Oracle® Application Server

Web Services Developer's Guide

10*g* (10.1.3.5.0)

E13982-01

July 2009

ORACLE®

Oracle Application Server Web Services Developer's Guide, 10*g* (10.1.3.5.0)

E13982-01

Copyright © 2006, 2009, Oracle and/or its affiliates. All rights reserved.

Primary Author: Joseph Ruzzi

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

Contributor: Ellen Siegal, editor

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this software or related documentation is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. 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, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of this software. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software in dangerous applications.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

This software and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

Preface xix

Intended Audience xix
Documentation Accessibility xix
Related Documents xx
Conventions xxiv

1 Web Services Overview

Understanding Web Services 1-1
Web Services Standards 1-2
Java 2 Enterprise Edition 1-2
Simple Object Access Protocol 1.1 and 1.2 1-2
Web Service Description Language 1.1 1-3
Web Service-Interoperability Basic Profile 1.1 1-3
Web Service Inspection Language 1.0 1-3
New and Enhanced Features 1-4
Web Service Security for Authentication, Integrity, and Confidentiality 1-4
Web Services Management Framework and Application Server Control 1-4
Web Services Metadata for the Java Platform (J2SE 5.0 Web Service Annotations) 1-5
REST Web Services 1-5
Enhanced Web Service Test Page 1-5
Ant Tasks for Configuration and Scripting 1-6
Custom Type Mapping Framework for Serialization 1-6
Database Web Services 1-6
SOAP Header Support 1-6
MIME and DIME Document Support 1-7
MTOM Binary Content as Attachments 1-7
Message Delivery Quality of Service 1-7
JMS Transport as an Alternative to HTTP 1-8
Web Services Provider Support 1-8
Web Services Invocation Framework for Describing WSDL Programming Artifacts 1-8
SOAP Message Auditing and Logging 1-8
Oracle BPEL 1-9
Oracle Web Services Manager 1-9
Compatibility with Previous Versions of OracleAS Web Services 1-9
Redeploying Applications on OracleAS Web Services 10.1.3.1 1-10

Deprecated Features 1-10
Clustered Environments and High Availability 1-10
OC4J in a Standalone Versus Oracle Application Server Environment 1-11

2 Development and Documentation Roadmap

3 OracleAS Web Services Architecture and Life Cycle

Architecture 3-1
Processing Components 3-1
Protocol Handlers 3-2
XML Processing 3-2
Policy Enforcement 3-2
JAX-RPC Handlers 3-3
Data Binding 3-3
Endpoint Implementation 3-3
Java Management Extensions (JMX) 3-4
Development Tools 3-4
Web Services Development Life Cycle 3-4
Create the Implementation 3-5
Assemble the Web Service 3-5
Assemble the Client 3-5
Deploy the Web Service 3-6
Test the Web Service 3-6
Perform Post Deployment Tasks 3-6

4 Getting Started

Supported Platforms 4-1
How to Install OC4J 4-1
How to Set Up Your Environment for OracleAS Web Services 4-1
How to Set Up Ant for WebServicesAssembler 4-3
How to Set Up Ant 1.6.5 Distributed with Oracle Application Server 4-3
How to Set Up Ant 1.6.5 Using a Previous Installation of Ant 4-4
How to Set Up Ant 1.5.2 Using a Previous Installation of Ant 4-5
How to Set Up the "oracle:" Namespace Prefix for Ant Tasks 4-6
Database Requirements 4-6

5 OracleAS Web Services Messages

OracleAS Web Services Message Formats 5-1
Understanding Message Formats 5-2
RPC and Document Styles 5-2
Literal and Encoded Uses 5-2
Supported Message Formats 5-2
Document-Literal Message Format 5-3
Sample Request Message with the Document-Literal Message Format 5-3
RPC-Encoded Message Format 5-4
Sample Messages with the RPC-Encoded Message Format 5-4

The xsi:type Attribute in RPC-Encoded Message Formats 5-5
Oracle Specific Type Support 5-6
Restrictions on RPC-Encoded Format 5-8
RPC-Literal Message Format 5-8
Sample Request Message with the RPC-Literal Message Format 5-9
How to Select Message Formats for a Web Service 5-9
Changing Message Formats in a Service Implementation 5-11
Message Format Recommendations 5-11
Working with SOAP Messages 5-11
Understanding the OraSAAJ APIs 5-12
How to Use the OraSAAJ APIs to Manipulate SOAP Messages 5-13
How to Work with Messages with Attachments Containing Binary Data 5-14
How to Convert XML Elements to SOAP Elements 5-14
How to Use SOAP 1.2 Formatted Messages in Bottom Up Web Service Assembly 5-14
How to Use SOAP 1.2 Formatted Messages in Top Down Web Service Assembly 5-15
Limitations 5-16
Additional Information 5-16

6 Assembling a Web Service from WSDL

Understanding Top Down Assembly 6-1
How to Assemble a Web Service Top Down 6-2
Prerequisites 6-2
Steps for Assembling the Web Service Top Down 6-3
Limitations 6-7
Additional Information 6-7

