle Application Server Web Services Developer's Guide*.

9 Using Web Services Invocation Framework

This chapter describes Oracle Application Server Web Services support for the Web Services Invocation Framework (WSIF). WSIF provides a standard API for working with representations of different Web service messaging protocols as an alternative to working directly with a JAX-RPC SOAP client API. The WSIF API allows the client to use native protocols to communicate with the service.

The WSIF API allows a Web service to communicate with Web service components, such as Java classes, EJBs, or database resources, through a single interface. In addition to JAX-RPC which understands only SOAP, WSIF clients can directly use other protocols, such as RMI, IIOP, or JDBC.

To support different messaging protocols, you can add different WSIF bindings to the WSDL. This is in contrast to the SOAP protocol, where you add a SOAP binding. WSIF and SOAP bindings can co-exist in the same WSDL. This enables you to define both native and SOAP bindings for a particular service. For example, you can expose an EJB with a WSIF port and a SOAP port. Clients can access the same Web service even though they may be using different protocols. The protocol that is used to communicate with the service is determined by selecting a particular binding in the client: either a SOAP or a WSIF binding.

This chapter contains the following sections.

- Understanding WSIF Architecture
- Configuring a WSIF Endpoint for Java Classes
- Configuring a WSIF Endpoint for EJBs
- Configuring a WSIF Endpoint for Database Resources
- Tool Support for WSIF

WSIF is sponsored by the Apache Software Group. For more information on WSIF, see the following Web site.

http://ws.apache.org/wsif/

Understanding WSIF Architecture

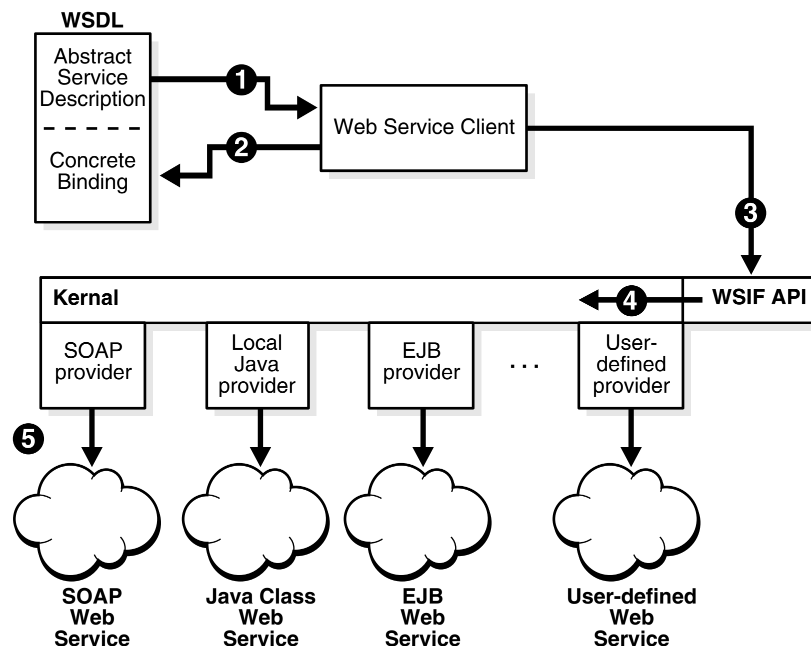
In the WSIF architecture, communications between the client and Web services are enabled through the WSDL. The WSDL becomes the normalized description of the software interface.

Figure 9–1 illustrates the WSIF architecture for OracleAS Web Services. The following steps describe the data flow through the architecture.

1. Program the WSIF client according to the content of the abstract service description portion of the WSDL. This portion of the WSDL includes port types, operations, and message exchanges.
2. At runtime, the WSIF client selects a concrete binding in the WSDL that determines which provider the WSIF runtime will use. Providers are protocol-specific pieces of code that provide implementation of the different WSIF bindings in the WSDL.
3. The WSIF client makes calls into the WSIF API to invoke the Web service. This API is based on the abstract service description portion of the WSDL. By invoking the service based on its abstract description, you can work with a service regardless of how it was implemented, its protocol, or where it resides.
4. The WSIF runtime in the kernal determines which provider to use, based on the binding selected in Step 2.
5. The provider interacts with the Web service.

Services can be invoked dynamically and without the need for generating stubs.

Figure 9–1 WSIF Architecture for OracleAS Web Services



WSIF supports providers for clients of Web services that expose JAX-RPC (SOAP), Java classes, EJBs, and database resources. You can also define your own providers. Defining your own providers is beyond the scope of this manual. For information on this topic, see the Apache Software Group Web page for WSIF.

`http://ws.apache.org/wsif/`

The following sections describe the WebServicesAssembler commands and Ant tasks that insert WSIF information into the WSDL.

Configuring a WSIF Endpoint for Java Classes

To insert WSIF Java extensions into the WSDL, OracleAS Web Services provides two arguments that can be used with the `assemble` or `genWsdl` WebServicesAssembler commands.

- `wsifJavaBinding`
- `wsifJavaPort`

As you assemble a Web service based on a Java class with the `assemble` or `genWsdl` commands, the `wsifJavaBinding` and `wsifJavaPort` arguments allow you to request a WSIF binding. For more information on creating a Web service based on a Java class, see "Assembling a Web Service with Java Classes" in the *Oracle Application Server Web Services Developer's Guide*.

The `wsifJavaBinding` or `wsifJavaPort` arguments add the appropriate WSDL extensions in the form of a WSIF binding that allow a WSIF client to access Java class exposed as a Web service. "WSIF Java Extensions to the WSDL" on page 9-6 lists the schema definitions that can be added to the WSDL to support WSIF Java bindings.

To configure the WSIF Java bindings for a single port, you can use either the `wsifJavaBinding` or `wsifJavaPort` argument. The `wsifJavaPort` argument also enables you to specify multiple ports of different types within an Ant task.

While the `wsifJavaBinding` argument can be used either on the command line or in an Ant task, the `wsifJavaPort` can be used only in an Ant task.

The following sections describe how to configure a WSIF endpoint for single and multiple Java ports.

- Configuring a WSIF Endpoint for a Single Java Port
- Configuring a WSIF Endpoint for Multiple Java Ports

Configuring a WSIF Endpoint for a Single Java Port

A WSIF endpoint for a single Java port can be configured with `wsifJavaBinding` or `wsifJavaPort`.

- Configuring a Single Java Port with wsifJavaBinding
- Configuring a Single Java Port with wsifJavaPort

Configuring a Single Java Port with wsifJavaBinding

The `wsifJavaBinding` argument can be used by either `assemble` or `genWsdl` command to specify the WSIF Java binding for a single port.

The `wsifJavaBinding` argument requires you to specify the `className` argument. The argument passes this value to the `java:address` element in the WSDL's `<port>` element.

Example 9–1 illustrates an WebServicesAssembler command line for `genWsdl` that calls `wsifJavaBinding`. Example 9–2 illustrates the corresponding Ant task. The argument causes the `java:address` element in the WSDL's `<port>` element to be populated with `classname="oracle.demo.hello.HelloImpl"`. The variable `${wsdemo.common.class.path}` represents the classpath.

Example 9–1 Command Line, Using wsifJavaBinding to Configure a Single Java Port

```
java -jar wsa.jar -genWsdl
                  -output wsdl
```

```
-style rpc
-use literal
-wsifJavaBinding true
-interfaceName oracle.demo.hello.HelloInterface
-className oracle.demo.hello.HelloImpl
-classpath ${wsdemo.common.class.path}:build/classes
```

Example 9–2 Ant Task, Using wsifJavaBinding to Configure a Single Java Port

```
<oracle:genWsdl
    output="wsdl"
    style="rpc"
    use="literal"
    wsifJavaBinding="true"
    >
    <oracle:porttype
         interfaceName="oracle.demo.hello.HelloInterface"
         className="oracle.demo.hello.HelloImpl"
    </oracle:porttype>
    <oracle:classpath>
         <oracle:pathelement path="${wsdemo.common.class.path}"/>
         <oracle:pathelement location="build/classes"/>
    </oracle:classpath>
</oracle:genWsdl>
```

Configuring a Single Java Port with wsifJavaPort

The `wsifJavaPort` argument can be used only in an Ant task. Although you typically use `wsifJavaPort` to specify WSIF Java bindings for multiple ports, you can also use it to specify a single port. The argument also has attributes that allow you to specify a WSIF port name, a Java class name for each port, and a custom class loader and its classpath. These attributes are passed to the `java:address` element in the WSDL's `<port>` element.

Table 9–1 describes the attributes that can be used by the `wsifJavaPort` argument. All of the attributes are optional.

Table 9–1 Attributes for the wsifJavaPort Argument

Attribute	Description
classLoader	Specifies the name of the Java class loader. If you do not provide a value for this attribute, the value for the system class loader will be used.
className	Specifies the name of the implementation class. If you do not provide a value for this attribute, WebServicesAssembler will use the class name for the parent Ant task if it was specified.
classPath	Specifies the class path for the Java class loader. If you do not provide a value for this attribute, the system class path will be used.
name	Specifies the name of the WSIF port in the Web service. This name will be used to identify the WSIF port in the WSDL's `port` element.

The Ant task for `genWsdl` in Example 9–3 inserts WSIF Java binding code into the WSDL's binding, binding operation, and port clauses. The `HelloImpl` class used as the `className` attribute to the `wsifJavaPort` argument, contains a `sayHello`

method that takes a `greeting` parameter of type `String`. The variable `${additional.class.path}` represents the classpath.

Example 9–3 Using wsifJavaPort to Configure a Single Java Port

```
<oracle:genWsdl
       output="wsdl"
       style="rpc"
       use="literal"
       >
       <oracle:porttype
          interfaceName="oracle.demo.hello.HelloInterface">
           <oracle:wsifJavaPort name="HelloServiceJavaPort"
                     className="oracle.demo.hello.HelloImpl"  />
       </oracle:porttype>
       <oracle:classpath>
           <oracle:pathelement path="${additional.class.path}" />
           <oracle:pathelement location="build/classes" />
       </oracle:classpath>
</oracle:genWsdl>
```

Example 9–4 illustrates the WSIF Java binding code that the Ant task in the previous example inserts into the WSDL's binding, binding operation, and port clauses.

Example 9–4 WSDL Extensions for WSIF Java Bindings

```
...
<java:binding/>
     <format:typeMapping encoding="Java" style="Java">
           <format:typeMap typeName="xsd:string" formatType="java.lang.String"/>
     </format:typeMapping>
     ...
<java:operation
      methodName="sayHello"
      parameterOrder="greeting"
      methodType="instance"
      returnType="result" />
      ...
<port name="HelloServiceJavaPort" binding="tns:JavaBinding">
    <java:address className="oracle.demo.hello.HelloImpl" />
</port>
...
```

Configuring a WSIF Endpoint for Multiple Java Ports

You can use the `wsifJavaPort` argument with the `assemble` and `genWsdl` commands to define WSIF bindings for multiple Java ports. Example 9–5 illustrates an Ant task that creates a WSIF port named `HelloServiceJavaPort` and a SOAP port named `HelloServiceSoapPort`. The SOAP port is created by adding the `<oracle:port name="...  "/>` subtask. Note that `className` is declared in the parent Ant task. The variable `${additional.class.path}` represents the classpath. Table 9–1 on page 9-4 describes the attributes that can be used by `wsifJavaPort`.

Example 9–5 Using wsifJavaPort to Configure a WSIF Java Port and a SOAP Java Port

```
<oracle:genWsdl
       output="wsdl"
       style="rpc"
```

```
       use="literal"
       >
       <oracle:porttype
         interfaceName="oracle.demo.hello.HelloInterface"
         className="oracle.demo.hello.HelloImpl">
           <oracle:wsifJavaPort name="HelloServiceJavaPort" />
           <oracle:port name="HelloServiceSoapPort" />
       </oracle:porttype>
       <oracle:classpath>
           <oracle:pathelement path="${additional.class.path}" />
           <oracle:pathelement location="build/classes" />
       </oracle:classpath>
</oracle:genWsdl>
```

WSIF Java Extensions to the WSDL

To comply with the WSIF Framework definition, OracleAS Web Services inserts binding information into the WSDL that allows a Java class to be represented as a Web service. Example 9–4 on page 9-5 illustrates how additional bindings, binding operations, and port clauses are added to the WSDL to describe a WSIF service. Example 9–6 illustrates the corresponding XML schema definitions for these extensions.

Example 9–6 Schema Definitions to Support WSIF Java Bindings

```
...
<!-- Java binding -->
<binding ... >
    <java:binding/>
    <format:typeMapping style="uri" encoding="..."/>?
        <format:typeMap typeName="qname"|elementName="qname"
              formatType="nmtoken"/>*
    </format:typeMapping>
    ...
    <operation>*
        <java:operation
            methodName="nmtoken"
            parameterOrder="nmtoken"?
            returnPart="nmtoken"?
            methodType="instance|static|constructor"? />?
        <input name="nmtoken"? />?
        <output name="nmtoken"? />?
        <fault name="nmtoken"? />?
    </operation>
</binding>
 ....
<service ... >
    <port name="nmtoken" >*
         <java:address
             className="nmtoken"
             classPath="nmtoken"?
             classLoader="nmtoken"? />
    </port>
</service>
...
```

Configuring a WSIF Endpoint for EJBs

To insert WSIF EJB extensions into the WSDL, the WebServicesAssembler tool provides two arguments that can be used with the `ejbAssemble` or `genWsdl` commands.

- `wsifEjbBinding`
- `wsifEjbPort`

As you assemble a Web service based on an EJB with the `ejbAssemble` command, the `wsifEjbBinding` or `wsifEjbPort` arguments allow you to request a WSIF binding. For more information on creating a Web service based on an EJB, see "Assembling a Web Service with EJBs" in the *Oracle Application Server Web Services Developer's Guide*.

These arguments add the appropriate WSDL extensions that allow a WSIF client to access an EJB exposed as a Web service. The section "WSIF EJB Extensions to the WSDL" on page 9-10 lists the schema definitions that can be added to the WSDL to support WSIF EJB bindings.

To configure the WSIF EJB binding for a single port, you can use either the `wsifEjbBinding` or `wsifEjbPort` argument. To configure WSIF EJB bindings for multiple ports, you must use the `wsifEjbPort` argument.

While the `wsifEjbBinding` argument can be used either on the command line or in an Ant task, the `wsifEjbPort` can be used only in an Ant task.

The following sections describe how to configure WSIF endpoints for single and multiple EJB ports.

- Configuring a WSIF Endpoint for a Single EJB Port
- Configuring a WSIF Endpoint for Multiple EJB Ports

Configuring a WSIF Endpoint for a Single EJB Port

A WSIF endpoint for a single EJB port can be configured with the `wsifEjbBinding` or `wsifEjbPort` argument.

- Configuring a Single EJB Port with wsifEjbBinding
- Configuring a Single EJB Port with wsifEjbPort

Configuring a Single EJB Port with wsifEjbBinding

The `wsifEjbBinding` argument can be used by either the `ejbAssemble` or `genWsdl` command to specify the WSIF EJB binding for a single port.

The `wsifEjbBinding` argument requires you to specify the `name` and `jndiName` arguments. The `wsifEjbBinding` argument passes these values to the `ejb:address` element in the WSDL's `<port>` element. The `jndiProviderURL` and `initialContextFactory` arguments provide the JNDI provider URL and JNDI initial context values. If you do not provide these arguments, then these values are obtained from the `jndi.properties` file. For more information on this argument, see "`wsifEjbBinding`" in the *Oracle Application Server Web Services Developer's Guide*.

Example 9–7 illustrates a WebServicesAssembler command line for `genWsdl` that calls `wsifEjbBinding`. Example 9–8 illustrates the corresponding Ant task. The argument causes the `ejb:address` element in the WSDL's `<port>` element to be populated with `className="oracle.demo.hello.HelloImpl"` and `jndiName="HelloService2EJB"`. The variable `${`*additional.class.path*`}` represents the classpath.

Example 9–7 Command Line, Using wsifEjbBinding to Configure a Single EJB Port

```
java -jar wsa.jar -genWsdl
                  -output wsdl
                  -style rpc
                  -use literal
                  -wsifEjbBinding true
                  -jndiName HelloService2EJB
                  -interfaceName oracle.demo.ejb.HelloServiceInf
                  -jndiProviderUrl deployer:oc4j:localhost:23791/HelloService
                  -initialContextFactory oracle.j2ee.rmi.RMIInitialContextFactory
                  -className oracle.demo.hello.HelloImpl
                  -classpath ${additional.class.path}:build/classes
```

Example 9–8 Ant Task, Using wsifEjbBinding to Configure a Single EJB Port

```
<oracle:genWsdl
      output="wsdl"
      style="rpc"
      use="literal"
      wsifEjbBinding="true"
      jndiName="HelloService2EJB"
      jndiProviderUrl="deployer:oc4j:localhost:23791/HelloService"
      initialContextFactory="oracle.j2ee.rmi.RMIInitialContextFactory"
      >
       <oracle:porttype
            interfaceName="oracle.demo.ejb.HelloServiceInf"
            className="oracle.demo.hello.HelloImpl">
       </oracle:porttype>
       <oracle:classpath>
            <pathelement path="${additional.class.path}"/>
            <pathelement location="build.ejb.dir"/>
       </oracle:classpath>
</oracle:genWsdl>
```

Configuring a Single EJB Port with wsifEjbPort

The `wsifEjbPort` argument can be used only in an Ant task. Although you typically use `wsifEjbPort` to specify WSIF EJB bindings for multiple ports, you can also use it to specify a single port. The argument has attributes that allow you to specify a WSIF port name, a class name, a JNDI name, a JNDI initial context, and a JNDI provider URL for each port. These attributes are passed to the `ejb:address` element in the WSDL. Table 9–2 describes the attributes that can be used by the `wsifEjbPort` argument. Except for `jndiName`, all of the attributes are optional.

Table 9–2 Attributes for the wsifEjbPort argument.

Attribute	Description
className	Specifies the class name of the EJB's home interface. If you do not provide a value for this attribute, WebServicesAssembler will use the class name for the parent Ant task if it was specified.
initialContextFactory	Specifies the name of the factory that will provide the initial context. If you not provide a value for this attribute, the value in the `jndi.properties` file will be used.
jndiName	(required) Specifies the JNDI name for the EJB.
jndiProviderURL	Specifies the URL for the JNDI Provider. If you do not provide a value for this attribute, the value in the `jndi.properties` file will be used.

Table 9–2 (Cont.) Attributes for the wsifEjbPort argument.

Attribute	Description
name	Specifies the name of the WSIF Port in the Web service. This name will be used to identify the WSIF port in the WSDL's `port` element.

The Ant task for `genWsdl` in Example 9–9 inserts WSIF EJB binding code into the WSDL's binding, binding operation, and port clauses. The `HelloHome` class used as the `className` attribute to the `wsifEJBPort` argument, contains a `sayHello` method that takes a `greeting` parameter of type `String`. The variable `${additional.class.path}` represents the classpath.

Example 9–9 Using wsifEjbPort to Configure a Single EJB Port

```
<oracle:genWsdl
    output="wsdl"
    style="rpc"
    use="literal"
    serviceName="${app.name}"
    >
    <oracle:porttype
         interfaceName="oracle.test.wsif.HelloServiceInf">
           <oracle:wsifEjbPort name="EjbPort"
            className="oracle.test.wsif.HelloHome"
            jndiName="HelloServiceBean"
            jndiProviderUrl="deployer:oc4j:localhost:23791/HelloService"
            initialContextFactory="oracle.j2ee.rmi.RMIInitialContextFactory"
            />
    </oracle:porttype>
    <oracle:classpath>
        <oracle:pathelement path="${additional.class.path}"/>
        <oracle:pathelement location="build/classes"/>
    </oracle:classpath>
</oracle:genWsdl>
```

Example 9–10 illustrates the WSIF EJB binding code that the previous example inserts into the WSDL's binding, binding operation, and port clauses.

Example 9–10 WSDL Extensions for WSIF EJB Bindings

```
...
<ejb:binding/>
    <format:typeMapping encoding="Java" style="Java">
        <format:typeMap typeName="xsd:string" formatType="java.lang.String"/>
    </format:typeMapping>
...
<operation name="sayHello">
    <ejb:operation
         methodName="sayHello"
         interface="remote"
         parameterOrder="greeting"
         returnPart="result"/>
...
<port name="EjbPort" binding="tns:EjbPortBinding">
    <ejb:address  className="oracle.test.wsif.HelloHome"
        jndiName="HelloServiceBean"
        initialContextFactory="oracle.j2ee.rmi.RMIInitialContextFactory"
        jndiProviderURL="deployer:oc4j:localhost:23791/HelloService"/>
```

```
</port>
...
```

Configuring a WSIF Endpoint for Multiple EJB Ports

Use the `wsifEjbPort` argument to define WSIF bindings for multiple EJB ports. This argument can be used by the `ejbAssemble` and the `genWsdl` Ant tasks. Example 9–11 illustrates the Ant task that creates a WSIF port named `EjbPort` and a SOAP port named `SoapPort`. The SOAP port is created by adding the `<oracle:port name="... "/>` subtask. The variable `${additional.class.path}` represents the classpath. Table 9–2 on page 9-8 describes the attributes that can be used by `wsifEjbPort`.

Example 9–11 Using wsifEjbPort to Configure Multiple EJB Ports

```
<oracle:genWsdl

    output="wsdl"
    style="rpc"
    use="literal"
    serviceName="${app.name}"
    >
      <oracle:porttype interfaceName="oracle.test.wsif.HelloServiceInf"
        >
        <oracle:port name="SoapPort" uri="TestWsdlService"/>
        <oracle:wsifEjbPort name="EjbPort"
          className="oracle.test.wsif.HelloHome"
          jndiName="HelloServiceBean"
          jndiProviderUrl="deployer:oc4j:localhost:23791/HelloService"
          initialContextFactory="oracle.j2ee.rmi.RMIInitialContextFactory"
        />
     </oracle:porttype>
    <oracle:classpath>
        <oracle:pathelement path="${additional.class.path}"/>
        <oracle:pathelement location="build/classes"/>
    </oracle:classpath>
</oracle:genWsdl>
```

WSIF EJB Extensions to the WSDL

To comply with the WSIF Framework definition, OracleAS Web Services inserts code into the WSDL that allows an EJB to be represented as a Web service. Example 9–10 on page 9-9 illustrates how additional bindings, binding operations, and port clauses are added to the WSDL to describe a WSIF service. Example 9–12 illustrates the corresponding XML schema definitions for these extensions.