7 Assembling a Web Service with Java Classes

Requirements for Writing Java Classes for Web Services 7-1
Java Classes and Stateless Web Services 7-2
Exposing Java Classes as a Stateless Web Service 7-3
How to Assemble A Stateless Web Service with Java Classes 7-3
Prerequisites 7-3
Steps for Assembling a Stateless Web Service with Java Classes 7-4
How to Write Java Implementations for Stateless Web Services 7-7
How to Define a Java Interface 7-7
How to Define a Java Class 7-8
Java Classes and Stateful Web Services 7-9
Exposing Java Classes as a Stateful Web Service 7-9
How to Assemble a Stateful Web Service 7-9
Prerequisites 7-9
Steps for Assembling a Stateful Web Service 7-9
How to Write Java Implementations for Stateful Web Services 7-13
How to Define a Java Interface 7-13
How to Define a Java Class 7-14
Packaging and Deploying Web Services 7-14
Tool Support for Exposing Java Classes as Web Services 7-14

Limitations ... 7-14
Additional Information ... 7-15

8 Assembling a Web Service with EJBs

Exposing EJBs as Web Services ... 8-1
Working with Version 2.0 EJBs ... 8-2
Working with Version 3.0 EJBs ... 8-2
How to Assemble a Web Service with EJBs ... 8-2
Prerequisites ... 8-2
Steps for Assembling a Web Service from an EJB ... 8-3
How to Write EJBs for Web Services ... 8-6
How to Write an EJB Service Endpoint Interface ... 8-6
How to Implement an EJB Interface for Web Services ... 8-7
Packaging and Deploying Web Services that Expose EJBs ... 8-8
How to Secure EJB-Based Web Services at the Transport-Level ... 8-9
Tool Support for Exposing EJBs as a Web Service ... 8-9
Limitations ... 8-9
Additional Information ... 8-9

9 Assembling Web Services with JMS Destinations

Understanding JMS Endpoint Web Services ... 9-1
Steps for Assembling a JMS Endpoint Web Service ... 9-3
Message Processing and Reply Messages ... 9-7
Limitations ... 9-7
Additional Information ... 9-8

10 Assembling Database Web Services

Understanding Database Web Services ... 10-1
Type Mapping Between SQL and XML ... 10-3
SQL to XML Type Mappings for Web Service Call-Ins ... 10-4
Changing the SQL to XML Mapping for Numeric Types ... 10-6
XML to SQL Type Mapping for Web Service Call-Outs ... 10-6
Developing Web Services that Expose Database Resources ... 10-6
How to Use Life Cycle for Web Service Call-in ... 10-7
WebServicesAssembler Support for Web Service Call-in ... 10-7
How to Assemble a Web Service from a PL/SQL Package ... 10-9
Prerequisites ... 10-9
Steps for Assembling a Web Service from a PL/SQL Package ... 10-9
Sample PL/SQL Package ... 10-12
How PL/SQL Function Names are Mapped to Web Service Operation Names ... 10-13
How to Access PL/SQL IN and IN OUT Parameters from Client Code ... 10-13
How to Access SQL XMLType Types from Client Code ... 10-15
How to Assemble a Web Service from a SQL Query or DML Statement ... 10-15
Prerequisites ... 10-15
Steps for Assembling a Web Service from a SQL Statement or Query ... 10-16
Sample SQL Statements ... 10-18

How SQL Queries are Mapped to Web Service Operations 10-18
How DML Operations are Mapped to Web Service Operations 10-22
How to Assemble a Web Service from an Oracle Streams AQ 10-22
Prerequisites 10-23
Steps for Assembling a Web Service from an Oracle AQ 10-23
Sample AQ Queue and Topic Declaration 10-26
Sample Web Service for a Queue Generated by WebServicesAssembler 10-26
Sample Web Service for a Topic Generated by WebServicesAssembler 10-28
How to Access an AQ Queue Exposed as a Web Service from Client Code 10-31
How to Access an Oracle AQ Queue with JMS 10-31
How to Assemble a Server-Side Java Class as a Web Service 10-32
Prerequisites 10-32
Steps for Assembling a Web Service from a Server-Side Java Class 10-32
Sample Server-Side Java Class 10-35
Sample Web Service Operations Generated from a Server-Side Java Class 10-35
Developing a Web Service Client in the Database 10-36
Understanding Web Service Call-Out 10-36
How to Call Web Services from the Database 10-38
How to Perform Web Services Call-Out using Static Proxy and JPublisher 10-38
How to Perform Web Services Call-Out using DII and the SYS.UTL_DBWS Utility . 10-40
Tool Support for Web Services that Expose Database Resources 10-40
Limitations 10-41
Additional Information 10-41

11 Assembling Web Services with Annotations

OracleAS Web Services and J2SE 5.0 Web Service Annotations 11-1
Supported Annotations 11-2
Supported J2SE 5.0 JDK Annotations 11-2
Using WebServiceRef within OracleAS Web Services 11-3
How the WebServiceRef Annotation Determines the Mapping File Name 11-4
Specifying WSDLs with Multiple Ports in the WebServiceRef Annotation 11-4
WebServiceRef Annotation Examples 11-5
Oracle Additions to J2SE 5.0 JDK Annotations 11-6
Deployment Annotation 11-7
Schema Annotation 11-8
WSIFEJBBinding Annotation 11-9
How to Use Annotations 11-11
Steps to Use Annotations to Assemble a Web Service from Java Classes 11-11
Steps to Use Annotations to Assemble a Web Service from a Version 3.0 EJB 11-12
How to Override Annotations 11-13
How to Override Annotation Values with WebServicesAssembler 11-13
How to Override Deployment Annotation Values with Deployment Descriptors 11-13
Sample Java File with Annotations 11-14
Limitations 11-16
Additional Information 11-16