Example 9–12 Schema Definitions to Support WSIF EJB Bindings

```
...
<!-- EJB binding -->
<binding ... >
    <ejb:binding/>
    <format:typeMapping style="uri" encoding="..."/>?
        <format:typeMap typeName="qname"|elementName="qname"
                   formatType="nmtoken"/>*
    </format:typeMapping>
    ...
    <operation>*
        <ejb:operation
```

```
            methodName="nmtoken"
            parameterOrder="nmtoken"?
            returnPart="nmtoken"?
            interface="home|remote"? />?
        <input name="nmtoken"? />?
        <output name="nmtoken"? />?
        <fault name="nmtoken"? />?
    </operation>
</binding>
...
<service ... >
    <port>*
         <ejb:address
             className="nmtoken"
             jndiName="nmtoken"
             initialContextFactory="nmtoken"?
             jndiProviderURL="url"?  />
    </port>
</service>
...
```

Configuring a WSIF Endpoint for Database Resources

To insert WSIF extensions for database resources into the WSDL, OracleAS Web Services provides two arguments that can be used with the `aqAssemble`, `dbJavaAssemble`, `plsqlAssemble`, `sqlAssemble` and `genWsdl` WebServicesAssembler commands.

- `wsifDbBinding`
- `wsifDbPort`

As you assemble a Web service based on a database resource with the `aqAssemble`, `dbJavaAssemble`, `plsqlAssemble`, `sqlAssemble`, or `genWsdl` commands, the `wsifDbBinding` and `wsifDbPort` arguments allow you to create a direct invocation to the database by using a WSIF binding. Note that database WSIF clients must provide a JNDI setup where the data source for connecting to the database can be determined at runtime.

These arguments add the appropriate WSDL extensions that allow a WSIF client to access database resources exposed as a Web service. The section "WSIF SQL Extensions to the WSDL" on page 9-16 lists the schema definitions that can be added to the WSDL to support WSIF database resource bindings.

To configure the WSIF database resource binding for a single port, you can use either the `wsifDbBinding` or `wsifDbPort` argument. To configure WSIF database resource bindings for multiple ports, you must use the `wsifDbPort` argument.

While the `wsifDbBinding` argument can be used either on the command line or in an Ant task, `wsifDbPort` can be used only in an Ant task.

The following sections describe how to configure WSIF endpoints for single and multiple database resource ports.

- Configuring a WSIF Endpoint for a Single Database Resource Port
- Configuring a WSIF Endpoint for Multiple Database Resource Ports

Configuring a WSIF Endpoint for a Single Database Resource Port

The WSIF endpoint for a single database resource port can be configured with `wsifDbBinding` or `wsifDbPort`.

- Configuring a Single Database Resource Port with wsifDbBinding
- Configuring a Single Database Resource Port with wsifDbPort

Configuring a Single Database Resource Port with wsifDbBinding

The `wsifDbBinding` argument can be used with `aqAssemble`, `dbJavaAssemble`, `plsqlAssemble`, `sqlAssemble`, and `genWsdl` WebServicesAssembler commands to specify the WSIF database resource bindings for a single port.

To establish the database connection, you must specify either `dataSource` or a combination of `dbUser` and `dbConnection`.

Table 9–3 Attributes for the wsifDbBinding and wsifDbPort Arguments

Attribute	Description
className	(required) Specifies the name of the Java class generated by Oracle JPublisher. If you do not provide a value for this attribute, WebServicesAssembler will derive a value based on the port name.
dataSource	Specifies the JNDI location of the data source used by the Web service. For more information on the argument, see "`datasource`" in the "Database Assembly Commands" section of the*Oracle Application Server Web Services Developer's Guide*.
dbConnection	Specifies the JDBC URL for the database. If you specify `dbConnection`, you must also provide a value for `dbUser`. See "`dbConnection`" in the "Database Assembly Commands" section of the*Oracle Application Server Web Services Developer's Guide* for more information on this attribute.
dbUser	Specifies the database schema and password in the form *user/password*. If you specify `dbUser`, you must also provide a value for `dbConnection`. See "`dbUser`" in the "Database Assembly Commands" section of the*Oracle Application Server Web Services Developer's Guide* for more information on this attribute.
name	(required for `wsifDBPort` only) Specifies the port name.

- If the `dataSource` argument appears in an Ant task or on the command line, it will provide assembly time and runtime access to the database.
- If the `dbConnection` and `dbUser` arguments appear in an Ant task or on the command line, they will provide assembly time access to the database.
- If the `dbConnection` and `dataSource` arguments appear in an Ant task or on the command line, then `dbConnection` provides assembly time access to the database while `dataSource` provides runtime access.
- If the `dbConnection`, `dbUser`, and `dataSource` arguments appear in an Ant task or on the command line, then `dbConnection` and `dbUser` will be used for assembly time access to the database. The `dataSource` argument will be used for runtime access.

Example 9–13 illustrates a WebServicesAssembler command line for `sqlAssemble` that calls `wsifDbBinding` to configure a single port for a database resource. Example 9–14 illustrates the corresponding Ant task. These examples provides values for `dbUser` and `dbConnection` which will be entered into the `<port>` element of the WSDL. The `useDataSource` argument is specified as `false`: the value of the

`dataSource` argument will not be used in the WSDL `<port>` element. The variables `${wsdemo.common.class.path}`, `${additional.class.path}`, and `${service.classes.dir}` represent classpath elements.

Example 9–13 Command Line, Using wsifDbBinding to Configure a Single Database Resource Port

```
java -jar wsa.jar -sqlAssemble
                  -dbUser scott/tiger
                  -dbConnection jdbc:oracle:thin:@...
                  -dataSource jdbc/OracleManagedDS
                  -appName SqlWsifTest
                  -serviceName sqlwsif
                  -output build
                  -ear dist/SqlWsifTest.ear
                  -style rpc
                  -use literal
                  -wsifDbBinding true
                  -sqlstatement "getEname=select ename from emp"
                  -sqlstatement "updateSal=update emp SET sal=sal+500 where
                                 ename=:{myname VARCHAR}"
                   -classpath ${CLASSPATH}:build/classes
```

Example 9–14 Ant Task, Using wsifDbBinding to Configure a Single Database Resource Port

```
<oracle:sqlAssemble
    dbUser="scott/tiger"
    dbConnection="jdbc:oracle:thin@…"
    dataSource="jdbc/OracleManagedDS"
    appName="SqlWsifTest"
    serviceName="sqlwsif"
    output="build"
    ear="dist/SqlWsifTest.ear"
    style="rpc"
    use="literal"
    debug="true"
    wsifDbBinding="true"
    >
    <oracle:port name="sqlwsif" uri="SqlWsifTest" />
    <sqlstatement value="getEname=select ename from emp" />
    <sqlstatement value="updateSal=update emp SET sal=sal+500 where ename=:{myname
VARCHAR}" />
    <oracle:classpath>
        <oracle:pathelement path="${wsdemo.common.class.path}"/>
        <oracle:pathelement path="${additional.class.path}"/>
        <oracle:pathelement location="client/classes"/>
        <oracle:pathelement location="${service.classes.dir}"/>
    </oracle:classpath>
</oracle:sqlAssemble>
```

Example 9–15 illustrates the WSIF database resource binding code that the previous example inserts into the WSDL's `<port>` element.

Example 9–15 WSDL Extensions for WSIF Database Resource Bindings

```
...
<service name="sqlwsif">
    <port name="JavaPort" binding="tns:JavaPortBinding">
        <java:address className="...User"
```

```
            dataSource="jdbc/OracleManagedDS"/>
    </port>
</service>
...
```

Configuring a Single Database Resource Port with wsifDbPort

The `wsifDbPort` argument can be used only in an Ant task. Although you typically use `wsifDbPort` to specify the WSIF database resource bindings for multiple ports, you can also use it to specify a single port. The argument has attributes that allow you to specify a WSIF port name, a class name for the Java files generated by Oracle JPublisher, and database connection information. These attributes are passed to the `java:address` element of the WSDL's `<port>` element.

Table 9–3 describes the attributes that can be used by the `wsifDbPort` argument. Except for `className` and `name`, all of the attributes are optional.

- If the `dataSource` argument appears in an Ant task or on the command line, it will provide assembly time and runtime access to the database.
- If the `dbConnection` and `dbUser` arguments appear in an Ant task or on the command line, they will provide assembly time access to the database.
- If the `dbConnection` and `dataSource` arguments appear in an Ant task or on the command line, then `dbConnection` provides assembly time access to the database while `dataSource` provides runtime access.
- If the `dbConnection`, `dbUser`, and `dataSource` arguments appear in an Ant task or on the command line, then `dbConnection` and `dbUser` will be used for assembly time access to the database. The `dataSource` argument will be used for runtime access.

Example 9–13 illustrates the WebServicesAssembler command line for `wsifDbPort` which inserts WSIF database resource code into the WSDL's binding, binding operation, and port clauses. Example 9–16 illustrates the corresponding Ant task. In the `wsifDbPort` argument, the name of the port is set to `JavaPort` and the name of the JPublisher-generated Java class is `oracle.generated.sqlwsifUser`.

Example 9–16   Ant Task, Using wsifDbPort to Configure a Single Database Resource Port

```
<oracle:sqlAssemble
    dataSource="jdbc/OracleManagedDS"
    appName="SqlWsifTest"
    portName="sqlwsif"
    serviceName="sqlwsif"
    output="build"
    ear="dist/SqlWsifTest.ear"
    style="rpc"
    use="literal"
    debug="true"
    >
    <oracle:port name="sqlwsif"  uri="SqlWsifTest" />
    <sqlstatement value="getEname=select ename from emp" />
    <sqlstatement value="updateSal=update emp SET sal=sal+500 where ename=:{myname
VARCHAR}" />
    <oracle:wsifDbPort
        name="JavaPort"
        className="oracle.generated.sqlwsifUser"
    />
    <oracle:classpath>
```

```
        <oracle:pathelement path="${wsdemo.common.class.path}"/>
        <oracle:pathelement path="${additional.class.path}"/>
        <oracle:pathelement location="client/classes"/>
        <oracle:pathelement location="service/classes"/>
    </oracle:classpath>
</oracle:sqlAssemble>
```

Example 9–17 illustrates the WSIF database resource binding code that the previous example inserts into the WSDL.

Example 9–17 WSDL Extensions for WSIF Database Resource Bindings

```
<service name="sqlwsif">
    <port name="JavaPort" binding="tns:JavaPortBinding">
        <java:address className="oracle.generated.sqlwsifUser"
            dataSource="jdbc/OracleManagedDS"/>
    </port>
</service>
```

Configuring a WSIF Endpoint for Multiple Database Resource Ports

Use the `wsifDbPort` argument to define WSIF bindings for multiple database resource ports. This argument can be used in the Ant tasks for the `aqAssemble`, `dbJavaAssemble`, `plsqlAssemble`, `sqlAssemble`, and `genWsdl` commands.

Example 9–18 illustrates an Ant task that creates a SOAP port and a WSIF port. The port operations are based on the `<sqlstatement value="..."/>` subtasks. The `wsifDbPort` subtask creates the `JavaPort` port. The `oracle:port` subtask creates the SOAP port. The variables `${wsdemo.common.class.path}`, `${additional.class.path}`, and `${service.classes.dir}` represent classpath elements. Table 9–3 describes the attributes that can be used by `wsifDbPort`.

Example 9–18 Ant Task, Using wsifDbPort to Configure Multiple Database Resource Ports

```
<oracle:sqlAssemble
        dbUser="scott/tiger"
        dbConnection="jdbc:oracle:thin:@${DB_HOST}:${DB_PORT}:${DB_SID}"
        dataSource="jdbc/OracleManagedDS"
        appName="wsifTest"
        portName="dbwsif"
        serviceName="dbwsif"
        output="build"
        ear="dist/wsifDbTest.ear"
        style="rpc"
        use="literal"
        debug="true"
        >
        <oracle:port name="dbwsif"  uri="wsifDbTest"/>
        <sqlstatement value="getEname=select ename from emp" />
        <sqlstatement value="updateSal=update emp SET sal=sal+500 where
            ename=:{myname VARCHAR}" />
        <oracle:wsifDbPort name="JavaPort"
                className="oracle.generated.sqlwsifUser"
        >
        <oracle:classpath>
            <oracle:pathelement path="${wsdemo.common.class.path}"/>
            <oracle:pathelement path="${additional.class.path}"/>
            <oracle:pathelement location="${client.classes.dir}"/>
```

```
            <oracle:pathelement location="${service.classes.dir}"/>
        </oracle:classpath>
    </oracle:sqlAssemble>
```

WSIF SQL Extensions to the WSDL

To comply with the WSIF Framework definition, OracleAS Web Services inserts code into the WSDL that allows a database resource to be represented as a Web service. Example 9–15 on page 9-13 illustrates how additional database resource bindings are placed in the WSDL. Specifically, the WSIF `JavaPortBinding` is further extended with the optional dataSource attribute. The presence of this attribute indicates that this is a database port and not a Java port. Example 9–19 illustrates the corresponding XML schema definitions for this extension.

Example 9–19   Schema Definitions to Support WSIF Database Resource Bindings

```
...
<definitions .... >

    <!-- Java binding -->
    <binding ... >
        <java:binding/>
        <format:typeMapping style="uri" encoding="..."/>?
            <format:typeMap typeName="qname"|elementName="qname"
formatType="nmtoken"/>*
        </format:typeMapping>
        <operation>*
            <java:operation
                methodName="nmtoken"
                parameterOrder="nmtoken"?
                returnPart="nmtoken"?
                methodType="instance|static|constructor"? />?
            <input name="nmtoken"? />?
            <output name="nmtoken"? />?
            <fault name="nmtoken"? />?
        </operation>
    </binding>

    <service ... >
        <port>*
             <java:address
                 className="nmtoken"
                 classPath="nmtoken"?
                 classLoader="nmtoken"?
                 dataSource="nmtoken"?
        </port>
    </service>
</definitions>
...
```

Writing a WSIF Client

A WSIF client is similar to a DII client in that it does not rely on any pre-generated stub. The WSDL plays a central role for the WSIF client. Since the WSDL contains all of the information to invoke the service, it must be available to the client at runtime. The WSIF client must create the WSIF service, port, operation, and messages, before it actually invokes the operation with the messages.

The WSIF API, which is based on the abstract service description portion of the WSDL, contains methods for obtaining this information from the WSDL. The WSIF API is available from the Apache Software Group Web site.

http://ws.apache.org/wsif/

> **Note:** There is no WebServicesAssembler support for generating WSIF clients.

To write a WSIF client, follow these general steps.

1. Create a new instance of a WSIF service factory.

   The `newInstance` method of the `WSIFServiceFactory` class instantiates a new WSIF service factory. For example:

   ```
   // create a service factory
   WSIFServiceFactory factory = WSIFServiceFactory.newInstance();
   ```

2. Create a WSIF service.

   Information in the WSDL file is used to build the WSIF service. The `WSIFServiceFactory` class provides a variety of `getService` methods to obtain the service definition from the WSDL. The location of the WSDL file can be specified either as a URL location available over HTTP, or as a file-based location available through the classLoader. Additional parameters for service creation are service name/namespace and portType/namespace. For example:

   ```
   WSIFService service = factory.getService( ... );
   ```

3. Create the WSIF port, operation, and messages.

   Use the WSIF service to create the WSIF port. Use one of the WSIF API `WSIFService.getPort` methods to get the WSIF port for the port type that this factory supports. For example:

   ```
   WSIFPort port = service.getPort("portName");
   ```

4. Create the WSIF operation.

   Use the WSIF port to create the WSIF operation. An operation is created using a service name or a combination of service name and input and output messages. There must be exactly one operation in this port's `portType` with this name. The WSIF API supports a variety of `createOperation` methods, depending on the type of J2EE component you are exposing as a Web service. For example:

   ```
   WSIFOperation operation = port.createOperation("operationName");
   ```

5. Create the input, output, and fault messages.

   Use the WSIF operation to create the messages. The `WSIFOperation` interface in the WSIF API supports a variety of methods to create input, output, and fault messages. For example:

   ```
   WSIFMessage input = operation.createInputMessage();
   WSIFMessage output = operation.createOutputMessage();
   WSIFMessage fault = operation.createFaultMessage();
   ```

6. Populate the input message.

   To populate the input message, set the parts (defined in the WSDL) of the input message. For example, use one of the `WSIFMessage.setObjectPart` methods

to allow the operation to pass a value to the service. You can also set message types using Java classes based on the type map, which is set by the service.

```
input.setObjectPart( ... );
```

7. Make the call to the service.

   Use one of the WSIF API "execute" methods to execute the operation. The signature allows for input, output and fault messages. The WSIF API supports a variety of "execute" methods, depending on the type of J2EE component you are exposing as a Web service. For example:

   ```
   operation.executeRequestResponseOperation( ... );
   ```

   The call can return a Boolean variable that is equal to `true` if the operation succeeds, or `false` otherwise. If the call fails, a fault message can be examined to determine the exact reason for the failure. The fault message is populated based on the `SOAPBody:Fault` element defined in the SOAP specification. If the call succeeds, the content of the output message can be extracted into Java classes based on the output message parts (similar to the population of the input messages).

Example 9–20 illustrates a sample DII client. The client performs an invocation on the `HelloService`'s `sayHello` method, and passes "Duke" as the value for the `name` parameter. The WSDL that defines the service is deployed at `http://localhost:8888/helloWSIFDii/helloWSIFDii?WSDL`. The `portType` is the QName `{http://hello.demo.oracle/}HelloInterface`, where `http://hello.demo.oracle/` is the `portType` namespace and `HelloInterface` is the port name. The name of the WSIF port is `HelloServicePort` and the name of the operation is `sayHello`. The `executeRequestResponseOperation` method makes the call to the service.

Example 9–20 Sample WSIF Client Code

```
import org.apache.wsif.*;
...
// create a service factory
WSIFServiceFactory factory = WSIFServiceFactory.newInstance();

// parse the WSDL
WSIFService service =
        factory.getService("http://localhost:8888/helloWSIFDii/helloWSIFDii?WSDL",
        null, null, "http://hello.demo.oracle/", "HelloInterface");

// create WSIF port, operation, and messages
WSIFPort port = service.getPort("HelloServicePort");
WSIFOperation operation = port.createOperation("sayHello");
WSIFMessage input = operation.createInputMessage();
WSIFMessage output = operation.createOutputMessage();
WSIFMessage fault = operation.createFaultMessage();
input.setObjectPart("name", "Duke");

// make the actual call to the service
operation.executeRequestResponseOperation(input, output, fault);
...
```

Writing a WSIF Client Using a Dynamic Proxy

A much simpler way of writing a WSIF client is to write it using a service endpoint interface. The WSIF runtime can create a corresponding implementation for you in the

form of a dynamic proxy. This technique assumes that you already have a compiled service endpoint interface. The WSIF API provides a `getStub` method (`org.apache.wsif.WSIFService.getStub`) that gets the dynamic proxy that implements the interface.

The following steps summarize how to get the interface as a dynamic proxy.

1. Create a new instance of a WSIF service factory.

   For example:

   ```
   WSIFServiceFactory factory = WSIFServiceFactory.newInstance();
   ```

2. Create a WSIF service.

   For example:

   ```
   WSIFService service = factory.getService( ... );
   ```

3. Pass the service port name and the service endpoint interface class to the `getStub` method.

4. Cast the dynamic proxy which is returned to the service endpoint interface.

   For example:

   ```
   HelloInterface stub = (HelloInterface) service.getStub( ... );
   ```

5. Call methods directly on the proxy that implements the interface.

   For example:

   ```
   String resp = stub.sayHello("Duke");
   ```

Example 9–21 illustrates using the WSIF client as a dynamic proxy. The port name `HttpSoap11` and the service endpoint interface class `HelloInterface.class` are passed to the `getStub` method. The returned dynamic proxy is cast to the `HelloInterface` service endpoint interface. Note that the `stub` parameter is also declared as type `HelloInterface`. Using the stub, you can then call methods on the interface directly.

Example 9–21 Using a WSIF Client as a Dynamic Proxy

```
// Create a service factory
WSIFServiceFactory factory = WSIFServiceFactory.newInstance();

// Parse the WSDL
WSIFService service = factory.getService(serviceURL + "?WSDL", null, null,
        "http://hello.demo.oracle/", "HelloInterface");

// Call the getStub method with the port name and
//    the service endpoint interface class.
// Cast the returned dynamic proxy to the service endpoint interface.
HelloInterface stub = (HelloInterface) service.getStub("HttpSoap11",
HelloInterface.class);

// Call methods directly on the interface
String resp = stub.sayHello("Duke");
```

Using genInterface to Generate a Service Endpoint Interface

The steps in the previous section assume that you have a WSDL and a service endpoint interface. If you do not have the service endpoint interface, one way to obtain it is to write it by hand, using the WSDL as a guide.

As an alternative to writing the interface by hand, you can pass the WSDL to the WebServicesAssembler `genInterface` command. The command will generate the service endpoint interface for you. This is especially convenient if the WSDL has a large number of operations that must be converted to methods.

When `genInterface` converts WSDL operations to Java methods, it will always make the first letter of the method name lowercase. If the original operation name in the WSDL begins with an uppercase character, this will cause a mismatch when you attempt to call methods on the proxy. Typically, a "`Method does not exist`" error will be returned.

To avoid this error, edit the service endpoint interface, if necessary, to ensure that method names exactly match the operation names in the WSDL, character-for-character.

Accessing the Database from a WSIF Client

The `wsifDbBinding` and `wsifDBPort` arguments add the appropriate bindings to the WSDL that allow a WSIF client to access database resources exposed as a Web service. The `dataSource` argument identifies the database that contains the resources.

If the `dbConnection` argument is used identify the database URL, then WebServicesAssembler generates its value into the WSDL file. To access the database, the WSIF client must pass the user name and password values at runtime. The generated code provides two methods to do this.

```
public void _setDataSourceUser(String);
public void _setDataSourcePassword(String);
```

The following are the complete paths of these methods. The *CLASSNAME* variable represents the value specified by the `className` argument. If `className` is not specified, it defaults to `portName`.

```
CLASSNAME.setDataSourceUser(String user);
CLASSNAME.setDataSourceUser(String password);
```

If the `dataSource` argument is specified in the command or Ant task, then these two methods are not needed. The client will already have the privileges of accessing the database.

Example 9–22 illustrates a client for the Web service assembled by the Ant task in Example 9–14. The `setDataSourceUser` and `setDataSourcePassword` methods, which pass the user name and password to the server, are highlighted in bold.

Example 9–22 Client Code to Pass a User Name and Password to the Service

```
WSIFServiceFactory factory = WSIFServiceFactory.newInstance();
WSIFService service = factory.getService(wsdlPath1, null, null,
                      "http://generated.oracle/", "sqlwsif");
oracle.generated.sqlwsif stub = null;
stub = (oracle.generated.sqlwsif)
service.getStub("JavaPort", oracle.generated.sqlwsif.class);
stub.setDataSourceUser("scott");
stub.setDataSourcePassword("tiger");
System.out.println("Run Sql...");
```

```
String[] names = stub.getEname();
```

Adding Management Configuration to a WSIF Client

You can add a Web service management configuration for security, reliability, and auditing to a WSIF client.

A Web service management configuration on the WSIF client is useful only when it invokes a SOAP port or operation that has a matching Web service management configuration.

OracleAS Web Services does not support Web service management configuration for Java and EJB ports on the Web service side.

If a WSIF client which has a Web service management configuration communicates with a Java or an EJB port, then the Web service management configuration is ignored. The Web service management is used, however, if SOAP port is used to invoke the service.

The *J2EE_HOME*/config/wsif-wsm-config.xml file provides a template where you can specify outgoing management policies for the client. The wsif-wsm-config.xml file is based on the oracle-webservices-client-10_0.xsd schema.

By default, the wsif-wsm-config.xml file resides in the *J2EE_HOME*/config directory. If you change the name of the file or call it from another location, you can use the wsif.wsm.config.file system property to override its default name and location.

To create the management configuration and make it available to the WSIF client:

- Edit the wsif-wsm-config.xml file in the *J2EE_HOME*/config directory, or
- Use the wsif-wsm-config.xml file as a template and store it in the location of your choice. Specify the new name and location of the file with the following system property.

  -Dwsif.ws.config.file=<*path to file*>

Example 9–23 illustrates the contents of the wsif-wsm-config.xml file. Table 9–4 describes the elements in the file.

Example 9–23 WSIF Client Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<oracle-webservice-clients xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="/oracle-webservices-client-10_0.xsd">
    <webservice-client>
        <service-qname namespaceURI="" localpart=""/>
        <port-info>
            <wsdl-port namespaceURI="" localpart=""/>
            <service-endpoint-interface></service-endpoint-interface>
            <stub-property>
                <name></name>
                <value></value>
            </stub-property>
            <call-property>
                <name></name>
                <value></value>
            </call-property>
            <runtime>
                <security>
```

Port-level Security configuration. For more information on security configuration elements, see the *Oracle Application Server Web Services Security Guide*.

```
                </security>
                <reliability>
```

Port-level Reliability configuration. See "Port Level Reliability Elements on the Client" on page 5-9 for a description of reliability configuration elements.

```
                </reliability>
            </runtime>
            <operations>
                <operation name="">
                    <runtime>
                        <security>
```

Operation-level Security configuration. For more information on security configuration elements, see the *Oracle Application Server Web Services Security Guide*.

```
                        </security>
```

Operation level auditing configuration. *See* "Managing Auditing on the Client" on page 6-4 for a description of auditing configuration elements.

```
                        <auditing/>
                        <reliability>
```

Operation-level reliability configuration. See "Operation Level Reliability Elements on the Client" on page 5-10 for a description of reliability configuration elements.

```
                        </reliability>
                    </runtime>
                </operation>
            </operations>
        </port-info>
    </webservice-client>
</oracle-webservice-clients>
```

Table 9–4 describes the subelements for the `<port-info>` element. This element provides all of the information for a port within a service reference. You can specify either a `service-endpoint-interface` or `wsdl-port` to indicate the port that the container will use for container-managed port selection. If you specify both, then the container will use the `wsdl-port` value. If you do not specify `wsdl-port` or `service-endpoint-interface`, then the `<port-info>` property values will apply to all available ports.

The `<port-info>` element also contains subelements that let you specify quality of service features that are available for the port and its operations.

Table 9–4 Subelements of the <port-info> Element

Element Name	Description
<wsdl-port>	Specifies the `namespaceURI` and `localpart` of a port in the WSDL that the container will use for container-managed port selection.

Table 9–4 (Cont.) Subelements of the <port-info> Element

Element Name	Description
<service-endpoint-interface>	Specifies the fully-qualified path to the service endpoint interface of a WSDL port. The container uses this port for container-managed port selection.
<stub-property>	Defines the stub property values applicable to the port defined by the `<port-info>` element. The `name` and `value` subelements of `<stub-property>` are described in Table 9–5.
<call-property>	Defines the call property values applicable to the port defined by the `<port-info>` element. The `name` and `value` sub-elements of `<call-property>` are described in Table 9–5.
<runtime>	Contains client-side quality of service runtime information (security and/or reliability) applicable to all the operations provided by the referenced Web service. Each child element contains configuration for a specific feature.
<operations>	Contains a sequence of elements, one for each operation. The `<operation>` subelement indicates an individual operation. Each of these subelements contain client-side quality of service configuration for a single operation provided by the referenced Web service. The `<operations>` element is a subelement of the `<port-info>` element. For a description of `<operations>`, and its subelements, see "Adding OC4J-Specific Platform Information" in the *Oracle Application Server Web Services Developer's Guide*.

Table 9–5 describes the `name` and `value` subelements of the `<stub-property>` and `<call-property>` elements.

Table 9–5 Subelements of <stub-property> and <call-property> Elements

Element Name	Description
<name>	Defines the name of any property supported by the JAX-RPC `Call` or `Stub` implementation. See the output of the Javadoc tool for the valid properties for `javax.xml.rpc.Call` and `javax.xml.rpc.Stub`.
<value>	Defines a JAX-RPC property value that should be set on a `Call` object or a `Stub` object before it is returned to the Web service client.

Adding Message Attachments in WSIF

You can enable WSIF clients to add attachments to messages.

- Adding Attachments with the WSIF API
- Adding Attachments with the OracleCall API

Adding Attachments with the WSIF API

To enable a WSIF client to add an attachment to a message, follow these general steps:

1. Create a `DataHandler` object to encapsulate the attachment.
2. Use the WSIF API's `WSIFOperation.createInputMessage` method to create an input message on the desired operation.

3. Use the WSIF API's `WSIFMessage.setObjectPart` method to add the attachment to the message.

Example 9–24 illustrates how an attachment can be added to a message by using the WSIF API. In the code fragment, the `JohnDoe.jpg` attachment is saved as a `DataHandler` object. The `operationName` represents the operation that will be enabled to handle messages with attachments. The `WSIFOperation`'s `createInputMessage` method creates an input message on the operation. The `WSIFMessage`'s `setObjectPart` method attaches the attachment to the input message with the part name `myAttachment`.

Example 9–24 Adding an Attachment to a Message with the WSIF API

```
...
DataHandler dataHandler1 = new DataHandler(new URL("file:JohnDoe.jpg"));
WSIFOperation operation = port.createOperation(operationName);
WSIFMessage input = operation.createInputMessage();
input.setObjectPart("myAttachment", dataHandler1);
...
```

Adding Attachments with the OracleCall API

The `oracle.webservices.OracleCall` API enables you to add attachments to messages that will be handled by WSIF clients. Using this technique, you save the attachment as a `DataHandler` object. You then add the attachment to the DII call object that you obtain from the WSIF port.

Example 9–25 illustrates how an attachment can be added to a message by using the `OracleCall` API. In the code fragment, the `JohnDoe.jpg` attachment is saved as a `DataHandler` object. The `getCall` method gets the DII call from the WSIF port. This call must be cast to a SOAP port, then to an `OracleCall` object. The `OracleCall.addAttachment` method adds the attachment to the SOAP message in the call. Note that the `addAttachment` method enables you to directly set the content-transfer-encoding and content-ID of the attachment MIME part.

Example 9–25 Adding an Attachment to a Message with the WSIF API

```
...
DataHandler dataHandler1 = new DataHandler(new URL("file:JohnDoe.jpg"));
OracleCall call = (OracleCall) ((WSIFPort_JaxRpc) port).getCall();
call.addAttachment(dataHandler1, "BASE64", "<ID1@photo>");
...
```

Tool Support for WSIF

The Java J2EE Web Service wizard for Java classes in JDeveloper enables you to create WSIF bindings in addition to SOAP bindings for a Web service. Enter values for the class loader and class path that let the WSIF clients locate the service implementation class. These values are comparable to the `classLoader` and `classPath` parameters on the command line or Ant task.

The Web Service wizard for EJBs enables you to specify the initial context factory, JNDI provider URL, and JNDI name so that WSIF clients can locate the EJB through JNDI. These values are comparable to the `initialContextFactory`, `jndiProviderUrl`, and `jndiName` parameters on the command line or Ant task.

The Database Web Service wizard enables you to specify the user, connection, and datasource information necessary for defining WSIF bindings for a PL/SQL procedure.

Limitations

See "Using the Web Service Invocation Framework" on page E-11.

Additional Information

For more information on:

- using message attachments, see Chapter 2, "Working with Message Attachments"
- adding management configurations to a Web service, see Chapter 3, "Managing Web Services"
- adding security information to a Web service, see Chapter 4, "Ensuring Web Services Security" and the *Oracle Application Server Web Services Security Guide*.
- adding reliability information to a Web service, see Chapter 5, "Ensuring Web Service Reliability".
- the contents of the `wsmgmt.xml` file which contains the security configuration, see Appendix A, "Understanding the Web Services Management Schema".
- assembling a Web service top down, see "Assembling a Web Service from a WSDL" in the *Oracle Application Server Web Services Developer's Guide*.
- using Java classes to assemble a Web service, see "Assembling a Web Service with Java Classes" in the *Oracle Application Server Web Services Developer's Guide*.
- using EJBs to assemble a Web service, see "Assembling a Web Service with EJBs" in the *Oracle Application Server Web Services Developer's Guide*.
- using JMS topics and destinations to assemble a Web service, see "Assembling Web Services with JMS Destinations" in the *Oracle Application Server Web Services Developer's Guide*.
- using database resources, such as PL/SQL packages, SQL queries, DML statements, Oracle Streams AQ, or server-side Java classes, to assemble a Web service, see "Assembling Database Web Services" in the *Oracle Application Server Web Services Developer's Guide*.
- assembling a J2EE Web Service client, see "Assembling a J2EE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.
- assembling a J2SE Web service client, see "Assembling a J2SE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.
- using WebServicesAssembler commands to assemble Web service artifacts, see "Using WebServicesAssembler" in the *Oracle Application Server Web Services Developer's Guide*.

10

Using Web Service Providers

This chapter describes how to use the Provider API in Oracle Application Server Web Services. The Provider API lets you define custom processing logic for a Web services endpoint that is not tied to any particular service endpoint implementation strategy, such as JAX-RPC. The Provider model can be used to provide common functionality to a number of endpoints. Rather than incorporating the same functions into many Web services, the Provider model enables you to add the logic into the runtime directly.

What is a Provider?

A Provider lets you bypass the JAX-RPC tie-based framework and allows you the flexibility of choosing your own mechanism for traversing and interpreting the message payload. Since it runs in the OC4J container, a Provider can take advantage of the container's services such as schema validation, security, and reliability.

When to Use a Provider

You can choose to employ a Provider as your endpoint implementation if you find that your performance is lacking when leveraging a standard JAX-RPC endpoint implementation. Because a Provider does not use the JAX-RPC framework, marshalling and unmarshalling of data is not performed. Therefore, based on your message traversal and interpretation mechanism, there may be less parsing and serialization effort before the message is processed.

Also, because a Provider enables you to process the SOAP message directly, you can perform much more sophisticated processing such as content-based routing. For example, inbound messages could be routed to a number of intermediate destinations (perhaps for some intermediate level of processing) before routing to a final destination.

Understanding the Provider API

The interfaces and classes in the `oracle.webservices.provider` package allow you to implement and use your own Provider to process SOAP messages.

- Provider Interface—your application must implement this interface to create your Provider.
- ProviderConfig Class—configuration class that is passed to the Provider during its initialization. This represents the information related to the Provider configuration. The Provider also uses this class to interact with the container.
- MessageContext Class—context class that is passed to the Provider's `processMessage` method when an invocation is to be serviced by the Provider.

The Provider instance can use this class to obtain transport protocol-related information.

- HTTPConstants Class—provides a set of well-defined properties for the HTTP protocol that the container will set on the `MessageContext`.

Provider Interface

Your Provider class must implement the `oracle.webservices.provider.Provider` interface. The following sections provide more information on the methods in the `Provider` interface that must be implemented.

- init Method
- processMessage Method
- destroy Method

init Method

```
public void init(ProviderConfig config) throws ProviderException;
```

The `init` method is called when the Provider is loaded and initialized. The method can be called only once during the lifecycle of the Provider. In response to the `init` method, the container passes an instance of `ProviderConfig` to the Provider. This instance provides configuration information related to the Provider. The `ProviderConfig` instance provides mechanisms to set and get properties and the `ServletContext`. The `init` and `destroy` methods provide the lifecycle mechanisms for the Provider much like the `ServiceLifecycle` interface provides the lifecycle mechanisms for JAX-RPC service endpoint implementation (SEI) classes.

The Provider uses the `ProviderConfig` instance to set or retrieve properties. If a property is set on the context, it will overwrite the corresponding static property set in the deployment descriptors. For example, the Provider can set the boolean `VALIDATION_SOAP_SCHEMA` property to `true` to let the container perform schema validation. Schema validation can be performed only if the WSDL is available; it cannot be performed in pass-through mode. In passthrough mode, the provider itself is not exposed as an endpoint. The default value of the `VALIDATION_SOAP_SCHEMA` property is `false`. Schema validation is based on the `QName` value of the child element of the SOAP Body.

If the appropriate load-on-startup directive is provided (for example, by using the `web.xml` file in the Provider application WAR file), then the `init` method will be called on application startup.

processMessage Method

```
public SOAPMessage processMessage(SOAPMessage request, MessageContext
messageContext) throws ProviderException;
```

The `processMessage` method is the Provider's main access method: it is invoked every time a SOAP request is dispatched to the Provider instance. The `processMessage` method is where you provide the processing logic for the SOAP message. For the SOAP protocol, this method takes a `SOAPMessage` instance as an argument. From this `SOAPMessage` instance, the Provider can obtain access to other parts of the message payload, such as the SOAP body, headers, and attachments.

The `processMessage` method also takes a `MessageContext` object. The message context defines the properties that are valid for the particular message.

This method returns a `SOAPMessage` instance. If the instance is not null, then the Provider intends to return a response to the sender, assuming the underlying transport supports it. If the underlying transport is a one-way protocol, such as JMS, then an attempt to send a response will cause an error to be thrown in the server logs. The return must be null for pure one-way calls and must not be null for request-response calls.

destroy Method

```
public void destroy() throws ProviderException;
```

The container invokes the `destroy` method to notify the Provider instance of its intent to remove the Provider from its working set. After the method is called, the Provider instance is destroyed and garbage collected.

ProviderConfig Class

This class contains the configuration-related information for the Provider instance. The information includes details that are also listed in the proprietary deployment descriptors.

The properties of the `ProviderConfig` are used to communicate with the container. For example, a policy object reference can be set as a `ProviderConfig` property inside `processMessage` (assuming the Provider implementation maintains a reference to the `ProviderConfig` instance as an attribute initialized during the `init` method invocation) or the `init` method of the Provider instance. The container can read this policy object and apply the appropriate management functions to the Provider instance.

The following section provides more information on methods on the `ProviderConfig` class.

addService and removeService Methods

```
public void addService(String pathInfo, URL wsdlURL, QName wsdlPort, QName
serviceName, Object policy) throws ProviderException;

public void removeService(String pathInfo);
```

The `addService` and `removeService` methods of the ProviderConfig class can be invoked in the Provider's `init` and `processMessage` methods to dynamically add and remove manageable service endpoints, each with a unique URI. Internally, the container maintains a repository of WSDLs and a map of the WSDLs with the URIs of the incoming requests. In addition, these dynamic endpoints can be managed by using the Application Server Control tool, allowing you to apply security, reliability, auditing, and logging. The Provider can make processing decisions for incoming requests based on the URI available from the Provider's `MessageContext`.

These methods allow the Provider instance to dynamically provide or remove services to or from the container. The container can perform a variety of tasks on the services, such as test-page presentation, request message schema validation, and Web service management processing. The service registration process also allows the container to perform Web service management processing for the registered services.

The `addService` and `removeService` methods both take a `pathInfo` parameter. This parameter contains the extra path information associated with the URL sent by the client when it invokes the endpoint. The extra path information follows the servlet path format, however, it precedes the query string and starts with a forward slash (/) character. The `pathInfo` parameter can also be set to an empty string.

The addService method can also take a policy parameter that defines the management features for the dynamic endpoint. The policy parameter must be an org.w3c.dom.Element that is compliant with the structure listed in Example 10–1. The definition of this element is listed as a type in the oracle-webservices-10_0.xsd schema.

Example 10–1 Definition of Provider-Policy-Type

```
<xsd:complexType name="provider-policy-type">
   <xsd:all>
      <xsd:element name="runtime" type="serverPortRuntimeType" minOccurs="0"/>
      <xsd:element name="operations" type="serverOperationsType" minOccurs="0"/>
   </xsd:all>
</xsd:complexType>
```

Example 10–2 illustrates a structure defined by this type, as it would appear in an oracle-webservices.xml deployment descriptor.

Example 10–2 XML Structure of Policy Element

```
<policy>
   <runtime enabled="logging,auditing,security,reliability">
      <logging>…</logging>
      <security>…</security>
      <reliability>…</reliability>
   </runtime>
   <operations>
      <operation name="…">
         <runtime>
            <logging>…</logging>
            <auditing>…<auditing>
            <reliability>…</reliability>
         </runtime>
      </operation>
   </operations>
</policy>
```

MessageContext Class

This class contains the definitions of the transport-agnostic message properties. These properties are updated on each call to processMessage. A standard set of constants is defined for the HTTP transport. Properties of the same name are available from the MessageContext class. The MessageContext class also contains the definitions of constants defined for non-HTTP transports.

HTTPConstants Class

This class provides a set of well-defined properties for the HTTP protocol that the container will set on the MessageContext. The Provider instance should ensure that these properties are not overwritten.

Provider Servlet

The ProviderServlet (oracle.j2ee.ws.server.provider.ProviderServlet) handles the routing of requests to the Provider instance. Once deployed, a ProviderServlet listens to a URL for incoming SOAP over HTTP requests. When the servlet receives a SOAP

message, ProviderServlet dispatches it to the processMessage method of the Provider instance.

If you are hosting the Provider in the servlet container, then specify this servlet in the <servlet-class> subelement of <servlet> in your Provider application's web.xml deployment descriptor. The servlet URL is specified in the <url-pattern> element.

Example 10–3 illustrates a fragment of a web.xml file. In the file, ProviderServlet is specified in the <servlet-class> element and the URL to which it listens, /LoggerService, is specified in the <url-pattern> element. This URL pattern is relative to the Web application's context root.

The <servlet-name> element provides the link between this entry and the Provider-specific information in oracle-webservices.xml illustrated in Example 10–5.

Example 10–3 web.xml Fragment, with ProviderServlet and the URL to Which it Listens

```
...
<servlet>
     <servlet-name>LoggerProviderPort</servlet-name>
     <display-name>LoggerProviderPort</display-name>
     <description>JAX-RPC endpoint  Provider Port</description>
     <servlet-class>oracle.j2ee.ws.server.provider.ProviderServlet</servlet-class>
     <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
     <servlet-name>LoggerProviderPort</servlet-name>
     <url-pattern>/LoggerService</url-pattern>
</servlet-mapping>
...
```

Extending ProviderServlet

You can extend ProviderServlet to tailor its functionality to a particular use case. Example 10–4 illustrates how you can extend the servlet's doGet method to allow for the retrieval of specific resources based on the HTTP query string. Note that in this case, if the query string is not processed by the extended servlet, the request should be passed to the parent servlet's doGet method. Note the call to super if the incoming GET request is not processed.

Example 10–4 Extension of a ProviderServlet Method with a Call to super

```
public class ExtendedProviderServlet extends ProviderServlet{

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        String queryString = request.getQueryString();

        if ((queryString != null) &&
                (queryString.equalsIgnoreCase("someotherquery"))) {
            response.setContentType("text/html; charset=UTF-8");
            response.setStatus(HttpServletResponse.SC_OK);
            OutputStream outputStream = response.getOutputStream();
            OutputStreamWriter writer = new OutputStreamWriter(outputStream);
            writer.write( "<HTML><BODY>TEST RESPONSE</BODY></HTML>" );
            writer.close();
        } else {
```

```
            super.doGet(request, response);
...
}
```

You can also extend `ProviderServlet` to use your own test-page generation mechanism instead of using the default test page provided by the servlet.

Making a Web Service Provider-Aware

For the Web service to recognize a Provider, you must add information about your Provider to the `oracle-webservices.xml` and `web.xml` deployment descriptors. The JDeveloper and Application Server Control tools do not provide support for adding this information; you must edit the files by hand.

- Editing the oracle-webservices.xml Deployment Descriptor
- Editing the web.xml Deployment Descriptor

Editing the oracle-webservices.xml Deployment Descriptor

To make the Web service Provider-aware, add the `<provider-description>` clause to the `oracle-webservices.xml` deployment descriptor. Example 10–5 illustrates a fragment of an `oracle-webservices.xml` deployment descriptor with this clause. "Provider Elements in oracle-webservices.xml" describes the elements that can appear in the `<provider-description>` clause.

Example 10–5 oracle-webservices.xml Fragment, with a <provider-description> Clause