12 Assembling REST Web Services

Understanding REST Web Services ... 12-1
WebServicesAssembler Support for REST Web Services ... 12-1
Commands that Can Assemble REST Web Services ... 12-2
Annotations that Support REST Web Services ... 12-2
How to Assemble REST Web Services ... 12-2
Steps to Assemble a REST Web Service Top Down ... 12-2
How to Access REST Web Service Operations ... 12-4
Steps to Assemble a REST Web Service Bottom Up ... 12-6
How to Access REST Web Service Operations ... 12-8
How to Test REST Web Services ... 12-9
REST Additions to Deployment Descriptors ... 12-9
How REST Builds Requests and Responses ... 12-9
HTTP GET Requests ... 12-9
HTTP POST Requests ... 12-11
REST Responses ... 12-11
Tool Support for REST Web Services ... 12-12
Limitations ... 12-12
Additional Information ... 12-12

13 Testing Web Services Deployment

Steps to Access and Test Web Service Operations ... 13-1
How to Access the Web Services Test Page ... 13-2
Accessing the Web Services Test Page through a Web Browser ... 13-2
Accessing the Web Services Test Page through Application Server Control ... 13-3
Accessing a Web Services Test Page Protected with SSL ... 13-3
Accessing the SSL-Protected Test Page when OracleAS Web Services Runs as Standalone ... 13-4
Accessing the SSL-Protected Test Page when OracleAS Web Services runs on Oracle HTTP Server 13-5
How to Use the Web Services Test Page ... 13-6
Viewing the Web Service WSDL ... 13-7
Editing Values in the Test Page ... 13-9
How to Edit the Test Page as XML Source ... 13-9
Testing WS-Security and Reliable Messaging Features ... 13-10
Reliable Messaging Parameters ... 13-10
WS-Security Parameters ... 13-11
Enabling HTTP Authentication for the Web Service Test ... 13-11
Stress Testing the Web Service Operation ... 13-12
Invoking the Web Service Test ... 13-13
Invoking the Test for a JAX-RPC Web Service ... 13-13
Invoking the Test for a REST Web Service ... 13-14
Invoke ... 13-15
Invoke REST POST ... 13-16
Invoke REST GET ... 13-16
How to Disable the Web Services Test Page ... 13-17
How to Obtain a Web Service WSDL Directly ... 13-18

Limitations ... 13-18
Additional Information ... 13-18

14 Assembling a J2EE Web Service Client

Understanding J2EE Web Service Clients ... 14-1
How to Assemble a J2EE Web Service Client ... 14-1
Prerequisites ... 14-2
Steps to Assemble a J2EE Web Service Client ... 14-2
How to Add J2EE Web Service Client Information to Deployment Descriptors ... 14-4
Steps to Deploy and Run an Application Client Module ... 14-5
How to Access a Web Service from a Client in the Same Module ... 14-6
How to Add a Port Component Link to a J2EE Client Deployment Descriptor ... 14-6
How to Add OC4J-Specific Platform Information for Deployment and Runtime ... 14-7
How to Add JAX-RPC Handlers for Message Processing to Deployment Descriptors ... 14-12
Writing J2EE Web Service Client Code ... 14-12
Steps for Writing a J2EE Web Service Client ... 14-12
How to Configure a J2EE Web Service Client for a Stateful Web Service ... 14-13
How to Configure a J2EE Client with Deployment Descriptor Files ... 14-14
How to Configure a J2EE Client Programmatically ... 14-14
How to Configure a J2EE Web Service Client to Make JMS Transport Calls ... 14-15
How to Enable Chunked Data Transfer for HTTP 1.1 ... 14-15
How to Set a Character Encoding for a SOAP Message ... 14-16
Understanding the Packaging Structure for J2EE Clients ... 14-17
Understanding the Packaging Structure for a Servlet or Web Application Client ... 14-17
Packaging Structure for Servlet or Web Application Clients ... 14-17
Relationship Between Deployment Descriptors for Servlet or Web Application Clients ... 14-18
Understanding the Packaging Structure for an EJB Client ... 14-19
Package Structure for EJB Application Clients ... 14-19
Relationship Between Deployment Descriptors for EJB Application Clients ... 14-20
Limitations ... 14-21
Additional Information ... 14-21

15 Assembling a J2SE Web Service Client

Understanding J2SE Web Service Clients ... 15-1
Understanding Static Stub Clients ... 15-1
Understanding JAX-RPC Dynamic Invocation Interface ... 15-2
How to Assemble a J2SE Web Service Client ... 15-2
Prerequisites ... 15-2
Steps to Assemble a J2SE Web Service Client with a Static Stub ... 15-2
Sample WSDL File ... 15-3
Writing Web Service Client Applications ... 15-5
Understanding the Client Utility Class File ... 15-5
Steps to Create a Service Factory ... 15-6
How to Enable Chunked Data Transfer for HTTP 1.1 ... 15-8
How to Set a Character Encoding for a SOAP Message on a J2SE Client ... 15-8

How to Set Cookies in a Web Service Client Stub 15-8
How to Use Dynamic Invocation Interface to Invoke Web Services 15-9
Basic Calls 15-9
Configured Calls 15-10
Examples of Web Service Clients that use DII 15-12
Tool Support for Assembling J2SE Web Service Clients 15-15
Additional Information 15-15