```
<oracle-webservices>
 <provider-description>
  <provider-description-name> some name</provider-description-name>
  <wsdl-file>/file location</wsdl-file>
  <wsdl-service-name xmlns:ns_="urn:oracle-ws">ns_:PhotoService</
wsdl-service-name >
  <provider-port>
             <provider-name>LoggerProviderPort</provider-name>
             <wsdl-port xmlns:ns_="urn:oracle-ws">ns_:PhotoIFPort</wsdl-port>
             <implementation-class>oracle.myprovider.JmsRetriever</
implementation-class >
            <servlet-link>LoggerProviderPort</servlet-link>
            <property
name="connection-factory-name">jms/ws/mdb/theQueueConnectionFactory</property>
            <property name="queue-name">jms/ws/mdb/theQueue</property>
             <policy>
               <runtime enabled="logging,auditing,security,reliability">
                  <logging>…</logging>
                  <security>…</security>
                  <reliability>…</reliability>
               </runtime>
               <operations>
                  <operation name="…">
                     <runtime>
                        <logging>…</logging>
                        <auditing>…<auditing>
                        <reliability>…</reliability>
                     </runtime>
                 </operation>
               </operations>
            </policy>
```

```
  </provider-port>
 </provider-description>
</oracle-webservices>
```

Provider Elements in oracle-webservices.xml

The top-level Provider element is `<provider-description>`. This element identifies a collection of Provider ports. Table 10–1 describes the subelements of `<provider-description>`.

Table 10–1 Subelements of the <provider-description> Element

Element Name	Description
<provider-description-name>	(optional) A name for the Provider description.
<wsdl-file>	(optional) Specifies the location of an associated WSDL for the Provider.
<wsdl-service-name>	(optional) Associates the WSDL's service name with the name of a Provider. This element should be provided if a WSDL has more than one service. This element is not required if the WSDL has only one service.
<property>	(optional) Specifies globally-defined properties; that is, properties that are available to all of the defined Provider ports. This element recognizes a `name` attribute.
<provider-port>	(required) Associates a WSDL port with a Web service interface and implementation. It defines the name of the port as a component. It also associates the port with a servlet endpoint. If a `wsdl-file` is not present, the `provider-port` defines a pass-through gateway. For information on the subelements of `<provider-port>`, see Table 10–2.

The `<provider-port>` element associates a WSDL port with a Web service interface and implementation. Table 10–2 describes the subelements of `<provider-port>`.

Table 10–2 Subelements of the <provider-port> Element

Element Name	Description
<provider-name>	(required) The name of the Provider. The `<provider-name>` within the `<provider-port>` must be unique within the deployment descriptor.
<wsdl-port>	(optional) Associates the Provider with a particular port of the WSDL defined by the `wsdl-file` element. This element is not required if no WSDL file is defined.
<expose-wsdl>	(optional) Indicates whether the WSDL should be exposed. Default is `true`.
<expose-testpage>	(optional) Indicates whether the test page should be exposed. Default is `true`.
<implementation-class>	(required) Specifies the name of the class implementing the `oracle.webservices.provider.Provider` interface.
<servlet-link>	(required) Associates a `provider-port` value with a servlet endpoint defined in the `/WEB-INF/web.xml` file.
<max-request-size>	(optional) When a positive value is specified, the service will limit the size of requests to that value (in bytes). Any request that exceeds the maximum length will generate an error. Default is `-1`, which indicates no limit.

Table 10–2 (Cont.) Subelements of the <provider-port> Element

Element Name	Description
<property>	(optional) A locally defined property. If a property with a particular name is defined both locally and globally, then the property defined locally overrides the one defined globally.
<policy>	Defines the Web service management policies for the Provider. Table 10–3 describes the subelements of `<policy>`.

The `<policy>` element defines the Web service management policies for the Provider. Example 10–2 illustrates the structure of the element as it would appear in an `oracle-webservices.xml` deployment descriptor. Table 10–3 describes the subelements of the `<policy>` element.

Table 10–3 Subelements of the <policy> Element

Element Name	Description
<operations>	Contains a sequence of elements, one for each operation. The `<operation>` subelement indicates an individual operation. Each of these subelements contain client-side quality of service configuration for a single operation provided by the referenced Web service.
<runtime>	Contains server-side quality of service runtime information (security, reliability, auditing, and logging) applicable to all the operations provided by the referenced Web service. Each child element contains configuration for a specific feature.

For more information on Web service management policies, see the following resources.

- security—*Oracle Application Server Web Services Security Guide*
- reliability—Chapter 5, "Ensuring Web Service Reliability"
- auditing and logging—Chapter 6, "Auditing and Logging Messages"

Editing the web.xml Deployment Descriptor

To make the Web service Provider-aware, add the `<servlet-class>` element to the `<servlet>` clause and the `<url-pattern>` element to the `<servlet-mapping>` clause. Example 10–6 illustrates a `web.xml` deployment descriptor with these elements. They are described in Table 10–4.

Example 10–6 Provider Elements in the web.xml Deployment Descriptor

```
<web-app><display-name/>
<description/>
<servlet>
     <servlet-name>LoggerProviderPort</servlet-name>
     <display-name>LoggerProviderPort</display-name>
     <description>JAX-RPC endpoint  Provider Port</description>
     <servlet-class>oracle.j2ee.ws.server.provider.ProviderServlet</servlet-class>
     <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
     <servlet-name>LoggerProviderPort</servlet-name>
     <url-pattern>/LoggerService</url-pattern>
</servlet-mapping>
</web-app>
```

Provider Elements in web.xml

Table 10–4 describes the elements which you must add to `web.xml` to make it Provider-aware.

Table 10–4 Provider Elements in web.xml

Element Name	Description
<servlet-class>	Specifies the name of the Provider servlet. The value can be either `oracle.j2ee.ws.server.provider.ProviderServlet` or a subclass as described in "Extending ProviderServlet" on page 10-5.
<url-pattern>	Specifies the URL pattern to use for the Web service. The Provider servlet will listen to URLs whose prefix is specified by this element. The URL pattern should be relative to the Provider servlet context for the Web module. For dynamic pass through mode, the `<url-pattern>` should be "`/*`".

Registering a Provider-Managed Endpoint

The following sections describe how to register a Provider-managed endpoint for your Web service.

- How to Register a Static Provider-Managed Endpoint
- How to Register a Dynamic Provider-Managed Endpoint

How to Register a Static Provider-Managed Endpoint

The following steps provide a general outline of how to register a static Provider-managed endpoint for your Web service. Follow the links for more detained information on each step.

1. Your Provider application must implement the `Provider` interface. The `Provider` methods `init`, `processMessage` and `destroy` must be implemented. See "Provider Interface" on page 10-2 for more information on this interface and its methods.

2. Edit the `oracle-webservices.xml` and `web.xml` deployment descriptors. See "Making a Web Service Provider-Aware" on page 10-6 for more information on editing these descriptors.

3. (optional) Provide a management configuration. The configuration is specified as a `policy` clause in the `oracle-webservices.xml` deployment descriptor. Example 10–2 provides an example of the `policy` clause structure in the deployment descriptor.

4. Package the service. See "Packaging Provider Web Application Provider Classes" on page 10-10 for more information on packaging a service which includes Provider classes.

How to Register a Dynamic Provider-Managed Endpoint

You can register additional Provider-managed endpoints using the `ProviderConfig.addService` method to configure Web service management features for those endpoints. For example, a Business Process Execution Language (BPEL) process manager might choose to register its processes using this mechanism.

Note that the generation, packaging, and deployment of the Provider implementation follows the same procedure outlined for a static Provider-managed endpoint. To

register dynamic Provider-managed endpoints, you must perform the following additional step.

- Register the service by invoking the `addService` method on the `providerConfig` reference. The implementation can optionally pass the policy configuration by using the method's policy parameter.

Example 10–7 illustrates how a Provider can dynamically add an endpoint. The `ADD_SERVICE_POLICY` property declares the Web service management features for the dynamically-added endpoint (in this case, security auditing and reliability). The `addService` invocation is made on the `providerConfig` reference (`config`) passed to the `init` method of the Provider. In this case, the `policy` parameter of the `addService` method passes `XMLUtil.elementFromString(ADD_SERVICE_POLICY)`. Note the WSDL is packaged as part of the WAR file and retrieved as a resource using the `servletContext`.

Example 10–7 Dynamically Registering a Provider-Managed Endpoint

```
...
 private static final String ADD_SERVICE_POLICY = "<policy>\n" +
            "                <runtime
enabled=\"security,auditing,reliability\">\n" +
            "                   <security>\n" +
            "                        <inbound>\n" +
            "                           <verify-username-token
password-type=\"PLAINTEXT\" require-nonce=\"false\" " +
            "                           require-created=\"false\"/>\n" +
            "                        </inbound>\n" +
            "                   </security>\n" +
            "                </runtime>\n" +
            "            </policy>";
...
  public void init(ProviderConfig config) throws ProviderException {
...
URL wsdlURL1 =
((ServletContext)config.getContainerContext()).getResource("/WEB-INF/wsdl/HelloSer
vice.wsdl");
...
        config.addService("/loanservice",wsdlURL1,new
QName("http://hello.demo.oracle/","HttpSoap11"),null,
                XMLUtil.elementFromString( ADD_SERVICE_POLICY ));
...
}
```

Packaging Provider Web Application Provider Classes

Provider Web applications are packaged like any standard Web application. If the Provider application has a WSDL file, store it in the `/WEB-INF/wsdl` directory in the WAR file. Store the Provider classes in the `/classes` directory which is a child of `/WEB-INF`. Example 10–8 illustrates the contents of an EAR file for a Provider Web application.

Example 10–8 Package Structure for a Web Service with a Provider Application

```
provider.ear
    |__/META-INF/application.xml
    |__ provider-web.war
           |__/WEB-INF
                  |__web.xml
```

```
|__oracle-webservices.xml
|__/wsdl/<wsdl_name>.wsdl (optional)
|__/lib
|__/classes
       |__ /provider instance classes
```

Deploying Provider Web Applications

A Provider Web application is deployed (using standard deployment mechanisms) in a J2EE servlet container. The container must have the Provider system classes and the supporting classes available in its runtime.

For more information on deploying Provider Web applications, see the *Oracle Containers for J2EE Deployment Guide*.

Testing Provider Web Application Deployment

The endpoints of a Provider Web application can be exposed by using a statically configured endpoint or added dynamically. If the endpoint provides an associated WSDL file, then you can invoke the Web Service Home Page to see whether deployment was successful. If the endpoint does not provide a WSDL file, a Test Page will not be available.

"Testing Web Service Deployment" in the *Oracle Application Server Web Services Developer's Guide* describes the Web Service Home page.

Managing Provider Endpoints

If a static or a dynamically-added Provider endpoint provides an associated WSDL file, then it can be configured through the Web Services interface in Application Server Control. For more information see the topic "Web Service Home Page" in the Application Server Control on-line help.

Assembling Clients for Provider Web Service Applications

A Provider Web Service application can be accessed by standard J2SE or J2EE static stub or DII clients. For more information on generating clients, see the following chapters:

- assembling a J2EE Web Service client, see "Assembling a J2EE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.
- assembling a J2SE Web service client, see "Assembling a J2SE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.
- "Testing Web Service Deployment" in the *Oracle Application Server Web Services Developer's Guide* describes the
- using WebServicesAssembler commands to assemble Web service artifacts, see "Using WebServicesAssembler" in the *Oracle Application Server Web Services Developer's Guide*.

Additional Information

For more information on:

- assembling a J2EE Web Service client, see "Assembling a J2EE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.
- assembling a J2SE Web service client, see "Assembling a J2SE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.
- testing whether the Provider Web service application deployed successfully, see "Testing Web Service Deployment" in the *Oracle Application Server Web Services Developer's Guide*.
- using WebServicesAssembler commands to assemble Web service artifacts, see "Using WebServicesAssembler" in the *Oracle Application Server Web Services Developer's Guide*.

A

Understanding the Web Services Management Schema

The Web services management policy file, `wsmgmt.xml`, defines the security, reliability, logging, and auditing features that can be assigned to a Web service. On deployment, the contents of the `oracle-webservices.xml` deployment descriptor which relate to management, are extracted from the file and copied to `wsmgmt.xml`.

The `wsmgmt.xml` file resides in the *ORACLE_HOME*`\j2ee\home\config` directory.

Note: Oracle Application Server Web Services strongly recommends that you do not edit or change the contents of `wsmgmt.xml`.

Management features have scope: they can be assigned globally, to a port, and to an operation within a port.

This appendix has these sections.

- Levels of Web Service Management
- wsmgmt.xml Listing

Levels of Web Service Management

The schema for the `wsmgmt.xml` management policy file allows Web service management features to be set at these levels:

- Global Level
- Port Level
- Operation Level

The tags and elements described in the following sections are used in the Web service management schema. For a listing of `wsmgmt.xml`, the XML version of the schema, see Example A–3, "Sample wsmgmt.xml File" on page A-3.

Global Level

The management features that are globally applied to a Web service appear between `<runtime>` elements at the top of `wsmgmt.xml`. If a feature is enabled, it will be applied at runtime. Only security features can be applied globally to a Web service. The security features are delimited by `<security>` elements.

The XML fragment in Example A–1 illustrates the global elements of the `wsmgmt.xml` file, simplified for clarity. The ellipses indicate additional security elements. For a

description of the elements that appear in the `<security>` element, see the *Oracle Application Server Web Services Security Guide*.

Example A–1 Global Elements of the wsmgmt.xml File

```
<wsmgmt>
   <runtime>
       <security>
         ...
       </security>
    </runtime>
</wsmgmt>
```

Port Level

The management features that can be configured for a port are delimited by `<port>` elements. Each Web service can have a number of ports defined in Web service management schema. A port is synonymous with a Java interface. For each port, security, reliability, and logging features can be set.

The security features that appear in the global level are repeated in the port level. Security features set at the port level override security features set at the global level.

The XML fragment in Example A–2 illustrates the port elements of the `wsmgmt.xml` file, simplified for clarity. The ellipses indicate additional elements. For a description of the elements that appear in the `<security>` element, see the *Oracle Application Server Web Services Security Guide*. For a description of the elements that appear in the `<reliability>` element, see "Server-Side Reliability Configuration Elements" on page 5-4. For a description of the elements that appear in the `<logging>` element, see "Server-Side Logging Configuration Elements" on page 6-6.

Example A–2 Port Level Elements in the wsmgmt.xml File

```
<wsmgmt>
   ...
   <port app="my-ear" web="my-war" service="my-service" port="my-service-port">
        <runtime enabled="security">
                <security>
                   ...
                </security>
                <reliability>
                  ...
                </reliability>
                <logging>
                   ...
                </logging>
        </runtime>
   </port>
  ...
<wsmgmt>
```

Table A–1 describes the settings for a port.

Table A–1 Attributes for the Port Element

Attribute Name	Description
port	Maps to the port name in `oracle-webservices.xml` and to the WSDL.

Table A–1 (Cont.) Attributes for the Port Element

Attribute Name	Description
web (or ejb for Web services that expose EJBs)	Maps to the WAR name which is obtained either from `server.xml` or the Web-side XML file.
service	Maps to the Web service name in `oracle-webservices.xml` and the WSDL.
app	Maps to the application name in `server.xml`.

The port component also has a `<runtime>` section where the port-level security, reliability, and logging features can be enabled or disabled. If a feature is enabled, it will be applied at runtime. This is done by including the name of the configuration element in the enabled attribute of the `<port>` element. The configuration of each feature (such as security or logging) can still be present even if it is disabled by virtue of not being present in the enabled attribute. The values of the enabled attribute are delimited by commas. The order of the features on the enabled attribute is currently ignored.

Operation Level

Each port can have multiple operations associated with it. In the `<port>` element, the `<operations>` element indicates the start of the list of operations belonging to a particular port. The `<operation>` element indicates the start of an individual operation.

Table A–2 describes the element for an operation.

Table A–2 General Setting for Operation

Element Name	Description
<operation>	The `name` attribute maps to an operation name in the WSDL.

For each operation, the `wsmgmt.xml` management policy file defines security, reliability, logging, and auditing features available.

The security, reliability, and logging features that are listed in the port level are repeated at the operation level. A security feature set at the operation level overrides the setting made at the port and global level. A reliability or logging feature set at the operation level, in contrast will be referenced by the corresponding feature set at the port level.

wsmgmt.xml Listing

Example A–3 contains a skeleton listing of the server-side Web services management file `wsmgmt.xml`. See the cross references for the listings and description of the elements in the security, reliability, and logging components.

Example A–3 Sample wsmgmt.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<wsmgmt xmlns:oc4j="http://xmlns.oracle.com/oracleas/schema/oc4j-10_0.xsd"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="D:\ade\las\j2ee\src\META-INF\META-INF\
     oracle-webservices-management-10_0.xsd"
schema-major-version="10" schema-minor-version="0">
   <runtime>
```

```
      <security>
          ...
```

For a description and listing of security elements, see the *Oracle Application Server Web Services Security Guide*.

```
          ...
      </security>
   </runtime>
   <port app="String" web="String" service="String" port="String">
      <runtime enabled="String">
         <security>
            ...
```

For a description and listing of security elements, see the *Oracle Application Server Web Services Security Guide*.

```
            ...
          </security>
         <reliability>
             ...
```

For a description and listing of reliability elements, see "Port-Level Reliability Elements on the Server" on page 5-5.

```
              ...
              ...
          </reliability>
         <logging>
             ...
```

For a description and listing of logging elements, see "Port-Level Logging Elements on the Server" on page 6-7.

```
            ...
          </logging>
      </runtime>
      <operations>
         <operation name="String">
            <runtime>
               <security>
                ...
```

For a description and listing of security elements, see the *Oracle Application Server Web Services Security Guide*.

```
               ...
               </security>
               <auditing request="false" response="false" fault="false"/>
```

For a listing and description of operation-level auditing elements, see "Server-Side Auditing Configuration Elements" on page 6-3.

```
               <reliability>
                  ...
```

For a listing and description of operation-level reliability elements, see "Operation Level Reliability Elements on the Server" on page 5-6.

```
                  ...
               </reliability>
               <logging>
                    ...
```

For a listing and description of operation-level logging elements, see "Operation Level Logging Elements on the Server" on page 6-7.

```
                   ...
                 </logging>
            </runtime>
```

```
            </operation>
        </operations>
    </port>
</wsmgmt>
```

B

JAX-RPC Mapping File Descriptor

The JAX-RPC mapping file is a standard XML file that describes the binding between the WSDL and the Java service endpoint interface(s). A number of WebServicesAssembler commands produce a JAX-RPC mapping file as part of their output. You can edit the contents of the file to customize the contents of the generated WSDL or service endpoint interface.

The published version of the JAX-RPC mapping file schema can be found at the following Web site.

`http://java.sun.com/xml/ns/j2ee/j2ee_jaxrpc_mapping_1_1.xsd`

This appendix has the following sections.

- Producing a JAX-RPC Mapping File
- Naming Conventions for the JAX-RPC Mapping File
- Customizing the WSDL or Service Endpoint Interface Contents

Producing a JAX-RPC Mapping File

These WebServicesAssembler commands generate a JAX-RPC mapping file as part of their output. Most of these commands are used when developing a Web service bottom up. For more information on these commands, see "WebServicesAssembler Commands" in the *Oracle Application Server Web Services Developer's Guide*.

- `aqAssemble`
- `assemble`
- `corbaAssemble`
- `dbJavaAssemble`
- `ejbAssemble`
- `jmsAssemble`
- `plsqlAssemble`
- `sqlAssemble`
- `genInterface`
- `genValueTypes` (produces a "partial" JAX-RPC mapping file)
- `genWsdl`

Naming Conventions for the JAX-RPC Mapping File

The generated JAX-RPC mapping file has the following naming pattern:

<*serviceName*>`java-wsdl-mapping.xml`

Where the prefix *serviceName* is the local part of the `qname` of the service.

The JAX-RPC mapping file is typically stored in the same directory as the `webservices.xml` file.

Customizing the WSDL or Service Endpoint Interface Contents

Editing the JAX-RPC mapping file enables you to customize the contents of the generated WSDL (for bottom up Web services development) or the contents of the generated service endpoint interface (for top down Web services development). "Customization Scenarios" describes some common situations where you might want to customize the WSDL or service endpoint interface.

To produce a mapping file that you can edit, WebServicesAssembler provides two commands: `genInterface` and `genWsdl`.

- for general top down Web service development, the `genInterface` command takes a WSDL as input and produces a service endpoint interface and a JAX-RPC mapping file as output.
- for general bottom up Web service development, the `genWsdl` command takes a service endpoint interface as input and produces a WSDL and a JAX-RPC mapping file as output.

After modifying the mapping file, you can use it as input to any of the WebServicesAssembler commands that develop a Web service top down or bottom up. The resulting service endpoint interface or WSDL will reflect the changes that you specified in the mapping file.

This discussion can be summarized in these steps:

1. Generate a mapping file.

   To do this, use the `genInterface` command for top down Web service development or `genWsdl` command for bottom up Web service development.

2. Edit the mapping file.

   The "Customization Scenarios" section describes some of the common ways in which you can edit the JAX-RPC mapping file.

3. Use the modified JAX-RPC mapping file as input to the appropriate WebServicesAssembler command to develop your Web service.

Customization Scenarios

The following sections describe situations where you might want to edit the JAX-RPC mapping file to generate a custom WSDL or service endpoint interface.

- Changing Namespace-to-Java Mappings
- Changing the Names of Java or WSDL Artifacts
- Generating Code into a Single Package from a WSDL with Multiple Namespaces
- Wrapping or Unwrapping Mapping for Document-Literal Operations
- Mapping Between SOAP Headers and Java Method Parameters

Changing Namespace-to-Java Mappings

In the mapping file, the `<package-mapping>` element identifies the mapping between the namespace and the Java package. The `<namespaceURI>` sub-element identifies the namespace and the `<package-type>` sub-element identifies the Java package. A sample `<package-mapping>` element is displayed in Example B–1.