16 Using JAX-RPC Handlers

Understanding Message Handlers 16-1
How to Write a JAX-RPC Handler 16-2
How to Configure and Register a Server-Side Handler with Ant Tasks 16-2
Ant Tasks that can Configure and Register Handlers 16-3
How to Edit webservices.xml to Register a Server-Side Handler 16-3
Client-Side JAX-RPC Handlers 16-4
How to Register a JAX-RPC Handler for J2EE Web Service Client 16-4
Using the handler Element in a J2EE Web Service Client 16-5
How to Register a JAX-RPC Handler for a J2SE Web Service Client 16-6
Limitations 16-6
Additional Information 16-6

17 Processing SOAP Headers

How to Process SOAP Headers with Parameter Mapping 17-1
Ant Tasks and Commands that can Map Headers to Parameters 17-3
How to Process SOAP Headers by Using Handlers 17-3
How to Process SOAP Headers by Using the ServiceLifecycle Interface 17-4
How to Get HTTP Response and Request Headers 17-4
How Stub Clients Can Get Headers with the ServiceLifecycle Interface 17-5
How DII Clients Can Get Headers with the OracleCall Interface 17-6
How DII Clients of WSIF Services Can Get Message Headers 17-6
Limitations 17-6
Additional Information 17-6

18 Using WebServicesAssembler

About the WebServicesAssembler Tool 18-1
Command Line Syntax 18-3
Setting Up Ant for WebServicesAssembler 18-3
WebServicesAssembler Commands 18-3
Web Service Assembly Commands 18-4
aqAssemble 18-5
assemble 18-7
corbaAssemble 18-10
dbJavaAssemble 18-12
ejbAssemble 18-14
jmsAssemble 18-16
plsqlAssemble 18-17

sqlAssemble 18-19
topDownAssemble 18-22
WSDL Management Commands 18-24
analyze 18-24
fetchWsdl 18-25
genConcreteWsdl 18-26
genQosWsdl 18-27
genWsdl 18-28
Java Generation Commands 18-31
genInterface 18-31
genProxy 18-34
genValueTypes 18-36
Deployment Descriptor Generation Commands 18-38
genApplicationDescriptor 18-38
genDDs 18-38
Maintenance Commands 18-40
help 18-40
version 18-41
WebServicesAssembler Arguments 18-41
General Web Services Assembly Arguments 18-41
appName 18-42
bindingName 18-42
classFileName 18-42
className 18-43
classpath 18-43
debug 18-43
ear 18-44
ejbName 18-45
emptySoapAction 18-45
help 18-46
initialContextFactory 18-46
input 18-46
interfaceFileName 18-47
interfaceName 18-47
jndiName 18-47
jndiProviderURL 18-47
mappingFileName 18-47
output 18-48
packageName 18-48
portName 18-48
portTypeName 18-49
restSupport 18-49
schema 18-49
searchSchema 18-50
serviceName 18-50
strictJaxrpcValidation 18-50
useDimeEncoding 18-50

war ... 18-51
Session Arguments ... 18-51
callScope ... 18-52
recoverable ... 18-52
session ... 18-52
timeout ... 18-52
CORBA Assembly Arguments ... 18-52
corbanameURL ... 18-53
corbaObjectPath ... 18-53
idlFile ... 18-53
idlInterfaceName ... 18-53
idljPath ... 18-53
ORBInitialHost ... 18-53
ORBInitialPort ... 18-53
ORBInitRef ... 18-53
Database Assembly Arguments ... 18-53
aqConnectionFactoryLocation ... 18-54
aqConnectionLocation ... 18-54
dataSource ... 18-54
dbConnection ... 18-54
dbJavaClassName ... 18-55
dbUser ... 18-55
jpubProp ... 18-55
sql ... 18-55
sqlstatement ... 18-56
sqlTimeout ... 18-57
sysUser ... 18-57
useDataSource ... 18-57
wsifDbBinding ... 18-57
wsifDbPort ... 18-57
JMS Assembly Arguments ... 18-58
deliveryMode ... 18-58
genJmsPropertyHeader ... 18-59
jmsTypeHeader ... 18-59
linkReceiveWithReplyTo ... 18-59
payloadBindingClassName ... 18-59
priority ... 18-59
receiveConnectionFactoryLocation ... 18-59
receiveQueueLocation ... 18-59
receiveTimeout ... 18-60
receiveTopicLocation ... 18-60
replyToConnectionFactoryLocation ... 18-60
replyToQueueLocation ... 18-60
replyToTopicLocation ... 18-60
sendConnectionFactoryLocation ... 18-60
sendQueueLocation ... 18-61
sendTopicLocation ... 18-61

timeToLive 18-61
topicDurableSubscriptionName 18-61
Proxy Arguments 18-61
endpointAddress 18-61
genJUnitTest 18-61
Deployment Descriptor Arguments 18-62
appendToExistingDDs 18-62
context 18-62
ddFileName 18-63
uri 18-63
WSDL Access Arguments 18-64
fetchWsdlImports 18-64
httpNonProxyHosts 18-64
httpProxyHost 18-64
httpProxyPort 18-64
importAbstractWsdl 18-64
wsdl 18-65
WSDL Management Arguments 18-65
createOneWayOperations 18-65
genQos 18-65
qualifiedElementForm 18-65
singleService 18-66
soapVersion 18-66
targetNamespace 18-66
typeNamespace 18-67
wsdlTimeout 18-67
Message Format Arguments 18-67
mtomSupport 18-68
style 18-68
use 18-68
Java Generation Arguments 18-68
dataBinding 18-69
mapHeadersToParameters 18-69
overwriteBeans 18-69
unwrapParameters 18-69
valueTypeClassName 18-69
valueTypePackagePrefix 18-70
wsifEjbBinding 18-70
wsifJavaBinding 18-71
Resolving Name Collisions 18-71
Default Algorithms to Map Between Target WSDL Namespaces and Java Package Names 18-72
Java Package Name to WSDL Namespace Mapping Algorithm 18-72
Mapping Java Artifacts to WSDL Artifacts 18-73
Mapping Java Types to XML Schema Types 18-73
WSDL Namespace to Java Package Name Mapping Algorithm 18-73