Example B–1 Package to Namespace Mapping Elements in the JAX-RPC Mapping File

```
<java-wsdl-mapping version="1.0">
    <package-mapping>
        <package-type>com.foo.mypackage</package-type>
        <namespaceURI>http://www.foo.com</namespaceURI>
    </package-mapping>
</java-wsdl-mapping>
```

You can change the mapping between the namespace and Java package by changing either of the values within the `<package-mapping>` element. The generated WSDL or service endpoint interface will use the new namespace or Java package value.

The `<package-mapping>` element can appear more than once in the mapping file.

Changing the Names of Java or WSDL Artifacts

In the mapping file, you can change the names of Java or WSDL artifacts. For example, for top down Web service development, assume that you want to change the mapping of the `wsdl-say-hello` operation in your given WSDL. You can use the WSDL as input to `genInterface`, then edit the resulting JAX-RPC mapping file to map the `wsdl-say-hello` WSDL operation from its default to a `javaSayHello` method that you provide.

Similarly, for bottom up Web service development, you can use a given service endpoint interface, such as `MyServiceEndpoint`, as input to `genWsdl`. You can then edit the resulting JAX-RPC mapping file to map the service endpoint interface to a port-type, such as `my-service-port-type`, in the WSDL.

See the JAX-RPC mapping file schema for more information on where the names of the Java and WSDL artifacts can be changed.

`http://java.sun.com/xml/ns/j2ee/j2ee_jaxrpc_mapping_1_1.xsd`

Generating Code into a Single Package from a WSDL with Multiple Namespaces

If you are working with a WSDL with multiple namespaces, WebServicesAssembler places all generated code into a package that corresponds to the namespace of the WSDL and schema value types. If the WSDL contains multiple namespaces, then one package will be created for each namespace. Importing multiple packages can be inconvenient when writing your implementation class.

WebServicesAssembler does not provide an argument that allows all code to be generated in the same package. For example, the `packageName` argument effects only the package name for service endpoint interface and any schema types that have the same target namespace as the WSDL. If there are schema value types in a different namespace, then WebServicesAssembler generates them into a different package by default.

You can direct WebServicesAssembler to generate a single package by editing the JAX-RPC mapping file. For each namespace, the file contains a `<package-mapping>` clause with a `<package-type>` and `<namespaceURI>` element. These elements identify the mapping between the namespace and a Java package. Edit each instance

of the `<package-type>` element to enter the name of the single package where you want the generated code to reside.

Example B–2 illustrates the contents of a JAX-RPC mapping file that has been edited to place the generated code from three different namespaces into the package `MyPackage`. The `<package-type>` and `<namespaceURI>` elements are highlighted in bold.

Example B–2 Mapping File that Specifies a Single Package for Multiple Namespaces

```
<java-wsdl-mapping version="1.1">
    <package-mapping>
        <package-type>MyPackage</package-type>
        <namespaceURI>namespace1 in the WSDL</namespaceURI>
    </package-mapping>
    <package-mapping>
        <package-type>MyPackage</package-type>
        <namespaceURI>namespace2 in the WSDL</namespaceURI>
    </package-mapping>
     ...
    <package-mapping>
        <package-type>MyPackage</package-type>
        <namespaceURI>namespaceN in the WSDL</namespaceURI>
    </package-mapping>
</java-wsdl-mapping>
```

How to Generate Code into a Single Package Using a WSDL with Multiple Namespaces

The following steps summarize how you can direct WebservicesAssembler to generate code into a single package when the WSDL contains multiple namespaces.

1. Generate a JAX-RPC mapping file for the Web service.

   Use the `genInterface` command if you are assembling a Web service top down or the `genWsdl` command if you are assembling a Web service bottom up. You can also write your own JAX-RPC mapping file by hand.

2. Edit the JAX-RPC mapping file so that each instance of the `<package-type>` element in the `<package-mapping>` clause contains the name of the single package where you want the generated code to reside.

3. Assemble the Web service using the appropriate `*Assemble` command. Use the `mappingFileName` argument to specify the edited JAX-RPC mapping file.

   All of the generated code will reside in the same package.

"Default Algorithms to Map Between Target WSDL Namespaces and Java Package Names" in the *Oracle Application Server Web Services Developer's Guide* provides more information on how WebServicesAssembler constructs a default type namespace from the package name.

Wrapping or Unwrapping Mapping for Document-Literal Operations

The `<service-endpoint-method-mapping>` element defines the mapping of Java methods to WSDL operations. If a `<wrapped-element/>` is specified under this element and you are working with document-literal WSDL operations, then each Java method parameter that is not mapped to a SOAP header is mapped to a sub-element of a `complexType`. This `complexType` is used as the type of a wrapper element. Any Java method parameters that are not mapped to a SOAP header, must be mapped to a sub-element of a `complexType` (that is, they must be wrapped).

Mapping Between SOAP Headers and Java Method Parameters

The `<wsdl-message-mapping>` element defines the unique mapping from a specific Java method parameter to a specific message and its part. Parts within a message context are uniquely identified with their names.

If `<soap-header/>` is specified under a `<wsdl-message-mapping>` element, then:

- this Java method parameter is exposed as a SOAP header for bottom up development
- the corresponding SOAP header is mapped to a Java method parameter for top down development

C

Web Service MBeans

An MBean, or "managed bean", is a Java object that represents a manageable resource in a distributed environment, such as an application, a service, a component, or a device.

An MBean has an exposed interface that allows a management client to control the resource. The interface is comprised of:

- Attributes, values of any type that the management client can get or set. Attributes are analogous to properties set on a JavaBean.
- Operations, methods with any signature and any return type that the client can invoke.
- Notifications that can be generated when specific events occur.

The individual MBean descriptions provide more detailed information on the attributes, operations, and notifications that are available for them.

The following sections describe the MBeans that pertain to Oracle Application Server Web Services and how they are initialized within its environment.

- Web Services MBean Descriptions
- Initializing MBeans

Web Services MBean Descriptions

This section provides a description of the components that define the functionality of an MBean and a summary of the MBeans that are available for OracleAS Web Services.

- Understanding MBean Components
- WebServicePort
- WebServiceOperation
- WSMServiceConfig
- WSMOperationConfig
- WSMHandlerGlobalConfig
- WSMHandlerServiceConfig
- WSMHandlerOperationConfig

Understanding MBean Components

The name of an MBean consists of a number of components, such as J2EEApplication, WebServicePort, handler, and so on. The value of the components are set when the MBean is registered with the MBean server. The full name of an MBean will be displayed in Applications Server Control when you access it. Table C–1 describes the values of the MBean components.

Table C–1 Definitions of JMX MBean Name Components

Component Value	Description
{application}	The name given to the application (that is, the EAR file) when it was deployed. This value can be found in the <application> element in the *ORACLE_HOME*/j2ee/home/config/server.xml file.
{web-module}	The name given to the Web module within the application's deployment descriptor. This value can be found as the value of the ID attribute in the <web-module> element of the orion-application.xml file in the META-INF directory of an application.
{service}	The name of the Web service as described by the service's deployment descriptor. This value can be found in the WEB-INF/oracle-webservices.xml file of a deployed Web module. The service name is the value of name attribute of each <webservice-description> element.
{port}	The name of the Web service port as described by the service's deployment descriptor. This value can be found in the WEB-INF/oracle-webservices.xml file of a deployed Web module. The port name is the value of the name attribute of each <port-component> element.
{operation}	The name of the Web service operation as described by the service's deployment descriptor. This value can be found in the WEB-INF/oracle-webservices.xml file of a deployed Web module. The port name is the value of the name attribute of each of each <operation> element.
{interceptor}	This value identifies the interceptor. The possible values currently are security, reliability, auditing, logging and owsm (Oracle Web Service Manager).

WebServicePort

*:j2eeType=WebServicePort,J2EEApplication={*application*}, WebModule={*web-module*},WebService={*service*},name={*port*},*

This MBean provides for the management of a Web service endpoint in the run-time. Specifically, this MBean provides information about the Web service port's structure, state, and performance. It also allows for the endpoint to be temporarily stopped and restarted.

Attributes: address, implementationType, path, state, stats, style, wsdl

Operations: start, stop

Statistics: ActiveRange, TotalFault, ServiceTime

Notifications: j2ee.state.running, j2ee.state.stopped

WebServiceOperation

*:j2eeType=WebServiceOperation,J2EEApplication={*application*}, WebModule={*web-module*},WebService={*service*},WebServicePort={*port*}, name={*operation*}

This MBean provides for the observation and management of each operation of a Web service endpoint. Many of the attributes also provide information about the implementation of the operation.

Attributes: `inputEncoding`, `outputEncoding`, `overloaded`, `sampleRequest`, `sampleResponse`, `stats`, `testPagePath`, `testPageURL`

Statistics: `RequestSize`, `ResponseSize`, `ActiveRange`, `FaultCount`, `ServiceTime`

WSMServiceConfig

`*:j2eeType=WSMServiceConfig`, `J2EEApplication=`{*application*}, `WebModule=`{*web-module*}, `WebService=`{*service*}, `WebServicePort=`{*port*}, `*`

This MBean is used only to create a hierarchy of MBeans within the system. It represents the port level configuration for all interceptors associated with each Web service port. There are however no useful attributes or operations provided by this MBean.

WSMOperationConfig

`*:j2eeType=WSMOperationConfig`, `J2EEApplication=`{*application*}, `WebModule=`{*web-module*}, `WebService=`{*service*}, `WebServicePort=`{*port*}, `operation=`{*operation*}, `*`

This MBean is used only to create a hierarchy of MBeans within the system. It represents the operation level configuration for all interceptors associated with each Web service operation. There are, however, no useful attributes or operations provided by this MBean.

WSMHandlerGlobalConfig

`*:j2eeType=WSMHandlerGlobalConfig`, `handler=`{*interceptor*}, `*`

This MBean provides the ability to get, set, and validate the global configuration settings for each individual interceptor. Currently only the security and reliability interceptors utilize global configuration.

Attributes: `stagedConfig`, `deployedConfig`

Operations: `validateConfig`

WSMHandlerServiceConfig

`*:j2eeType=WSMHandlerServiceConfig`, `J2EEApplication=`{*application*}, `WebModule=`{*web-module*}, `WebService=`{*service*}, `WebServicePort=`{*port*}, `handler=`{*interceptor*}, `*`

This MBean provides access to the port level configuration of each interceptor. Each interceptor may also be enabled or disabled for the Web service port for the MBean by using the `stagedEnabled` attribute. The other attributes and operation allow the configuration to be retrieved, set and validated.

Attributes: `stagedEnabled`, `stagedConfig`, `deployedConfig`

Operations: `validateConfig`

WSMHandlerOperationConfig

`*:j2eeType=WSMHandlerOperationConfig`, `J2EEApplication=`{*application*}, `WebModule=`{*web-module*}, `WebService=`{*service*}, `WebServicePort=`{*port*}, `handler=`{*interceptor*}, `operation=`{*operation*}, `*`

This MBean provides access to the operation level configuration of each interceptor. This MBean can be used to retrieve, set and validate the configuration.

Attributes: `stagedConfig`, `deployedConfig`

Operations: `validateConfig`

Initializing MBeans

The OracleAS Web Services runtime for the current release is implemented as a servlet; it relies upon the servlet container for the management of its life cycle. JMX MBeans are created and registered as part of the Servlet `init` method of the Web service runtime. Depending on the configuration of the `web.xml` deployment descriptor in the J2EE Web Archive (that is, the WAR file), the servlet container may not initialize the Web service servlet until the first request to the servlet is encountered.

To instruct the servlet container to initialize the servlet when the Web service is deployed or the container starts, set the `load-on-startup` element in the `web.xml` deployment descriptor to a nonzero value. For example:

```
<load-on-startup>1</load-on-startup>
```

This will result in the JMX MBean for the Web service being available as early as possible.

OracleAS Web Services recommends that all Web services deployed on the current release provide the `<load-on-startup>1</load-on-startup>` setting in their `web.xml` deployment descriptor. This will ensure that the Application Server Control application works correctly.

D

Mapping Java Types to XML and WSDL Types

This appendix describes the mappings between Java types and XML types that are supported under Oracle Application Server Web Services.

- Mapping Java Types to XML Types
- Mapping Java Primitive Types to XML Types
- OC4J Support for Java Value Types
- Mapping Support for Arrays
- Mapping Java Collection Classes to XML Types
- Support for Java Beans Components

Mapping Java Types to XML Types

Table D–1 describes the mapping between Java data types and XMl data types that OracleAS Web Services supports.

- The **Java Type** column lists the supported Java types.
- The **XML Type: RPC-Literal, Document-Literal** column lists the mapping between the Java type and the corresponding XML type for RPC-literal and document-literal format.
- The **XML Type: RPC-Encoded** column lists the mapping between the Java type and the corresponding RPC-encoded format.

For example, `java.lang.String` is mapped to the `xsd:string` XML data type. You should note that not every Java type can be used as a method parameter or return type.

The namespace for the document-literal and RCP-literal formats uses an `xsd` prefix and has the following definition.

```
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

The SOAP 1.1 namespace for the RPC-encoded format uses a `soap-enc` prefix and has the following definition.

```
xmlns:soap-enc="http://schemas.xmlsoap.org/soap/encoding
```

Table D–1 Mapping Java Types to XML Types

Java Types	XML Type: RPC-Literal, Document-Literal	XML Type: RPC-Encoded
java.lang.Boolean	xsd:boolean	soap-enc:boolean
java.lang.Byte	xsd:byte	soap-enc:byte
java.lang.Double	xsd:double	soap-enc:double
java.lang.Float	xsd:float	soap-enc:float
java.lang.Integer	xsd:integer	soap-enc:integer
java.lang.Long	xsd:long	soap-enc:long
java.lang.Short	xsd:short	soap-enc:short
java.lang.String	xsd:string	xsd:string
java.math.BigDecimal	xsd:decimal	xsd:decimal
java.math.BigInteger	xsd:int	xsd:int
java.net.URI	xsd:anyURI	xsd:anyURI
java.util.Calendar	xsd:dateTime	xsd:dateTime
java.util.Date	xsd:dateTime	xsd:dateTime
java.xml.QName	xsd:QName	xsd:QName

Using Java Null Values in Bottom Up Mapping

It is good practice not to rely on any mapping from Java `null` values to XML `nil` values (see also Chapter 1, "Ensuring Interoperable Web Services"). Special care needs to be taken in the RPC case.

- In RPC-literal message format, `Null` parameter values cannot be mapped to `nil` values.
- In RPC-encoded message format, SOAP-encoded type values can preserve `nil`.
- In all cases where a value type (Java Bean) is used as a parameter, you should not use Java `null` values.

Mapping Java Primitive Types to XML Types

Table D–2 describes the mapping between Java primitive types and XML types that OracleAS Web Services supports. The types listed in the **XML Type** column applies to RPC-encoded, RPC-literal, and document-literal formats.

Table D–2 Mapping Java Primitive Types to XML Types

Java Primitive Type	XML Type
boolean	xsd:boolean
byte	xsd:byte
double	xsd:double
float	xsd:float
int	xsd:int
long	xsd:long
short	xsd:short

OC4J Support for Java Value Types

Java value types can contain a number of attributes. The following list describes the requirements on the Java class corresponding to the Java value type.

- the Java class must have an empty public constructor
- the Java class must not implement `java.rmi.Remote`
- the Java class must contain attributes of supported types
  - an attribute can be a public field that is not `final` or `transient`, or
  - an attribute can be represented by public `get`ter and `set`ter methods
- If the Java class contains a single array-valued attribute (such as `int[]` or `String[]`) then the name of the class must not contain the word "Array". You might want to rename the class to "Ary", "List", or something similar.

Representing a Java Value Type as a Schema Type

A Java value type can be represented by a schema type. The schema type that corresponds to a Java value type has the following general format.

```
<complexType name="classname of the value type">
  <sequence>
     ...attributes...
  </sequence>
</complexType>
```

Mapping Support for Arrays

The following sections describe the mapping support OracleAS Web Services offers for arrays.

- All Formats
- Document-Literal and RPC-Literal Formats
- RPC-Encoded Format

All Formats

Byte arrays are a special case. The Java type `byte[]` maps to the XML type `xsd:base64Binary` for document-literal, RPC-literal, and RPC-encoded formats.

Document-Literal and RPC-Literal Formats

Table D–3 provides examples of how OracleAS Web Services supports arrays with members of supported types in the RPC-literal and document-literal formats. For example, an array containing elements of type `int` maps to the XML type `xsd:int` with the `minOccurs` and `maxOccurs` attributes. Table D–1 lists the supported type mappings for RPC-literal and document-literal formats.

Note: Byte arrays are an exception to the list of supported types. Table D–4 lists the XML type mapping for byte arrays.

Table D–3 Mapping Arrays Containing Java Types to XML Types for Document-Literal and RPC-Literal

Java Type	XML Type: RPC-Literal, Document-Literal	Additional Properties for the XML Type
int[]	xsd:int	minOccurs="0", maxOccurs="unbounded"
String[]	xsd:string	minOccurs="0", maxOccurs="unbounded"

Multi-dimensional arrays, such as `Double[][]`, are not supported for the document-literal or RPC-literal formats in the current release. However, it is possible to work around this limitation by wrapping the array in a Java value type and then using an array of these value types. "OC4J Support for Java Value Types" on page D-3 describes the requirements on Java value types. Note the restriction on the names for value types described in this section.

RPC-Encoded Format

For the RPC-encoded format, JAX-RPC supports arrays that contain members of supported types. Table D–1 lists the supported types for RPC-encoded.

Note: Byte arrays are an exception to the list of supported types. Table D–4 lists the XML type mapping for byte arrays.

Example D–1 illustrates how a `String[]` array is represented in the WSDL. The `String[]` array is the target of the WSDL construct `wsdl:arrayType`, where `wsdl` is the namespace for the WSDL 1.1 schema prefix. This namespace has the following definition.

```
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
```

Example D–1 WSDL Definition of an Array of Strings for the RPC-Encoded Format

```
<complexType name="ArrayOfString">
  <complexContent>
    <restriction base="soap-enc:Array">
      <attribute ref="soap-enc:arrayType" wsdl:arrayType="string[]"/>
    </restriction>
  </complexContent>
</complexType>
```

Example D–2 illustrates the definition of a multidimensional array. The array of strings (`ArrayofString[]`) defined in the previous example is used in the `wsdl:arrayType="tns:ArrayofString[]"` attribute (highlighted in bold) to create an array which contains arrays of strings. The `tns` is a WSDL construct that defines a local namespace.

Example D–2 WSDL Definition of a Multi-Dimensional Array of an Array of Strings

```
<complexType name="ArrayOfArrayOfString">
  <complexContent>
    <restriction base="soap-enc:Array">
      <attribute ref="soap-enc:arrayType" wsdl:arrayType="tns:ArrayOfString[]"/>
    </restriction>
  </complexContent>
```

```
</complexType>
```

Mapping Java Collection Classes to XML Types

Table D–4 and Table D–5 describe the mapping between Java `Collection` and `Map` data types and the XML types that OracleAS Web Services supports. This is an Oracle-proprietary mapping and may not be compatible with data type support provided by other vendors. "Definitions for Oracle-Proprietary Collection Data Types" on page D-6 provides more information on how these types are defined.

Note that in OracleAS Web Services, you can nest `Collections` and `Maps` data types. For example, you can specify a `Collection` as an item within a `Collection`.

The Java `Collection` and `Map` data types are supported in RPC-literal, document-literal, and RPC-encoded formats. These data types map to the same XML types for the three formats. The Oracle-proprietary collection types use the following proprietary namespace for RPC-literal and document-literal formats.

```
xmlns:owi="http://www.oracle.com/webservices/internal/literal"
```

The Oracle-proprietary collection types use the following proprietary namespace for the RPC-encoded format.

```
xmlns:owi="http://www.oracle.com/webservices/internal"
```

Table D–4 Mapping Java Collection Classes to XML Types

Java Type	Java Class (java.util)	XML Type
Collection classes:	**Collection**	owi:collection
	List	owi:list
	ArrayList	owi:arrayList
	LinkedList	owi:linkedList
	Stack	owi:stack
	Vector	owi:vector
Set classes:	**Set**	owi:set
	HashSet	owi:hashSet
	TreeSet	owi:treeSet
Map classes:	**Map**	owi:map
	HashMap	owi:hashmap
	Hashtable	owi:hashtable
	Properties	owi:properties
	TreeMap	owi:treemap

Limitations on Using Collection and Map Data Types

This section describes the limitations on using `Collection` and `Map` data types in OracleAS Web Services.

- Instances of `java.lang.Object` cannot be used in `Collections` or `Maps` because there is no XML schema construct to describe the instances of `java.lang.Object`.

- If you use instances of non-built in types (such as Beans) in a `Collection`, then you must generate schemas for them. See "Oracle-Specific Type Support" in the *Oracle Application Server Web Services Developer's Guide* for more information on this topic.
- In J2SE 5.0, `Collections` and `Maps` can be parameterized. For example, `Collection<Integer>`. However, the code generated by WebServicesAssembler does not support parameterized `Collection` and `Map`, because OracleAS Web Services does not preserve parameterized type metadata in the generated schema.

The following limitations apply to `Collection` and `Map` data types when you are working with document-literal and RPC-literal message formats.

- Object graph structure is NOT preserved between serialization and deserialization. See "Object Graph" on page E-10 for more information on limitations when working with object graphs.
- Recursive or circular reference structures return an error. This is illustrated in the following code samples.

```
// the following lines of code will return an error
ArrayList a = new ArrayList();
a.add(a);
```

  The following code sample illustrates how `Object` is not supported as a `Collection` element.