Mapping the WSDL Service Endpoint Interface and Related Endpoint Artifacts to Java Package and Class Names 18-74
Mapping WSDL Value Types and Related Artifacts to Java Names and Types 18-75
How to Specify a Namespace 18-75
How to Specify a Root Package Name 18-75
How to Establish a Database Connection 18-75
Additional Ant Support for WebServicesAssembler 18-76
How to Use Multiple Instances of an Argument in Ant 18-76
How to Configure Proxy Generation in an Ant Task 18-77
How to Generate Handler and Port Information into a Proxy 18-78
How to Configure a Port in an Ant Task 18-78
How to Configure a Port Type in an Ant Task 18-80
How to Configure Handlers in an Ant Task 18-81
Attributes and Child Tags for handler Tags 18-81
Sample Handler Configuration 18-83
Ant Tasks that Can Configure Handlers 18-83
Configuring Multiple Handlers in an Ant Task 18-84
How to Add Files to an Archive 18-84
How to Control a WebServicesAssembler Build 18-84
How to Assemble Support for MTOM Encoded Attachemnts into a Web Service 18-85
How to Assign Multiple Web Services to an EAR or WAR Archive 18-85
Limitations on Assigning Multiple Web Services to a WAR File 18-87
How to Represent Java Method Parameter Names in the WSDL 18-87
Limitations 18-88
Additional Information 18-88

19 Packaging and Deploying Web Services

Understanding Web Service Packaging 19-1
Packaging Structure for Web Service Applications 19-2
Packaging for a Web Service Based on Java Classes 19-2
Packaging for a Web Service Based on EJBs 19-3
Descriptions of Packaged Files 19-3
Relationships Between Deployment Descriptor Files 19-5
Relationships Between webservices.xml and ejb-jar.xml 19-5
Relationships Between webservices.xml and oracle-webservices.xml 19-5
Relationships Between webservices.xml and web.xml 19-6
Tool Support for Packaging 19-7
Packaging Support with WebServicesAssembler 19-7
WebServicesAssembler Commands that Perform Packaging 19-7
Managing Deployment Descriptors 19-8
Commands that Create Deployment Descriptors 19-8
Arguments that Affect Deployment Descriptor Contents 19-9
Packaging Support with Oracle JDeveloper 19-10
Understanding Web Service Deployment 19-10
Tool Support for Deployment 19-11
Command Line Support for Deployment 19-11
A Sample Deployment Using admin_client.jar 19-11

Ant Task Support for Deployment 19-11
Deployment Support with Oracle JDeveloper 19-12
Deployment Support with Application Server Control 19-13
Limitations 19-13
Additional Information 19-13

A Web Service Client APIs and JARs

Web Services API Packages A-1
Setting the Classpath for a Web Service Proxy A-2
Simplifying the Classpath with wsclient_extended.jar A-2
Classpath Components for Clients using a Client-Side Proxy A-3
OC4J Security-Related Client JAR Files A-4
WS-Security-Related Client JAR Files A-4
Reliability-Related Client JAR File A-5
JMS Transport-Related Client JAR File A-5
Database Web Services-Related Client JAR Files A-5
Sample Classpath Commands A-6

B Oracle Implementation of the WSDL 1.1 API

Understanding the OraWSDL APIs B-1

C oracle-webservices.xml Deployment Descriptor Schema

Hierarchy of XML Elements in oracle-webservices.xml C-1
Elements and Attributes of oracle-webservices.xml C-3
<auth-method> C-3
<context-root> C-3
<download-external-imports> C-3
<ejb-transport-login-config> C-4
<ejb-transport-security-constraint> C-4
<endpoint-address-uri> C-4
<expose-testpage> C-5
<expose-wsdl> C-5
<implementation-class> C-5
<implementor> C-5
<jms-address> C-6
<max-request-size> C-6
<operation> C-7
<operations> C-7
<oracle-webservices> C-7
<param> C-8
<policy> C-8
<port-component> C-8
<property> C-9
<provider-description> C-9
<provider-description-name> C-9
<provider-name> C-9

<provider-port> C-10
<realm-name> C-10
<resolve-relative-imports> C-10
<rest-support> C-10
<role-name> C-10
<runtime> C-11
<servlet-link> C-11
<soap-port> C-11
<transport-guarantee> C-12
<use-dime-encoding> C-12
<web-site> C-12
<webservice-description> C-13
<wsdl-file> C-13
<wsdl-port> C-14
<wsdl-publish-location> C-14
<wsdl-service-name> C-14
<wsdl-url> C-14
Securing EJB-Based Web Services at the Transport Level C-15
oracle-webservices.xml File Listing C-16

D service-ref-mapping Schema

Hierarchy of service-ref-mapping Schema D-1
Elements and Attributes of service-ref-mapping Schema D-2
<call-property> D-2
<name> D-2
<operation> D-3
<operations> D-3
<port-info> D-4
<runtime> D-4
<service-endpoint-interface> D-4
<service-impl-class> D-4
<service-qname> D-5
<service-ref-mapping> D-5
<stub-property> D-5
<value> D-6
<wsdl-file> D-6
<wsdl-location> D-6
<wsdl-port> D-7
service-ref-mapping File Listing D-7

E Error Message Prefixes

F Troubleshooting

OracleAS Web Services Messages F-1
Assembling Web Services from a WSDL F-5
Schema Features Limitations F-6

Schema Features that are Mapped to a SOAPElement ... F-6
RPC Encoded Does Not Support Complex Types With Attributes ... F-6
Assembling Web Services from Java Classes ... F-6
Assembling Web Services From EJBs ... F-7
Assembling Web Services with JMS Destinations ... F-7
Developing Web Services From Database Resources ... F-7
Assembling Web Services with Annotations ... F-8
Assembling REST Web Services ... F-8
Testing Web Service Deployment ... F-9
Assembling a J2EE Web Service Client ... F-9
Understanding JAX-RPC Handlers ... F-9
Processing SOAP Headers ... F-9
Using WebServicesAssembler ... F-10
Packaging and Deploying Web Services ... F-12
Ensuring Interoperable Web Services ... F-13
Working with Message Attachments ... F-14
Managing Web Services ... F-14
Ensuring Web Service Reliability ... F-14
Auditing and Logging Messages ... F-15
Custom Serialization of Java Value Types ... F-15
Using JMS as a Web Service Transport ... F-16
Using the Web Service Invocation Framework ... F-16
Using Web Service Providers ... F-16

G Third Party Licenses

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

Index

Preface

This guide describes how to use Oracle Application Server Web Services and the Oracle 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.

This preface contains these topics:

- Intended Audience
- Documentation Accessibility
- Related Documents
- Conventions

Intended Audience

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

- Assemble Web services from Java classes, EJBs, Database Resources, and JMS queues
- Assemble J2SE and J2EE Web service clients
- Work with SOAP messages in RPC-literal, RPC-encoded and document-literal formats

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. 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.

Deaf/Hard of Hearing Access to Oracle Support Services

To reach Oracle Support Services, use a telecommunications relay service (TRS) to call Oracle Support at 1.800.223.1711. An Oracle Support Services engineer will handle technical issues and provide customer support according to the Oracle service request process. Information about TRS is available at `http://www.fcc.gov/cgb/consumerfacts/trs.html`, and a list of phone numbers is available at `http://www.fcc.gov/cgb/dro/trsphonebk.html`.

Related Documents

For more information, see these Oracle resources:

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

  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.

  For your convenience, "Contents of the Oracle Application Server Advanced Web Services Developer's Guide" on page xxi lists the contents of the *Oracle Application Server Advanced Web Services Developer's Guide*.

- *Oracle Application Server Web Services Java API Reference*

  The *Reference* provides the output of the Javadoc tool for the OracleAS Web Services Java API.

- *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. The strategies that can be employed are 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 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 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*

Oracle Web Services Manager is a comprehensive solution for managing service oriented architectures. It allows IT managements to centrally define policies that govern Web services operations such as access policy, logging policy, and content validation, and then wrap these policies around services, with no modification to existing web services required.

- *Oracle Web Services Manager Quick Start Guide*
- *Oracle Web Services Manager Installation Guide*
- *Oracle Web Services Manager Administrator's Guide*
- *Oracle Web Services Manager Deployment Guide*
- *Oracle Web Services Manager Extensibility Guide*

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

`http://oraclestore.oracle.com/`

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

This book is designed to be used with the *Oracle Application Server Advanced Web Services Developer's Guide*. The "*Advanced*" book describes topics beyond basic Web service assembly.

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

- Chapter 1, "Ensuring Interoperable Web Services"
- Chapter 2, "Working with Message Attachments"
- Chapter 3, "Managing Web Services"
- Chapter 4, "Ensuring Web Services Security"
- Chapter 5, "Ensuring Web Service Reliability"
- Chapter 6, "Auditing and Tracing Messages"
- Chapter 7, "Implementing Custom Serialization of Java Value Types"
- Chapter 8, "Using JMS as a Web Service Transport"
- Chapter 9, "Using Web Services Invocation Framework"
- Chapter 10, "Using Web Service Providers"
- Appendix A, "Understanding the Web Services Management Schema"
- Appendix B, ""Oracle Web Services Client Schema"
- Appendix C, "OracleAS Web Services Reliability Schema"
- Appendix D, ""OracleAS Web Services Auditing Schema"
- Appendix E, "OracleAS Web Services Tracing Schema"
- Appendix F, "JAX-RPC Mapping File Descriptor"
- Appendix G, "Web Service MBeans"
- Appendix H, "Mapping Java Types to XML and WSDL Types"
- Appendix I, "Troubleshooting"
- Appendix J, "Third Party Licenses"

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 5 (1.5) API specification:

  `http://java.sun.com/javaee/5/docs/api`

- 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 OC4J schemas, including proprietary deployment descriptors:

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

- Oracle UDDI v2.0 server implementation:

  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/

- 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

Web Services Overview

This chapter provides an overview of Oracle Application Server Web Services for the 10.1.3.1 release. Chapter 3, "OracleAS Web Services Architecture and Life Cycle", describes the architecture of Oracle Application Server Web Services.

- Understanding Web Services
- Web Services Standards
- New and Enhanced Features
- Compatibility with Previous Versions of OracleAS Web Services
- Clustered Environments and High Availability
- OC4J in a Standalone Versus Oracle Application Server Environment

Understanding Web Services

Web services comprise a set of messaging protocols, programming standards, and network registration and discovery facilities. When they are used together, these features enable the publication of business functions to authorized parties over the Internet from any device connected to the Web.