```
// the following lines of code will return an error
ArrayList a = new ArrayList();
a.add(new Object());
```

Definitions for Oracle-Proprietary Collection Data Types

Table D–5 provides more information on how the Oracle-proprietary XML types for collections are defined.

Table D–5 Additional Information about Oracle-Proprietary XML Types

Proprietary XML Type	Additional Information about the XML Type

owi:collection	The following is the XML schema type of the `owi:collection` type for the RPC-literal and document-literal formats. To construct the type definition for the RPC-encoded format, replace `type="anyType"` with an array of type `xsd:anyType`. Example D–1 provides an example of creating an array of a data type.
	`<complexType name="collection">`
	`  <sequence>`
	`    <element name="item" type="anyType" minOccurs="0" maxOccurs="unbounded"/>`
	`  </sequence>`
	`</complexType>`

owi:list	a simple extension of `owi:collection`
owi:arrayList	a simple extension of `owi:list`
owi:linkedList	a simple extension of `owi:list`
owi:stack	a simple extension of `owi:list`
owi:vector	a simple extension of `owi:list`

Table D–5 (Cont.) Additional Information about Oracle-Proprietary XML Types

Proprietary XML Type	Additional Information about the XML Type
owi:set	a simple extension of `owi:collection`
owi:hashSet	a simple extension of `owi:set`
owi:treeSet	a simple extension of `owi:set`

owi:map	The following is the XML schema type of the `owi:map` type for the RPC-literal and document-literal formats. To construct the type definition for the RPC-encoded format, replace `type="owi:mapEntry"` with an array of `owi:mapEntry`. Example D–1 provides an example of creating an array of a data type.
	`<complexType name="map">`
	`  <sequence>`
	`    <element name="mapEntry" `**`type="owi:mapEntry"`**
	`             minOccurs="0" maxOccurs="unbounded"/>`
	`  </sequence>`
	`</complexType>`
	The `owi:mapEntry` type used in the preceding code sample has the following definition.
	`<complexType name="mapEntry">`
	`  <sequence>`
	`    <element name="key" type="anyType"/>`
	`    <element name="value" type="anyType"/>`
	`  </sequence>`
	`</complexType>`

owi:hashMap	a simple extension of `owi:map`
owi:hashtable	a simple extension of `owi:map`
owi:properties	a simple extension of `owi:map`
owi:treeMap	a simple extension of `owi:map`

Support for Java Beans Components

Java beans can have any number of properties. Each property must have a `getter` and `setter` method. The properties must belong to supported Java types.

E Troubleshooting

This appendix provides solutions to possible problems that may occur when working with Oracle Application Server Web Services. The section titles in this appendix correspond to chapter titles in the *Oracle Application Server Web Services Developer's Guide* and the *Oracle Application Server Advanced Web Services Developer's Guide*.

OracleAS Web Services Messages

Cannot Serialize or Deserialize Array-Valued Elements to Collection Types

If you use a Java Collection type (such as `java.util.Map`, `java.util.Collection`, or a subclass of these) as a parameter or return type in your RPC-encoded Web service, then the runtime cannot properly serialize or deserialize array-valued elements to these collection parameters.

To ensure that serializers and deserializers are registered for Java array types when using the RPC-encoded message format, create a Java value type to represent each Java array.

1. Create a Java value type for each Java array type that you want to use.

   > **Note:** Ensure that the Java value type does not contain the word "Array" in its name. "Array" is a recognized pattern.

   The following example represents the contents of the `demo/StringAry.java` file. A wrapper class, `StringAry`, represents the Java `String[]` array. Note that the name of the class uses the suffix "Ary".

   ```
   package demo;
   public class StringAry
   { public StringAry() { }
     public String[] getValue() { return m_value; }
     public void setValue(String[] value) { m_value=value; }
     private String[] m_value;
   }
   ```

2. Ensure that the proper serializers and deserializers are registered for all of your value types.

   To do this, use the `valueType` argument when you assemble the Web service. In the following example, the argument specifies the `demo/StringAry.java` file created in Step 1.

   ```
   java wsa.jar -assemble -valueType demo.StringAry ...
   ```

3. Use the value types you defined for setting and retrieving array-valued elements in your collection type parameter.

   For example, assume that you have the following class definition.

```
package demo;
public class Service extends java.rmi.Remote
{ java.util.Map getMap(String input)throws java.rmi.RemoteException
  { ... }
}
```

   You can write the following code to return a `String[]` value as one of the elements in the map.

```
HashMap map = new HashMap();
String[] str_array = new String[]{"a","b","c"};
StringAry sa = new StringAry();
sa.setValue(str_array);
map.put("myArray", sa);
return map;
```

Errors Occur When Publishing a Web Service that Uses Multi-Dimensional Arrays

An error occurs when attempting to publish a Web service that uses multidimensional arrays. For example, an error can be returned when you attempt to publish a Java class that contains a method which takes a multidimensional array as an input or as a return argument.

There are two possible solutions to this problem:

- Create a Java Bean for each dimension of the array
- Use RPC-encoded message format to publish the Web service

Creating a Java Bean for each Dimensional of the Array: You can wrap each dimension of the array into a Java value type and work around the limitation.

> **Note:** Ensure that the Java value type does not contain the word "Array" in its name. "Array" is a recognized pattern.

In the following example, the `public static class StringAry` wraps the inner array of strings. The `public StringAry[]` represents an array of the inner array. That is, it contains an array of `String` Java value types. Note that the code sample uses the suffix "Ary".

```
package demo;
public interface SampleItf extends java.rmi.Remote

 // wrap the inner array as a Java value type
{ public static class StringAry
  { public StringAry() { }
    public String[] getValue() { return m_value; }
    public void setValue(String[] value) { m_value=value; }
    private String[] m_value;
  }

  // create an  array of the inner array elements
  public StringAry[] echoString2(StringAry[] input)
        throws java.rmi.RemoteException;
}
```

The `Service` class illustrates how you can then publish the `StringAry[]` array of `String` Java value types.

```
package demo;
public class Sample implements java.rmi.Remote, SampleItf
{ public SampleItf.StringAry[] echoString2(SampleItf.StringAry[] input)
         throws java.rmi.RemoteException
  { return input; }
}
```

Using RPC-Encoded Style to Publish a Web Service: You can use the RPC-encoded style to publish a Web service that uses multidimensional arrays. For example:

```
package demo;
public interface SampleItf extends java.rmi.Remote
{ public String[][] echoString2(String[][] input)
         throws java.rmi.RemoteException;
}


package demo;
public class Sample implements java.rmi.Remote, SampleItf
{ public String[][] echoString2(String[][] input)
         throws java.rmi.RemoteException
  { return input; }
}
```

Restrictions on RPC-Encoded Format and Data Binding

OracleAS Web Services does not support the combination of RPC-encoded message formats and `databinding=false`. This combination is not considered a "best practice" within the industry.

Document-Encoded Message Format is not Supported by OracleAS Web Services

Even though the combination of `style="document"` and `use="encoded"` is valid according to the SOAP specification, it is not supported by any of the major Web Services platforms, including OracleAS Web Services.

Document-Literal Bare Message Format is Limited to One Input Part

OracleAS Web Services supports only one part as input in the bare case. All other input parameters must be mapped into SOAP header parts.

Serialization of BigDecimal Values May Introduce Rounding Errors

There are several constructors available for `java.Math.BigDecimal`. The constructors can take the following types of input.

- a double-precision floating point
- an integer and a scale factor
- a String representation of a decimal number

You should be careful when you use the `BigDecimal(double)` constructor. It can allow rounding errors to enter into your calculations. Instead, use the `integer` or `String`-based constructors.

For example, consider the following statements which take the value 123.45.

```
...
double d = 1234.45;
```

```
System.out.println(d);
System.out.println(new BigDecimal(d));
...
```

These statements produce the following output. The second value might not be the value you expected.

```
1234.45
1234.450000000000045474735088646411895751953125
```

Assembling Web Services from a WSDL

Restrictions on Using Document Literal Message Formats

If you attempt to assemble a Web service top down that uses a document-literal message format, WebServicesAssembler will return a warning if it detects two or more operations in the WSDL that use the same input message. This is because the OC4J runtime will not be able to distinguish which method is being invoked.

For example, the following WSDL fragment will cause WebServicesAssembler to return a warning. The fragment defines the `addRelationship` and `addRelationship3` operations. Each of these operations use the `addRelationshipRequest` input message.

```
...
<operation name="addRelationship">
     <input name="addRelationship1Request"
message="tns:addRelationship1Request"/>
            <output name="addRelationship1Response"
message="tns:addRelationship1Response"/>
        </operation>
         <operation name="addRelationship3">
            <input name="addRelationship1Request"
message="tns:addRelationship1Request"/>
            <output name="addRelationship1Response"
message="tns:addRelationship1Response"/>
        </operation>
...
```

If you were to invoke the `addRelationship` operation from your client, then depending on the order in which the operations appear in your implementation class, either `addRelationship` or `addRelationship3` would be invoked.

Assembling Web Services from Java Classes

Limitations on Stateful Web Services

The support that OracleAS Web Services offers for stateful Web services is limited to services based on Java classes. These services contain Oracle-proprietary extensions and you should not consider them to be interoperable unless the service provider makes scopes with the same semantics available.

The support that OracleAS Web Services offers for stateful Web services is HTTP-based. Stateful Web services will work only for SOAP/HTTP endpoints and will not work for SOAP/JMS endpoints.

Assembling Web Services from Java Classes—Differences Between Releases 10.1.3 and 10.1.2

Note the following differences between Oracle Web Services release 10.1.2 (and earlier) and release 10.1.3.

- In release 10.1.2 there was no requirement to extend `RemoteInterface` or for methods to throw `RemoteException`. This is now required in release 10.1.3.
- In release 10.1.2 it was possible to publish a class by itself without providing an interface. In release 10.1.3 you must provide an interface to publish a class.

Assembling Web Services From EJBs

Setting the Transaction Demarcation for EJBs

An EJB exposed as a Web service should not have `TX_REQUIRED` or `TX_MANDATORY` set as its transaction demarcation.

Assembling Web Services with JMS Destinations

Supported Types for Message Payloads

For JMS endpoint Web services, OracleAS Web Services supports only instances of `java.lang.String` or `javax.xml.soap.SOAPElement` as the payload of JMS messages.

JMS Properties in the SOAP Message Header

Only a limited number of JMS properties can be transmitted by the SOAP header. If the value of the `genJmsPropertyHeader` argument is `true` (default), then the following JMS properties can be transmitted by the SOAP header.

- message-ID
- correlation-ID
- the reply-to-destination, including its name, type, and factory

Developing Web Services From Database Resources

Datatype Restrictions

- Streams are not supported in Oracle Streams AQ Web services.
- SQL Type `SYS.ANYDATA` is not supported in PL/SQL Web services.
- `REF CURSOR` as a parameter is not supported in PL/SQL Web services.
- `REF CURSOR` returned as Oracle `WebRowSet` and `XDB RowSet` does not support complex types in the result.
- Due to a limitation in JDBC, PL/SQL stored procedures do not support the following SQL types as `OUT` or `INOUT` parameters.
  - `char` types, including `char`, `character`, and `nchar`
  - `long` types including `long` and `long raw`

Differences Between Database Web Services for 10.1.3 and Earlier Releases

A Web service client written for a database Web service generated under release 9.0.4 or 10.1.2, will fail if you try to use it against a database Web service generated bottom up under release 10.1.3. This will be true even if the PL/SQL structures have remained the same.

One of the reasons for this is that the SQL collection type was mapped into a complex type with a single array property in releases 9.0.4 and 10.1.2. In release 10.1.3, it is mapped directly into array instead.

If you regenerate the Web service client, you will have to rewrite the client code. This is because the regenerated code will now be employing an `array[]` instead of a `BeanWrappingArray`.

Assembling Web Services with Annotations

Web Service Metadata Features that are Not Supported

There are parts of the Web Services Metadata for the Java Platform specification that OracleAS Web Services does not support. For example, OracleAS Web Services does not support the "Start With WSDL" or "Start With WSDL and Java" modes defined in sections 2.2.2 and 2.2.3 of the "Java Architecture for XML Binding" (JAXB) specification. OracleAS Web Services supports only the "Start With Java" mode.

If you use the `assemble` or the `genWsdl` WebServicesAssembler commands to generate a WSDL to use with J2SE 5.0 Web Service annotations, you must specify them differently than if you were using them to process files that do not contain annotations. See the "assemble" and "genWsdl" sections of the "Using WebServicesAssembler" chapter in the *Oracle Application Server Web Services Developer's Guide* for more information on using these commands to generate WSDLs for use with J2SE 5.0 Web Service annotations.

Assembling REST Web Services

Restrictions on REST Web Services Support

The following list describes the limitations in OracleAS Web Services support for REST Web Services.

- REST support is available only for Web service applications with literal operations (both request and response should be literal).
- HTTP GET is supported only for Web service operations without (required) complex parameters.
- Some browsers limit the size of the HTTP GET URL. Try to keep the size of the URL small by using a limited number of parameters and short parameter values.
- REST Web services send only simple XML messages. You cannot send messages with attachments.
- Many management features, such as security and reliability, are not available with REST Web services. This is because SOAP headers, which are typically used to carry this information, cannot be used with REST invocations of services.
- REST invocations cannot be made from generated Stubs or DII clients. Invocations from those clients will be made in SOAP.
- There is no REST support for the Provider framework.

- Operation names in REST cannot contain multibyte characters.
- REST services cannot be managed through Application Server Control.

Testing Web Service Deployment

The Web Service Home Page has the following limitations:

- The Web Service Home Page offers only basic support for WS-Security. The editors can enter only a username and password into the SOAP envelope. To enter any other complex or advanced WS-Security features, such as encryption and signing, in the SOAP request, you must edit the request directly in the Invocation Page.
- The Web Service Home Page does not allow you to upload attachments.
- You cannot invoke WSIF services using the test page.
- WSDL files that contain proprietary extensions may not work properly in the Web Service Home Page. For example, services that use JMS as a transport cannot be tested by using the Home Page.

Assembling a J2EE Web Service Client

Client Applications and Thread Usage

If the client application creates its own threads for its processing (for example, if it enables an asynchronous call using a separate thread), the application server must be started with the `-userThreads` option.

```
java -jar oc4j.jar -userThreads
```

The `-userThreads` option enables context lookup and class loading support from user-created threads.

Understanding JAX-RPC Handlers

WebServicesAssembler provides Ant tasks that let you configure JAX-RPC message handlers. Handlers cannot be configured by using the WebServicesAssembler command line.

Processing SOAP Headers

Strong Typing and the ServiceLifecycle Interface

Although the `ServiceLifecycle` interface enables you to access SOAP header blocks that may not have been declared in the WSDL file, the blocks are not strongly typed. You may also need to know the XML structure of a SOAP header in order to process it. For strong typing of SOAP header blocks, make sure that the `mapHeadersToParameters` argument for WebServicesAssembler is set to `true` (`true` is the default value). This is only possible if the SOAP header has been declared in the WSDL file and the type of the SOAP header is a supported JAX-RPC type.

Using WebServicesAssembler

Long file names cause deployment to fail

If the combined length of the generated file and directory names passes a certain size limit, then deployment will fail and throw an error. This size limit varies for different operating systems. For example, on the Windows operating system, the size limit is 255 characters

The length of the names is controlled by WebServicesAssembler and the deployment code. WebServicesAssembler generates file names based on the method name in the Java class or the operation name in the WSDL. The deployment code creates directories for code generation based on the names of the EAR and the WAR files.

To avoid the generation of file and directory names that are too long, limit the number of characters in the following names to a reasonable length.

- method names in Java classes
- operation names in the WSDL
- directory name for the location of the OC4J installation
- file name for a WAR file
- file name for a EAR file

You can also avoid this problem by upgrading to a more recent version of the J2SE 5.0 JDK (jdk-1_5_0_06 or later).

Getting More Information on WebServicesAssembler Errors

You can get detailed diagnostic information on errors returned by WebServicesAssembler by including the `debug` argument in the command line or Ant task. For more information on this argument, see the "debug" section of the "Using WebServicesAssembler" chapter in the *Oracle Application Server Web Services Developer's Guide*.

WebServicesAssembler Cannot Compile Files

If WebServicesAssembler cannot compile files successfully, it will return the error:

```
java? java.io.IOException: CreateProcess: javac -encoding UTF-8 -classpath
```

The `javac` compiler must be available so that WebServicesAssembler can compile your Java files. Make sure that *`JAVA_HOME`*`/bin` is in your path.

WebServicesAssembler Cannot Find Required Classes

You may need to use some classes that are common to all J2EE 1.4 applications. All standard J2EE 1.4 classes and Oracle database classes are included automatically. When using Ant tasks, WebServicesAssembler must search for the JARs that contain these classes.

A WebServicesAssembler Ant task tries to load these extra classes by searching for `wsa.jar`. The task searches for the following Ant properties or environment variables in the following order. If the task does not find `wsa.jar`, or if a property is not defined, it searches the next property.

1. `oc4j.home`—an Ant property that specifies the root installation directory for OC4J. This property can be used instead of an environment variable.
2. `OC4J_HOME`—an environment variable that specifies the root installation directory for OC4J.

3. `oracle.home`—an Ant property that specifies the root installation directory for Oracle products. This property can be used instead of an environment variable.
4. `ORACLE_HOME`—an environment variable that specifies the root installation directory for Oracle products.

When the Ant task finds `wsa.jar`, it loads all of the classes listed in its manifest file, relative to the location of `wsa.jar`.

Packaging and Deploying Web Services

Packaging for J2EE Clients

The current tool set cannot package J2EE Web service clients. You must package the client manually. "Packaging a J2EE Client" in the *Oracle Application Server Web Services Developer's Guide* provides more information on how to package a J2EE Web service client.

Getting the Correct Endpoint Address when the WSDL Has More than One HTTP Port

If a you want to enter values for the `<web-site>` or `<wsdl-publish-location>` elements in `oracle-webservices.xml`, then the returned WSDL may not have the correct endpoint addresses if the WSDL has more than one HTTP port.

The WebServicesAssembler tool does not insert the `<web-site>` or `<wsdl-publish-location>` elements into the `oracle-webservices.xml` file that it creates. You must insert these elements manually.

Ensuring Interoperable Web Services

Leading Underscores in WebServicesAssembler Generated Names

The default behavior of the WebServicesAssembler tool is to generate namespaces from the Java package name. If the Java package name begins with a leading underscore ("_"), then the generated namespace URI will contain an underscore. Some versions of the .NET WSDL tool may not be able to consume these namespaces, even though the generated namespace is valid.

To work around this .NET issue, use the WebServicesAssembler arguments `targetNamespace` and/or `mappingFileName` to avoid the default package-derived namespace.

Working with Message Attachments

Adding Faults with swaRef Attachments to a Web Service

In OracleAS Web Services, you can add only `swaRef` MIME type attachments to SOAP fault messages. It does not support the adding of `SWA` type attachments.

Faults with attachments can be added to a Web service only when you are assembling it from a WSDL (top down). They cannot be added when assembling a Web service bottom up.

Supported Message Formats for Attachments

Only RPC-literal and document-literal Web services are supported by the WS-I Attachments Profile 1.0. Thus, only those types of services can use `swaRef` MIME type.

WebServicesAssembler will not be able to assemble a Web service that can pass `swaRef` MIME attachments if you specify an RPC-encoded message format. To assemble the service, you must select a different format.

Managing Web Services

Limitations on Application Server Control

Application Server Control cannot modify everything that can be expressed in the `wsmgmt.xml` file. For example, it cannot be used to change parts of the reliability configuration.

Ensuring Web Service Reliability

Reliability Limitations for OracleAS Web Services

- The reliability process on the client lives only as long as the client process. If the client process becomes permanently unavailable, then any messages that needed to be retried will be ignored.
- Asynchronous polling capabilities can be enabled only at the port level and only by using a configuration.

Auditing and Logging Messages

Limitations on xpath Queries

An `xpath` query must return a primitive type. This means that the query must return the context of text nodes or attribute values.

The primitive type that the `xpath` query returns should have a small number of characters. For example, it should not exceed 120 characters.

Custom Serialization of Java Value Types

This section describes limitations on the custom serialization of non-standard data types.

Literal Use

This release supports only literal as the `use` part of the message format. This includes RPC-literal and document-literal. RPC-encoded is not supported in this release.

Object Graph

Because RPC-encoded is not supported in this release, the initial support of serialization and deserialization does not allow object graph marshalling using `href`. If a Java Object has multiple references either in the parameters of a request or in the return value of a response, then serialization and deserialization might not preserve the object graph.

WSDL- and Service-Level Configuration

`SoapElementSerializer` is configured for each Service or for each WSDL. For example, a `SoapElementSerializer` implementation representing the mapping of `dateTime` to `oracle.sql.DATE` can be configured to replace the default mapping of `dateTime` to `java.util.Calendar`. Under this configuration, every instance of the mapping is replaced. Configuration for each operation or message level is not supported in this release.

Sub Tree Serialization

Each custom serializer gets the full XML sub-tree, and performs the serialization and deserialization of the entire XML element object model. For example, assume you have two custom serializers developed and configured for two top-level `complexTypes`, `TypeA` and `TypeB`. `TypeA` has a sub-element of `TypeB`. Even though a custom serializer has been configured for `TypeB`, this custom serializer cannot be automatically invoked by the OC4J runtime when the sub-element of `TypeB` is serialized inside the custom serializer for `TypeA`. The custom serializer of `TypeA` must handle the sub-element of `TypeB` by itself. The custom serializer of `TypeA` can, in turn, call the custom serializer of `TypeB`, but it is up to the implementation strategy.

Document-Literal Wrapper

Assume that a a custom serializer is used to handle a global `complexType` that is referred by a global element to define the single part of a document-literal operation. If you use the `unwrapParameters` argument to unwrap the return type and response type, it will be ignored for the operation(s) that use the global element as the body part of the input message.

Using JMS as a Web Service Transport

Interoperability of Messages when using JMS as a Transport Mechanism

The WSDL extensions that enable JMS as a transport mechanism are Oracle-proprietary. The messages produced by the Web service may not be interoperable with applications or services provided by other vendors.

Retrieving Client Responses from JMS Web Service Transport

If the client process becomes unavailable without receiving its response and later returns, there is no facility provided for it to retrieve its old responses from the queue.

Using the Web Service Invocation Framework

This section describes limitations in the OracleAS Web Services support for the Web Services Invocation Framework (WSIF).

- Database WSIF can pass only the data source, not the JDBC connection, to the provider for database access.
- Database WSIF is stateless. Each operation obtains a JDBC connection when it begins and closes it when it ends. Autocommit is always on for the JDBC connection. It is recommended that you use connection pooling when setting up data sources to reduce database overhead.
- Oracle's Application Server Control Web Services Management and Monitoring can only directly monitor SOAP services; it cannot monitor any service interactions that utilize WSIF bindings, such as Java, EJB, or database WSIF bindings. By bypassing the SOAP protocol entirely, you are also bypassing the

management infrastructure for Web services provided by Oracle Application Server Control.

Using Dynamic Invocation Interface to Invoke Web Services

To invoke a Web service by using the Dynamic Invocation Interface (DII) requires a number of steps. At each step, you typically have to make some choices. The examples at the end of this section display choices made at each of the steps.

Using DII to invoke a Web Service consists of the following general steps:

1. Create the call object.
2. Register parameters.
3. Invoke the Web service.

You can create the call object either with or without a WSDL. If you do not have a WSDL, or decide not to use the WSDL for creating the call dynamically, then follow the steps under "Basic Calls". If you do have a WSDL to construct the call, then follow the instructions under "Configured Calls".

Basic Calls

For a basic call, the call object is created dynamically without a WSDL. The following steps provide more information on how to construct a basic call.

1. You are constructing a call object dynamically, without a WSDL. For examples, see:
   - Example E–1, "Basic Call with parameter registration and Java bindings"
   - Example E–4, "Basic Call with SOAPElement, but without parameter registration"
   - Example E–6, "Basic Call with document-literal invocation and SOAPElement, but without parameter registration"
2. Register parameters.
   - Case 1: You are constructing the SOAP request yourself as a `SOAPElement`, and are receiving the response as a `SOAPElement`. In this case, you do not have to register parameters or return types. For examples of this case, see:
     - Example E–4, "Basic Call with SOAPElement, but without parameter registration"
     - Example E–6, "Basic Call with document-literal invocation and SOAPElement, but without parameter registration"
   - Case 2: You are explicitly registering the parameters and returns (parts) that are being used in the Web service invocation in your Basic Call, including the part name, the XML and Java type names, and the parameter mode. In this case, you can furnish the individual parameters as Java object instances. For an example of this case, see:
     - Example E–1, "Basic Call with parameter registration and Java bindings"
3. Invoke the Web service. You can invoke the Web service either by using `SOAPElement` or by Java bindings.
   - Case 1: Using `SOAPElement`. If you are using a document-literal invocation, then you will typically construct a `SOAPElement` for your message and pass it

to the `invoke()` method. Note that this invocation employs the public, Oracle-specific API from `OracleCall`. For examples of this case, see:

  – Example E–4, "Basic Call with SOAPElement, but without parameter registration"
  – Example E–6, "Basic Call with document-literal invocation and SOAPElement, but without parameter registration"

- Case 2: Using Java bindings. If you are using an RPC-literal or RPC-encoded invocation, then you will typically supply an array containing Java objects in the `invoke()` method and cast the return object to the anticipated return type. For examples of this case, see:
  – Example E–1, "Basic Call with parameter registration and Java bindings"
  – Example E–2, "Configured Call with Java bindings, but without parameter registration"
  – Example E–3, "Configured Call with registration of wrapper parameters and Java bindings"
  – Example E–5, "Configured Call with a WSDL, complex return parameter registration, and Java bindings"
  – Example E–7, "Configured Call with RPC-encoded invocation, complex parameter registration, and Java bindings"

Configured Calls

For a configured call, the call object is constructed from a WSDL. The following steps provide more information on how to construct a configured call.

1. Provide the WSDL for constructing the call object. For examples, see:
   - Example E–2, "Configured Call with Java bindings, but without parameter registration"
   - Example E–3, "Configured Call with registration of wrapper parameters and Java bindings"
   - Example E–5, "Configured Call with a WSDL, complex return parameter registration, and Java bindings"
   - Example E–7, "Configured Call with RPC-encoded invocation, complex parameter registration, and Java bindings"
2. Register parameters. For a Configured Call, you must register parameters for the following cases:
   - Case 1: You are employing a complex or other type that is not being mapped to a primitive Java type (or an Object variant of a primitive Java type). For examples of this case, see:
     – Example E–5, "Configured Call with a WSDL, complex return parameter registration, and Java bindings"
     – Example E–7, "Configured Call with RPC-encoded invocation, complex parameter registration, and Java bindings".
   - Case 2: You are using a document-literal wrapped style and do not want to create a `SOAPElement` like the ones illustrated in Example E–4 and Example E–6 for Basic Call. In this case, the name of the parameter must be the `QName` of the wrapper. For an example of this case, see:

  - Example E–3, "Configured Call with registration of wrapper parameters and Java bindings"
- Case 3: If Case 1 and Case 2 do not apply, then you do not have to register parameters or returns. For an example of this case, see:
  - Example E–2, "Configured Call with Java bindings, but without parameter registration"

3. Invoke the Web service. You can invoke the Web service either by using `SOAPElement` or by Java bindings.
   - Case 1: Using `SOAPElement`. If you are using a document-literal invocation, then you will typically construct a `SOAPElement` for your message and pass it to the `invoke()` method. Note that this invocation employs the public, Oracle-specific API from `OracleCall`. For examples of this case, see:
     - Example E–4, "Basic Call with SOAPElement, but without parameter registration"
     - Example E–6, "Basic Call with document-literal invocation and SOAPElement, but without parameter registration"
   - Case 2: Using Java bindings. If you are using an RPC-literal or RPC-encoded invocation, then you will typically supply an array containing Java objects in the `invoke()` method and cast the return object to the anticipated return type. For examples of this case, see:
     - Example E–1, "Basic Call with parameter registration and Java bindings"
     - Example E–2, "Configured Call with Java bindings, but without parameter registration"
     - Example E–3, "Configured Call with registration of wrapper parameters and Java bindings"
     - Example E–5, "Configured Call with a WSDL, complex return parameter registration, and Java bindings"
     - Example E–7, "Configured Call with RPC-encoded invocation, complex parameter registration, and Java bindings"

Examples of Web Service Clients that use DII

This section provides a variety of client examples that use basic calls or configured calls to invoke a Web service.

The following code snippet illustrates an `import` statement that can be used by the following code examples.

```
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.Service;
import javax.xml.rpc.Call;
import javax.xml.rpc.ParameterMode;
import javax.xml.namespace.QName;
import javax.xml.soap.SOAPFactory;
import javax.xml.soap.SOAPElement;
import java.net.URL;
import oracle.webservices.OracleCall;
import oracle.xml.parser.v2.XMLElement;
```

Example E–1 Basic Call with parameter registration and Java bindings

```
// (1) Creation of call object without WSDL.
String endpoint = "http://localhost:8888/echo/DiiDocEchoService";
ServiceFactory sf = ServiceFactory.newInstance();
Service service = sf.createService(new QName("http://echo.demo.oracle/", "tns"));
Call call = service.createCall();

// (2) Configuration of call and registration of parameters.
call.setTargetEndpointAddress(endpoint);
call.setProperty(Call.SOAPACTION_USE_PROPERTY, new Boolean(true));
call.setProperty(Call.SOAPACTION_URI_PROPERTY, "");
call.setProperty(Call.ENCODINGSTYLE_URI_PROPERTY,
"http://schemas.xmlsoap.org/soap/encoding/");
call.setProperty(Call.OPERATION_STYLE_PROPERTY, "rpc");
QName QNAME_TYPE_STRING = new QName("http://www.w3.org/2001/XMLSchema", "string");
call.addParameter("s", QNAME_TYPE_STRING, ParameterMode.IN);
call.setReturnType(QNAME_TYPE_STRING);

// (3) Invocation.
System.out.println("Response is " + call.invoke(new Object[]{"hello"}));
```

Example E–2 Configured Call with Java bindings, but without parameter registration

```
/// (1) Creation of call object using WSDL.
String namespace = "http://www.xmethods.net/sd/CurrencyExchangeService.wsdl";
URL wsdl=new URL(namespace);
ServiceFactory factory = ServiceFactory.newInstance();
QName serviceName = new QName(namespace, "CurrencyExchangeService");
Service service = factory.createService(wsdl, serviceName);
QName portName = new QName(namespace, "CurrencyExchangePort");
Call call = service.createCall(portName);

// (2) Registration of parameters.
//     -> taken from the WSDL

// (3) Configuration of operation and invocation.
QName operationName = new QName("urn:xmethods-CurrencyExchange", "getRate");
call.setOperationName(operationName);
Float rate = (Float) call.invoke(new Object[]{"usa", "canada"});
System.out.println("getRate: " + rate);
```

Example E–3 Configured Call with registration of wrapper parameters and Java bindings

```
// (1) Creation of call object using WSDL.
String namespace = "http://server.hello/jaxws";
ServiceFactory factory = ServiceFactory.newInstance();
QName serviceName = new QName(namespace, "HelloImplService");
URL wsdl=new URL(namespace+"?WSDL");
Service service = factory.createService(wsdl, serviceName);
QName portName = new QName(namespace, "HelloImpl");
Call call = service.createCall(portName);

// (2) Registration of SayHello and SayHelloResponse wrapper classes
//     These must be available in the classpath.
String TYPE_NAMESPACE_VALUE = "http://server.hello/jaxws";
QName reqQname = new QName(TYPE_NAMESPACE_VALUE,"sayHelloElement");
```

```
QName respQName = new QName(TYPE_NAMESPACE_VALUE,"sayHelloResponseElement");
call.addParameter("name", reqQname, SayHello.class, ParameterMode.IN );
call.setReturnType(respQName, SayHelloResponse.class );

// (3) Invocation
SayHello input = new SayHello("Duke");
Object[] params = new Object[] { input };
SayHelloResponse result = (SayHelloResponse) call.invoke( params );
String response = result.getResult();
```

Example E–4 Basic Call with SOAPElement, but without parameter registration

```
// (1) Creation of call object without WSDL
ServiceFactory sf = ServiceFactory.newInstance();
Service service = sf.createService(new QName("http://echo.demo.oracle/", "tns"));
Call call = service.createCall();
call.setTargetEndpointAddress("http://localhost:8888/echo/DiiDocEchoService");

// (2) No registration of parameters

// (3a) Direct creation of payload as SOAPElement
SOAPFactory soapfactory = SOAPFactory.newInstance();
SOAPElement m1 = soapfactory.createElement("echoStringElement", "tns",
"http://echo.demo.oracle/");
SOAPElement m2 = soapfactory.createElement("s", "tns",
"http://echo.demo.oracle/");
m2.addTextNode("Bob");
m1.addChildElement(m2);
System.out.println("Request is: ");
((XMLElement) m1).print(System.out);

// (3b) Invocation
SOAPElement resp = (SOAPElement) ((OracleCall) call).invoke(m1);
System.out.println("Response is: ");
((XMLElement) resp).print(System.out);
```

Example E–5 Configured Call with a WSDL, complex return parameter registration, and Java bindings

```
// (0) Preparing a complex argument value
Integer  req_I = new Integer( Integer.MAX_VALUE );
String   req_s = "testDocLitBindingAnonymAll & <body>";
Integer  req_inner_I = new Integer( Integer.MIN_VALUE );
String   req_inner_s = "<inner> & <body>";
int[]    req_inner_i = {0,Integer.MAX_VALUE, Integer.MIN_VALUE};
InnerSequence req_inner = new InnerSequence();
req_inner.setVarInteger( req_inner_I );
req_inner.setVarString ( req_inner_s );
req_inner.setVarInt    ( req_inner_i );
EchoAnonymAllElement req = new EchoAnonymAllElement();
req.setVarInteger   ( req_I );
req.setVarString    ( req_s );
req.setInnerSequence( req_inner);

// (1) Creation of call object using the WSDL
```

```
String TARGET_NS = "http://soapinterop.org/DocLitBinding";
String TYPE_NS = "http://soapinterop.org/xsd";
String XSD_NS = "http://www.w3.org/2001/XMLSchema";
QName SERVICE_NAME = new QName( TARGET_NS, "DocLitBindingService" );
QName PORT_NAME = new QName( TARGET_NS, "DocLitBindingPort");
String wsdlUrl = "http://"+getProperty("HOST")+
                 ":"+getProperty("HTTP_PORT")+
                 "/doclit_binding/doclit_binding";

QName operation = new QName(TARGET_NS, "echoAnonymAll");
ServiceFactory factory = ServiceFactory.newInstance();
Service srv = factory.createService( new URL(wsdlUrl + "?WSDL"), SERVICE_NAME);
Call call = srv.createCall( PORT_NAME, operation );

// (2) Registration of complex return parameter
call.setReturnType(new QName(TYPE_NS, "EchoAnonymAllElement"),
EchoAnonymAllElement.class);

// (3) Invocation
EchoAnonymAllElement res = (EchoAnonymAllElement) call.invoke( new Object[]{req}
);
System.out.println( "AnonymAll body  : " +res.getVarString() );
System.out.println( "AnonymAll inner : " +res.getInnerSequence() );
```

Example E–6 Basic Call with document-literal invocation and SOAPElement, but without parameter registration

```
// (1) Creation of Basic Call
ServiceFactory sf = ServiceFactory.newInstance();
Service service = sf.createService(new QName("http://echo.demo.oracle/", "tns"));
String endpoint="http://localhost:8888/test/echo";
Call call = service.createCall();
call.setTargetEndpointAddress(endpoint);

// (2) No parameter registration

// (3) Invocation using SOAPElement
SOAPFactory soapfactory = SOAPFactory.newInstance();
SOAPElement m1 = soapfactory.createElement("echoStringElement", "tns",
"http://echo.demo.oracle/");
SOAPElement m2 = soapfactory.createElement("s", "tns",
"http://echo.demo.oracle/");
m2.addTextNode("Bob");
m1.addChildElement(m2);
System.out.println("Request is: ");
((XMLElement) m1).print(System.out);
SOAPElement resp = (SOAPElement) ((OracleCall) call).invoke(m1);
System.out.println("Response is: ");
((XMLElement) resp).print(System.out);
```

Example E–7 Configured Call with RPC-encoded invocation, complex parameter registration, and Java bindings

```
// (1) Creation of ConfiguredCall using WSDL
ServiceFactory sf = ServiceFactory.newInstance();
String endpoint="http://localhost:8888/test/echo";
```

```
Service service = sf.createService(new java.net.URL(endpoint + "?WSDL"), new
QName("http://echo.demo.oracle/", "DiiRpcEchoService"));
Call call = service.createCall(new QName("http://echo.demo.oracle/",
"HttpSoap11"), new QName("http://echo.demo.oracle/", "echoStrings"));
call.setProperty(Call.SOAPACTION_USE_PROPERTY, new Boolean(true));
call.setProperty(Call.SOAPACTION_URI_PROPERTY, "");
call.setProperty(Call.ENCODINGSTYLE_URI_PROPERTY,
"http://schemas.xmlsoap.org/soap/encoding/");
call.setProperty(Call.OPERATION_STYLE_PROPERTY, "rpc");

// (2) Registration of complex input and return arguments
QName stringArray = new QName("http://echo.demo.oracle/", "stringArray");
call.addParameter("s", stringArray, String[].class, ParameterMode.IN);
call.setReturnType(stringArray, String[].class);

// (3) Invocation
String[] request = new String[]{"bugs", "little pieces", "candy"};
String[] resp = (String[]) call.invoke(new Object[]{request});
System.out.println("Response is: ");
for (int i = 0; i < resp.length; i++) {
   System.out.print(resp[i] + " ");
}
System.out.println();
```

F

Third Party Licenses

This appendix includes the Third Party License for all the third party products included with Oracle Application Server Web Services Security.

Apache

This program contains third-party code from the Apache Software Foundation ("Apache"). Under the terms of the Apache license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Apache software, and the terms contained in the following notices do not change those rights.

The Apache license agreements apply to the following included Apache components:

- Apache HTTP Server
- Apache JServ
- mod_jserv
- Regular Expression package version 1.3
- Apache Expression Language packaged in commons-el.jar
- mod_mm 1.1.3
- Apache XML Signature and Apache XML Encryption v. 1.4 for Java and 1.0 for C++
- log4j 1.1.1
- BCEL v. 5
- XML-RPC v. 1.1
- Batik v. 1.5.1
- ANT 1.6.2 and 1.6.5
- Crimson v. 1.1.3
- ant.jar
- wsif.jar
- bcel.jar
- soap.jar
- Jakarta CLI 1.0
- jakarta-regexp-1.3.jar

- JSP Standard Tag Library 1.0.6 and 1.1
- Struts 1.1
- Velocity 1.3
- svnClientAdapter
- commons-logging.jar
- wsif.jar
- commons-el.jar
- standard.jar
- jstl.jar

The Apache Software License

License for Apache Web Server 1.3.29


```
/* ====================================================================
 * The Apache Software License, Version 1.1
 *
 * Copyright (c) 2000-2002 The Apache Software Foundation.  All rights
 * reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 *
 * 3. The end-user documentation included with the redistribution,
 *    if any, must include the following acknowledgment:
 *       "This product includes software developed by the
 *        Apache Software Foundation (http://www.apache.org/)."
 *    Alternately, this acknowledgment may appear in the software itself,
 *    if and wherever such third-party acknowledgments normally appear.
 *
 * 4. The names "Apache" and "Apache Software Foundation" must
 *    not be used to endorse or promote products derived from this
 *    software without prior written permission. For written
 *    permission, please contact apache@apache.org.
 *
 * 5. Products derived from this software may not be called "Apache",
 *    nor may "Apache" appear in their name, without prior written
 *    permission of the Apache Software Foundation.
 *
 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED.  IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR
 * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
```

```
 * USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
 * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 * ====================================================================
 *
 * This software consists of voluntary contributions made by many
 * individuals on behalf of the Apache Software Foundation.  For more
 * information on the Apache Software Foundation, please see
 * <http://www.apache.org/>.
 *
 * Portions of this software are based upon public domain software
 * originally written at the National Center for Supercomputing
Applications,
 * University of Illinois, Urbana-Champaign.
```


License for Apache Web Server 2.0


```
Copyright (c) 1999-2004, The Apache Software Foundation
Licensed under the Apache License, Version 2.0 (the "License"); you may not use
this file except in compliance with the License.  You may obtain a copy of the
License at http://www.apache.org/licenses/LICENSE-2.0
Unless required by applicable law or agreed to in writing, software distributed
under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
CONDITIONS OF ANY KIND, either express or implied.  See the License for the
specific language governing permissions and limitations under the License.
Copyright (c) 1999-2004, The Apache Software Foundation
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1. Definitions.

      "License" shall mean the terms and conditions for use, reproduction,
      and distribution as defined by Sections 1 through 9 of this document.

      "Licensor" shall mean the copyright owner or entity authorized by
      the copyright owner that is granting the License.

      "Legal Entity" shall mean the union of the acting entity and all
      other entities that control, are controlled by, or are under common
      control with that entity. For the purposes of this definition,
      "control" means (i) the power, direct or indirect, to cause the
      direction or management of such entity, whether by contract or
      otherwise, or (ii) ownership of fifty percent (50%) or more of the
      outstanding shares, or (iii) beneficial ownership of such entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making modifications,
      including but not limited to software source code, documentation
      source, and configuration files.

      "Object" form shall mean any form resulting from mechanical
      transformation or translation of a Source form, including but
      not limited to compiled object code, generated documentation,
```

and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or
Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices
stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works
that You distribute, all copyright, patent, trademark, and
attribution notices from the Source form of the Work,
excluding those notices that do not pertain to any part of
the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its
distribution, then any Derivative Works that You distribute must
include a readable copy of the attribution notices contained
within such NOTICE file, excluding those notices that do not
pertain to any part of the Derivative Works, in at least one
of the following places: within a NOTICE text file distributed
as part of the Derivative Works; within the Source form or
documentation, if provided along with the Derivative Works; or,
within a display generated by the Derivative Works, if and
wherever such third-party notices normally appear. The contents
of the NOTICE file are for informational purposes only and
do not modify the License. You may add Your own attribution
notices within Derivative Works that You distribute, alongside
or as an addendum to the NOTICE text from the Work, provided
that such additional attribution notices cannot be construed
as modifying the License.

You may add Your own copyright statement to Your modifications and
may provide additional or different license terms and conditions
for use, reproduction, or distribution of Your modifications, or
for any such Derivative Works as a whole, provided Your use,
reproduction, and distribution of the Work otherwise complies with
the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
any Contribution intentionally submitted for inclusion in the Work
by You to the Licensor shall be under the terms and conditions of
this License, without any additional terms or conditions.
Notwithstanding the above, nothing herein shall supersede or modify
the terms of any separate license agreement you may have executed
with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
names, trademarks, service marks, or product names of the Licensor,
except as required for reasonable and customary use in describing the
origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
agreed to in writing, Licensor provides the Work (and each
Contributor provides its Contributions) on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied, including, without limitation, any warranties or conditions
of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
PARTICULAR PURPOSE. You are solely responsible for determining the
appropriateness of using or redistributing the Work and assume any
risks associated with Your exercise of permissions under this License.

```
8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.
```

Apache SOAP

This program contains third-party code from the Apache Software Foundation ("Apache"). Under the terms of the Apache license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Apache software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the Apache software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or Apache.

Apache SOAP License

Apache SOAP license 2.3.1

```
Copyright (c) 1999 The Apache Software Foundation.  All rights reserved.
TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION
1. Definitions.

      "License" shall mean the terms and conditions for use, reproduction,
      and distribution as defined by Sections 1 through 9 of this document.

      "Licensor" shall mean the copyright owner or entity authorized by
      the copyright owner that is granting the License.

      "Legal Entity" shall mean the union of the acting entity and all
      other entities that control, are controlled by, or are under common
      control with that entity. For the purposes of this definition,
      "control" means (i) the power, direct or indirect, to cause the
      direction or management of such entity, whether by contract or
      otherwise, or (ii) ownership of fifty percent (50%) or more of the
      outstanding shares, or (iii) beneficial ownership of such entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.
```

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses

granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS,

```
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

JSR 110

This program contains third-party code from IBM Corporation ("IBM"). Under the terms of the IBM license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the IBM software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the IBM software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or IBM.

```
Copyright IBM Corporation 2003 - All rights reserved

Java APIs for the WSDL specification are available at:
http://www-124.ibm.com/developerworks/projects/wsdl4j/
```

Jaxen

This program contains third-party code from the Apache Software Foundation ("Apache") and from the Jaxen Project ("Jaxen"). Under the terms of the Apache and Jaxen licenses, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Apache and Jaxen software, and the terms contained in the following notices do not change those rights.

The Jaxen License

Copyright (C) 2000-2002 bob mcwhirter & James Strachan. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution.