A Web service is a software application identified by a Universal Resource Identifier (URI), whose interfaces and binding are capable of being defined, described, and discovered by XML artifacts. A Web service supports direct interactions with other software applications using XML-based messages and Internet-based products.

A Web service:

- Exposes and describes itself—A Web service defines its functionality and attributes so that other applications can understand it. By providing a Web Service Description Language (WSDL) file, a Web service makes its functionality available to other applications.
- Enables other services to locate it on the Web—A Web service can be registered in a Universal Description, Discover, and Integration (UDDI) Registry so that applications can locate it.
- Can be invoked—Once a Web service has been located and examined, the remote application can invoke the service using an Internet standard protocol.
- A Web service style is either two-way request and response, or one-way. It can use either synchronous or asynchronous communication. However, the fundamental unit of exchange between a Web service client and a Web service, of either style or type of communication, is a message.

Web services offer a standards-based infrastructure through which any business can do the following:

- Offer appropriate internal business processes as value-added services that can be used by other organizations
- Integrate its internal business processes and dynamically link them with those of its business partners

Web Services Standards

With the current release, Oracle has extended the Web services infrastructure to implement versions 1.4 and 5 (1.5) of the Java 2 Enterprise Edition (J2EE) specification for Web services. This section lists the standards that this release of Web services complies with.

- Java 2 Enterprise Edition
- Simple Object Access Protocol 1.1 and 1.2
- Web Service Description Language 1.1
- Web Service-Interoperability Basic Profile 1.1
- Web Service Inspection Language 1.0

Java 2 Enterprise Edition

The current release of Web services is compatible with these Java J2EE standards:

- JAX-RPC 1.1—defines client APIs, support for message handlers, and ways to implement service endpoints
- EJB 2.1—defines the EJB API. The standard JAX-RPC 1.1 specifies how to expose an EJB as a Web service endpoint
- SAAJ 1.2—defines how to process SOAP messages with attachments
- Enterprise Web Services 1.1 specification—(also known as JSRs 109 and 921) specifies how to deploy and execute Web services

Simple Object Access Protocol 1.1 and 1.2

The Simple Object Access Protocol (SOAP) is a lightweight, XML-based protocol for exchanging information in a decentralized distributed environment. SOAP supports different styles of information exchange, including Remote Procedure Call style (RPC) and message-oriented exchange. RPC style information exchange allows for synchronous request and response processing, where an endpoint receives a procedure oriented message and replies with a correlated response message. Message-oriented information exchange supports organizations and applications that must exchange business or other styles of documents in which a message is sent but the sender may not expect or wait for an immediate response (asynchronous). Message-oriented information exchange is also called document-style exchange.

SOAP has these features:

- Protocol independence
- Language independence
- Platform and operating system independence
- Support for RPC/Encoded and Document/Literal message formats

- Support for SOAP XML messages incorporating attachments (using the multipart MIME structure)

The current release of Oracle Application Server Web services supports the SOAP 1.2 protocol and is backward compatible with SOAP 1.1.

See Also:

`http://www.w3.org/TR/SOAP` for more detailed information on the SOAP 1.1 and 1.2 specifications.

Web Service Description Language 1.1

Web Services Description Language (WSDL) is an XML format for describing network services containing RPC-oriented and message-oriented information. Programmers or automated development tools can create WSDL files to describe a service and can make this description available over the Internet. Client-side programmers and development tools can use published WSDL descriptions to obtain information about available Web services and to build and create client proxies or program templates that access available services.

See Also:

`http://www.w3.org/TR/wsdl` for information on the WSDL format.

Web Service-Interoperability Basic Profile 1.1

Web Service-Interoperability (WS-I) is an organization that promotes Web services interoperability across platforms, applications, and programming languages. The current release conforms to the WS-I Basic Profile 1.1.

Web Service Inspection Language 1.0

Web Services Inspection Language (WS-Inspection or WSIL) is a lightweight Web services directory protocol that provides an extensible schema for a single document catalog of services. The document lists services with some metadata (including extensible data) and links to other directories.

Typically, a WSIL document lists services deployed on a particular server or cluster. It includes anything that is exposed as a provider, including BPEL processes. WSIL allows tighter integration between SOA components in that it provides BPEL and OWSM users with a simple way to browse services deployed to OC4J.

The WSIL document can be viewed and managed as a Web application by Application Server Control and by BPEL and OWSM tools. This application relies on MBeans registered with Oracle's Web services stack to generate its list of services.

The Web application is typically located at the server root or under the `services` context:

`http://`*`yourdomain.com`*`/inspection.wsil`, or

`http://`*`yourdomain.com`*`/services/inspection.wsil`

To move the Web application, you can redeploy it and map it to a different context path.

To increase security, you can disable WSIL by removing the Web application. This will prevent information about other services deployed on the server from being exposed.