The name "Jaxen" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact license@jaxen.org.

Products derived from this software may not be called "Jaxen", nor may "Jaxen" appear in their name, without prior written permission from the Jaxen Project Management (pm@jaxen.org).

In addition, we request (but do not require) that you include in the end-user documentation provided with the redistribution and/or in the software itself an acknowledgment equivalent to the following: "This product includes software developed by the Jaxen Project (http://www.jaxen.org/)." Alternatively, the acknowledgment may be graphical using the logos available at http://www.jaxen.org/.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE Jaxen AUTHORS OR THE PROJECT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This software consists of voluntary contributions made by many individuals on behalf of the Jaxen Project and was originally created by bob mcwhirter and James Strachan . For more information on the Jaxen Project, please see http://www.jaxen.org/.

SAXPath

This program contains third-party code from SAXPath. Under the terms of the SAXPath license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the SAXPath software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the SAXPath software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or SAXPath.

The SAXPath License

Copyright (C) 2000-2002 werken digital. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution.

The name "SAXPath" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact license@saxpath.org.

Products derived from this software may not be called "SAXPath", nor may "SAXPath" appear in their name, without prior written permission from the SAXPath Project Management (pm@saxpath.org).

In addition, we request (but do not require) that you include in the end-user documentation provided with the redistribution and/or in the software itself an acknowledgment equivalent to the following: "This product includes software developed by the SAXPath Project (http://www.saxpath.org/)." Alternatively, the acknowledgment may be graphical using the logos available at http://www.saxpath.org/.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE SAXPath AUTHORS OR THE PROJECT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. This software consists of voluntary contributions made by many individuals on behalf of the SAXPath Project and was originally created by bob mcwhirter and James Strachan . For more information on the SAXPath Project, please see http://www.saxpath.org/.

W3C DOM

This program contains third-party code from the World Wide Web Consortium ("W3C"). Under the terms of the W3C license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the W3C software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the W3C software is provided by Oracle AS IS and without warranty or support of any kind from Oracle or W3C.

The W3C License

W3C® SOFTWARE NOTICE AND LICENSE
http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231
This work (and included software, documentation such as READMEs, or other related items) is being provided by the copyright holders under the following license. By obtaining, using and/or copying this work, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions.

Permission to copy, modify, and distribute this software and its documentation, with or without modification, for any purpose and without fee or royalty is hereby

granted, provided that you include the following on ALL copies of the software and documentation or portions thereof, including modifications:

The full text of this NOTICE in a location viewable to users of the redistributed or derivative work.
Any pre-existing intellectual property disclaimers, notices, or terms and conditions. If none exist, the W3C Software Short Notice should be included (hypertext is preferred, text is permitted) within the body of any redistributed or derivative code.
Notice of any changes or modifications to the files, including the date changes were made. (We recommend you provide URIs to the location from which the code is derived.)
THIS SOFTWARE AND DOCUMENTATION IS PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE OR DOCUMENTATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

COPYRIGHT HOLDERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE SOFTWARE OR DOCUMENTATION.

The name and trademarks of copyright holders may NOT be used in advertising or publicity pertaining to the software without specific, written prior permission. Title to copyright in this software and any associated documentation will at all times remain with copyright holders.

Index

Symbols

<auditing> element, 6-3, 6-4
<call-property> element, 3-10, 9-23
<dime:message> WSDL extension, 2-22
<expose-testpage> element, 10-7
<expose-wsdl> element, 10-7
<fault> element, logging, 6-8
<implementation-class> element, 10-7
<load-on-startup> element, C-4
<logging> element, A-2
<max-request-size> element, 10-7
<name> element, 9-23
<namespace> element, logging, 6-7
<namespaceURI> element, B-3
<operation> element, 6-4, 10-8, A-3
<operations> element, 3-5, 9-23, 10-8, A-3
<package-mapping> element, B-3
<package-type> element, B-3, B-4
<policy> element, 10-8
<port> element, 3-5, 9-4, A-2, A-3
<port-component> element, 3-5
<port-info> element, 3-13, 6-4
<property> element, 10-7, 10-8
<provider-description> element, 10-6, 10-7
<provider-description-name> element, 10-7
<provider-name> element, 10-7
<provider-port> element, 10-7
<reliability> element, A-2
<request> element, logging, 6-8
<response> element, logging, 6-8
<runtime> element, 3-5, 9-23, 10-8, A-1
<security> element, A-1
<service-endpoint-interface> element, 9-23
<service-endpoint-method-mapping> element, B-4
<service-ref> element, 3-13
<service-ref-mapping> element, 3-8, 3-13, 6-4
<service-ref-name> element, 3-13
<servlet> element, 10-5, 10-8
<servlet-class> element, 10-5, 10-8, 10-9
<servlet-link> element, 10-7
<servlet-mapping> element, 10-8
<soap-header> element, B-5
<stream-attachments> WSDL extension, 2-19
<stub-property> element, 3-11, 9-23
<url-pattern> element, 10-5, 10-8, 10-9
<value> element, 9-23
<wrapped-element> element, B-4
<wsdl-file> element, 10-7
<wsdl-message-mapping> element, B-5
<wsdl-port> element, 9-22, 10-7
<wsdl-service-name> element, 10-7

A

annotations
  J2SE 5.0 Web Services Annotations
    limitations, E-6
Apache software
  license, F-2
APIs
  Provider API, 10-1
  streaming attachment, 2-16
  WSIF, 9-17
app, attribute for <port> element, A-3
application clients, static
  configuring management information, 3-15
Application Server Control
  configuring server side management
    information, 3-5
  support for auditing, 6-14
  support for logging, 6-14
  support for Web service management, 3-16
  support for Web Service Providers, 10-11
aqAssemble command, 9-12, 9-14, 9-15
assemble command, 9-3, 9-4, 9-5
Attachment interface, 2-21
AttachmentFactory class, 2-21
AttachmentFault interface, 2-13
AttachmentPart API
  creating a new instance, 2-6
attachments
  adding to WSIF clients, 9-23
    with OracleCall API, 9-24
Attachments API, 2-19
Attachments interface, 2-20
audit file location, 6-3
audit messages
  processing, 6-2
  sample, 6-2
auditing
  adding to a J2EE client, 6-12

adding to a J2SE client proxy, 6-11
adding to a Web service bottom up, 6-9
adding to a Web service top down, 6-11
Application Server Control support, 6-14
audit file location, 6-3
client-side management, 6-4
defined, 6-1
fault messages, 6-3
JDeveloper support, 6-14
operation-level client-side elements, 6-4
operation-level server-side elements, 6-3
performance impacts, 6-1
request messages, 6-2
response messages, 6-2
server-side management, 6-3
WebServicesAssembler support, 6-9

B

bottom up Web service assembly
streaming attachments, 2-16
swaRef MIME attachments, 2-7

C

capability assertions, 3-18
assembling into a Web service, 3-19
classLoader
attribute for wsifJavaPort argument, 9-4
className
attribute for wsifDbBinding argument, 9-12
attribute for wsifEjbPort argument, 9-8
attribute for wsifJavaPort argument, 9-4
classPath
attribute for wsifJavaPort argument, 9-4
client
SOAP faults with swaRef attachments, 2-14
clients
thread usage, E-7
compliance testing tool for interoperability, 1-4
correlation-ID
JMS message property, E-5

D

data binding extensions and interoperability issues, 1-7
data flow
in a J2EE client, 3-8
in a J2SE client, 3-6
data type design and interoperability, 1-3
data types and interoperability, 1-3
database
accessing with a WSIF client, 9-20
database resources
limitations on exposing as a Web service, E-5
WSDL extensions for WSIF, 9-16
dataSource
attribute for wsifDbBinding argument, 9-12
dbConnection
attribute for wsifDbBinding argument, 9-12
dbJavaAssemble command, 9-12, 9-14, 9-15
dbUser
attribute for wsifDbBinding argument, 9-12
deployment
Provider Web application, 10-11
testing a Provider Web application deployment, 10-11
DII client
configuring management information dynamically, 3-10
DIME encoded attachments, 2-21
interoperable, 2-21
Oracle-proprietary, 2-23
WSDL extensions, 2-22
document-literal operations
wrapping and unwrapping the mapping, B-4

E

EJB clients, static
configuring management information, 3-14
EJB port
configuring a WSIF endpoint, 9-7, 9-10
ejbAssemble command, 9-7, 9-8, 9-10
EJBs
WSDL extensions for WSIF, 9-10

F

fault attribute, auditing, 6-3, 6-4
fault messages
auditing, 6-3
logging, 6-6

G

genInterface command, 9-20
genJmsPropertyHeader argument, E-5
genWsdl command, 9-3, 9-4, 9-5, 9-7, 9-8, 9-10, 9-12, 9-14, 9-15
global level Web service management, A-1

H

header processing
limitations, E-7

I

IncomingAttachments Interface, 2-20
initialContextFactory
attribute for wsifEjbPort argument, 9-8
INOUT parameter
limitations on SQL types, E-5
interoperability
compliance testing tool, 1-4
data type design, 1-3
data types, 1-3
defined, 1-1
diagnosing issues, 1-4
DIME encoded attachments, 2-21

public organizations, 1-2
RPC-encoded message format, 1-4
using null values, 1-4
using platform native types, 1-4
Web service design, 1-3
interoperability issues
diagnosing and solving, 1-4
illegal XML characters, 1-9
invalid WSDLs, 1-6
proprietary data binding extensions, 1-7
SOAPAction values, 1-10
using null values, 1-13
interoperable Web Services, guidelines, 1-3

J

J2EE client
adding auditing, 6-12
configuring management information
with JDeveloper, 3-9
with WebServicesAssembler, 3-9
configuring management information dynamically, 3-12
configuring managment information, 3-8
management data flow, 3-8
J2EE clients
for Provider Web applications, 10-11
J2EE components, Oracle-proprietary deployment descriptors, and standard J2EE deployment descriptors, 3-13
J2SE client
adding auditing, 6-11
configuring management information
with JDeveloper, 3-7
with WebServicesAssembler, 3-7
configuring managment information, 3-7
management data flow, 3-6
J2SE clients
for Provider Web applications, 10-11
Java classes
configuring a WSIF endpoint, 9-3
WSDL extensions for WSIF, 9-6
Java mappings
changing via the JAX-RPC mapping file, B-3
Java method parameters, mapping to SOAP headers, B-5
Java port
configuring a WSIF endpoint, 9-3, 9-5
JAX-RPC mapping file, B-1
creating, B-1
document-literal operations
wrapping and unwrapping mapping, B-4
mapping SOAP headers and Java method parameters, B-5
naming conventions, B-2
using to change namespace to Java mappings, B-3
using to change the names of Java artifacts, B-3
using to change the names of WSDL artifacts, B-3
using to customize the WSDL, B-2
using to generate code into a single package, B-3
JDeveloper
configuring J2EE client management information, 3-9
configuring J2SE client management information, 3-7
configuring server side management information, 3-4
support for auditing, 6-14
support for logging, 6-14
JDeveloper support for WSIF, 9-24
JMS endpoint Web service
limitations, E-5
JMS message property
correlation-ID, E-5
message-ID, E-5
reply-to-destination, E-5
jndiName
attribute for wsifEjbPort argument, 9-8
jndiProviderURL
attribute for wsifEjbPort argument, 9-8
JSP clients, static
configuring management information, 3-14

L

levels of Web service management, A-1
life cycle for Web service management, 3-4
limitations
for Web service management, E-10
for WSIF, E-11
packaging, E-9
log file location, 6-6
log record, sample, 6-5
logging
adding to a Web service bottom up, 6-9
adding to a Web service top down, 6-11
Application Server Control support, 6-14
defined, 6-5
fault messages, 6-6
JDeveloper support, 6-14
log file location, 6-6
operation-level server-side elements, 6-7
performance impacts, 6-5
port-level server-side elements, 6-7
request messages, 6-6
response messages, 6-6
logging messages, processing, 6-5

M

management data flow for Web services, 3-2
management environment for Web services, 3-2
mapping
wrapping and unwrapping, B-4
mapping file, JAX-RPC, B-1
MBeans
described, C-1
initializing, C-4
naming components, C-2

WebServiceOperation, C-2
WebServicePort, C-2
WSMHandlerGlobalConfig, C-3
WSMHandlerOperationConfig, C-3, C-4
WSMHandlerServiceConfig, C-3
WSMOperationConfig, C-3
WSMServiceConfig, C-3
message attachments
DIME encoded attachments, 2-21
MIME attachments, 2-1
streaming attachments, 2-15
SWA attachments, 2-8
swaRef MIME attachments, 2-2
message flow for Web service management, 3-2
message-ID
JMS message property, E-5
MIME attachments, 2-1
adding SOAP faults, 2-11

N

name
attribute for wsifDbPort argument, 9-12
attribute for wsifEjbPort argument, 9-9
attribute for wsifJavaPort argument, 9-4
namespace mappings
changing via the JAX-RPC mapping file, B-3
null values and interoperability, 1-13
null values, effects on interoperability, 1-4

O

operation level Web service management, A-3
oracle, 10-4
Oracle HTTP Server
third party licenses, F-1
OracleCall API, 9-24
oracle.j2ee.ws.server.provider.ProviderServlet class, 10-4
extending, 10-5
Oracle-proprietary deployment descriptors, standard J2EE deployment descriptors, and J2EE components, 3-13
oracle-webservices-10_0.xsd schema, 10-4
oracle.webservices.attachments package, 2-16, 2-19
oracle-webservices-client-10_0.xsd schema, 3-6, 9-21
oracle.webservices.OracleCall API, 9-24
oracle.webservices.provider package, 10-1
oracle-webservices.xml
provider elements, 10-7
OUT parameter
limitations on SQL types, E-5
OutgoingAttachments interface, 2-20

P

packaging
limitations, E-9
Provider Web Application, 10-10
platform native types and interoperability, 1-4
plsqlAssemble command, 9-12, 9-14, 9-15
port level Web service management, A-2
port, attribute for <port> element, A-2
Provider
defined, 10-1
Provider API, 10-1
dynamically adding a managable service endpoint, 10-3
dynamically removing a managable service endpoint, 10-3
HTTPConstants class, 10-4
initializing a provider, 10-2
MessageContext class, 10-4
Provider interface, 10-2
ProviderConfig class, 10-3
removing a Provider instance, 10-3
provider package, 10-1
ProviderServlet class, 10-4
extending, 10-5
proxy Web service client
configuring management information dynamically, 3-11

R

REF CURSOR parameter, E-5
reply-to-destination
JMS message property, E-5
request attribute, auditing, 6-3, 6-4
request messages
auditing, 6-2
logging, 6-6
response attribute, auditing, 6-3, 6-4
response messages
auditing, 6-2
logging, 6-6
RPC-encoded message format and interoperability, 1-4

S

sample audit message, 6-2
sample log record, 6-5
security, Web services, 4-1
service endpoint interface
customizing, B-2
handling swaRef MIME attachments, 2-7
implementing with swaRef MIME attachments, 2-5
service, attribute for <port> element, A-3
Servlet clients, static
configuring management information, 3-14
SOAP faults
adding to MIME attachments, 2-11
adding to swaRef MIME attachments in the WSDL, 2-13
with swaRef attachments, 2-13
with swaRef attachments on the client, 2-14
SOAP headers
mapping to Java method parameters, B-5
soapAction attribute

controlling the value in ŽNET, 1-12
controlling the value in OC4J, 1-11
described, 1-11
SOAPAction values and interoperability issues, 1-10
SOAPBuilders community, 1-2
sqlAssemble command, 9-12, 9-14, 9-15
standard J2EE deployment descriptors, Oracle-proprietary deployment descriptors, and J2EE components, 3-13
static clients
configuring management information, 3-12
static proxy Web service client
configuring management information dynamically, 3-11
streaming attachments, 2-15
APIs, 2-16
bottom up Web service assembly, 2-16
limitations, 2-16
size recommendations, 2-15
supported format, 2-15
top down Web service assembly, 2-19
writing stub code, 2-18
WSDL extensions, 2-19
stub code
for streaming attachments, 2-18
SWA attachments, 2-8
top down Web service assembly, 2-9
swaRef attachments
throwing faults with, 2-13
swaRef MIME attachments, 2-2
adding SOAP faults in the WSDL, 2-13
adding to a WSDL, 2-4
assembling a WSDL, 2-8
bottom up Web service assembly, 2-7
implementing a service endpoint interface, 2-5
top down Web service assembly, 2-2
writing service endpoint interfaces for, 2-7

T

third party licenses, F-1
tool support
Web service management with Application Server Control, 3-16
Web Service Providers, 10-11
tool support for WSIF, 9-24
top down Web service assembly
limitations, E-4
streaming attachments, 2-19
SWA attachments, 2-9
swaRef MIME attachments, 2-2

U

unwrapParameters argument, E-11
useDimeEncoding argument, 2-23
userThreads option (Oracle Application Server), E-7

W

Web service design and interoperability, 1-3
Web Service Home Page
limitations, E-7
Web Service Interoperability Organization, 1-2
Web service management
configuring for a J2EE client, 3-8
with JDeveloper, 3-9
with WebServicesAssembler, 3-9
configuring for a J2SE client, 3-7
with JDeveloper, 3-7
with WebServicesAssembler, 3-7
configuring for the server side, 3-4
by hand, 3-4
with Application Server Control, 3-5
with JDeveloper, 3-4
with WebServicesAssembler, 3-5
configuring static application clients, 3-15
configuring static clients, 3-12
configuring static EJB clients, 3-14
configuring static JSP clients, 3-14
configuring static Servlet clients, 3-14
configuring WSIF clients, 9-21
data flow, 3-2
data flow in a J2EE client, 3-8
data flow in a J2SE client, 3-6
defiend, 3-1
DII client dynamic configuration, 3-10
dynamic client side configuration, 3-9
global level, A-1
J2EE client dynamic configuration, 3-12
levels, A-1
life cycle, 3-4
limitations, E-10
management environment, 3-2
message flow, 3-2
operation level, A-3
port level, A-2
proxy Web service client dynamic configuration, 3-11
static proxy Web service client dynamic configuration, 3-11
Web Service Provider
assembling clients, 10-11
deploying a Provider Web application, 10-11
managing Provider endpoints, 10-11
packaging, 10-10
registering a dynamic endpoint, 10-9
registering a static endpoint, 10-9
testing a Provider Web application deployment, 10-11
tool support, 10-11
Web Service Providers
dynamically adding a managable service endpoint, 10-3
dynamically removing a managable service endpoint, 10-3
elements in oracle-webservices.xml, 10-7
extending the ProviderServlet class, 10-5
HTTPConstants class, 10-4
initializing a provider, 10-2
MessageContext class, 10-4

Provider interface, 10-2
ProviderConfig class, 10-3
ProviderServlet class, 10-4
removing a Provider instance, 10-3
routing Provider instance, 10-4
Web service providers
oracle.webservices.provider package, 10-1
provider package, 10-1
Web services
guidelines for interoperability, 1-3
Java classes
limitations, E-4
Providers, 10-1
Web services management schema (WSMGMT), A-1
Web services security, 4-1
web, attribute for <port> element, A-3
WebRowSet format, E-5
WebServiceOperation MBean, C-2
WebServicePort MBean, C-2
WebServicesAssembler
auditing support, 6-9
configuring J2EE client management information, 3-9
configuring J2SE client management information, 3-7
configuring server side management information, 3-5
limitations, E-8
logging support, 6-9
WebServicesAssembler arguments
wsifDbBinding, 9-12
wsifDbPort, 9-14, 9-15
wsifEjbBinding, 9-7
wsifEjbPort, 9-8, 9-10
wsifJavaBinding, 9-3
wsifJavaPort, 9-3, 9-4, 9-5
WebServicesAssembler commands
aqAssemble, 9-12, 9-14, 9-15
assemble, 9-3, 9-4, 9-5
dbJavaAssemble, 9-12, 9-14, 9-15
ejbAssemble, 9-7, 9-8, 9-10
genInterface, 9-20
genWsdl, 9-3, 9-4, 9-5, 9-7, 9-8, 9-10, 9-12, 9-14, 9-15
plsqlAssemble, 9-12, 9-14, 9-15
sqlAssemble, 9-12, 9-14, 9-15
WSDL
with multiple namespaces
generating code into a single package, B-3
WSDL file
adding swaRef MIME attachments, 2-4
assembling with swaRef MIME attachments, 2-8
changing the names of WSDL artifacts, B-3
customizing with the maping file, B-2
DIME encoded attachments, 2-22
streaming attachments, 2-19
WSDLs and interoperability issues, 1-6
WS-I basic profile-compliant Web service, defined, 1-2
WSIF
JDeveloper support, 9-24
limitations, E-11
tool support, 9-24
WSDL extensions for database resources, 9-16
WSDL extensions for EJBs, 9-10
WSDL extensions for Java classes, 9-6
WSIF API, 9-17
WSIF architecture, 9-1
WSIF clients
accessing the database, 9-20
adding a Web service management configuration, 9-21
adding attachments, 9-23
with OracleCall API, 9-24
using a dynamic proxy, 9-18
writing, 9-16
WSIF endpoint
configuring for a single EJB port, 9-7
configuring for a single Java port, 9-3
configuring for Java classes, 9-3
configuring for multiple EJB ports, 9-10
configuring for multiple Java ports, 9-5
WSIF, described, 9-1
wsifDbBinding argument, 9-12
wsifDbPort argument, 9-14, 9-15
wsifEjbBinding argument, 9-7
wsifEjbPort argument, 9-8, 9-10
wsifJavaBinding argument, 9-3
wsifJavaPort argument, 9-3, 9-4, 9-5
WS-I.org, 1-2
WSM_INTERCEPTOR_PIPELINE_CONFIG property, 3-9
wsmgmt.xml management policy file, 3-3, 3-5, A-1, A-2, E-10
wsmgmt.xml management policy file, listing, A-3
WSMHandlerGlobalConfig MBean, C-3
WSMHandlerOperationConfig MBean, C-3, C-4
WSMHandlerServiceConfig MBean, C-3
WSMOperationConfig MBean, C-3
WSMServiceConfig MBean, C-3

X

XDB rowset format, E-5
XML characters and interoperability issues, 1-9
xpath queries, E-10