New and Enhanced Features

In addition to the preceding standards, the current release of OracleAS Web Services contains these new and enhanced features:

- Web Service Security for Authentication, Integrity, and Confidentiality
- Web Services Management Framework and Application Server Control
- Web Services Metadata for the Java Platform (J2SE 5.0 Web Service Annotations)
- REST Web Services
- Enhanced Web Service Test Page
- Ant Tasks for Configuration and Scripting
- Custom Type Mapping Framework for Serialization
- Database Web Services
- SOAP Header Support
- MIME and DIME Document Support
- Message Delivery Quality of Service
- JMS Transport as an Alternative to HTTP
- Web Services Provider Support
- Web Services Invocation Framework for Describing WSDL Programming Artifacts
- SOAP Message Auditing and Logging
- Oracle BPEL

Web Service Security for Authentication, Integrity, and Confidentiality

The WS-Security standard, published and maintained by the Organization of the Advancement of Structure Information Standards (OASIS), provides profiles for Web services authentication, message encryption, and digital signatures. The current release provides an implementation of WS-Security with the following capabilities:

- XML Signature
- XML Encryption
- Username Token
- X.509 Token
- SAML Token

See Also:

Oracle Application Server Web Services Security Guide for more information on Web service security.

Web Services Management Framework and Application Server Control

The management framework provides configuration and monitoring capabilities for security, reliability, logging, and auditing through the Web-based Application Server Control. These features are exposed through a series of Java Management Extensions (JMX) Management beans (Mbeans).

See Also:

- "Managing Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on Web services management.
- "Web Service MBeans" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on the Web service features that you can control through MBeans.

Web Services Metadata for the Java Platform (J2SE 5.0 Web Service Annotations)

The current release provides support for J2SE 5.0 Web Service annotations (also known as the Web Services Metadata for the Java Platform (JSR-181) specification). The specification defines an annotated Java syntax for programming Web services.

See Also:

Chapter 11, "Assembling Web Services with Annotations" for a description of how annotations are supported and of Oracle extensions to the specification.

REST Web Services

The current release provides support for Representational State Transfer (REST) services. This architecture leverages the architectural principles of the Web. It uses the semantics of HTTP whenever possible.

Unlike SOAP Web Services, REST is a "style" and has no standards or tools support from vendors. Also, REST Web services use XML documents, not SOAP envelopes, for sending messages.

You can assemble REST Web services with the WebServicesAssembler tool or by adding J2SE 5.0 Web Service annotations to your source files. You can then use the Web Service Test Page to see if they deployed successfully and to test their functionality.

See Also:

Chapter 12, "Assembling REST Web Services" for more information on assembling REST Web services.

Enhanced Web Service Test Page

The functionality of the Web Service Test Page has been expanded beyond testing whether the Web Service deployed correctly. The Test Page lets you exercise Web service operations with different input values. You can display and edit the SOAP request that will be sent to the service. It also displays the response returned by the service. If your Web service defines security and reliability features, then editors in the Test Page allow you to exercise the service with different security and reliability values.

The Test Page also enables you to exercise Web Service Providers and REST Web Services. For a REST Web service, the Test Page provides the same functionality as for JAX-RPC Web Services, however, it also lets you invoke the REST POST and REST GET operations.

See Also:

Chapter 13, "Testing Web Services Deployment" for more information on how to use the Web Service Test Page.

Ant Tasks for Configuration and Scripting

The current release provides Ant tasks for Web services development, with a focus on enabling the scripting and automation of Web service client and server development. Sample Ant tasks are provided throughout this manual.

See Also:

- "How to Set Up Ant for WebServicesAssembler" on page 4-3 provides many of the details for setting up Ant for your environment.
- Chapter 18, "Using WebServicesAssembler" provides examples of how to write Ant tasks for Web service assembly commands.

Custom Type Mapping Framework for Serialization

Web services in complex systems are often required to map data types beyond the native types automatically serialized into XML by the Web services runtime. The current release offers a custom type-mapping framework for mapping custom data types.

See Also:

"Custom Serialization of Java Value Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*. for more information on working with nonstandard data types in your Web service.

Database Web Services

The current release continues to support publishing PL/SQL as a Web service and using OracleAS Web Services as the Java runtime in the Oracle Database 10*g* for call-outs to Web services.

The current release extends the runtime and tool support to declaratively define the following database artifacts as Web services:

- PL/SQL stored procedures
- SQL queries
- DML statements
- Java classes loaded within the database virtual Java machine
- Oracle Streams AQ (Advanced Queues)

See Also:

Chapter 10, "Assembling Database Web Services" for more information on implementing database artifacts as a Web service.

SOAP Header Support

The current release facilitates advanced manipulation of SOAP headers using two JAX-RPC-compliant mechanisms:

- A mechanism to programmatically intercept SOAP headers using JAX-RPC handlers. This mechanism allows processing SOAP headers out-of-band from the processing of the main SOAP message.
- A mechanism to automatically map SOAP headers to member variables in Java classes. This mechanism enables you to treat SOAP headers as variables in the

Web service implementation through a declarative mapping process instead of a programmatic process.

See Also:

- Chapter 16, "Using JAX-RPC Handlers" for more information on SOAP headers.
- Chapter 17, "Processing SOAP Headers" for more information on processing SOAP headers.

MIME and DIME Document Support

To facilitate efficient transfer of binary documents and large XML documents, the current release supports MIME and DIME attachments:

- Multipurpose Internet Mail Extensions (MIME) attachments conform to the WS-I Attachment Profile for both WSDL to Java consumption, and Java to WSDL publication of Web services requiring SOAP with MIME attachments.
- Direct Internet Message Encapsulation (DIME) attachments conforming to the Microsoft Corporation analog to MIME. Oracle offers DIME support for backward compatibility with other implementations, because Microsoft no longer encourages DIME for attachments.

See Also:

"Working with Message Attachments" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on how OracleAS Web Services supports MIME and DIME attachments.

MTOM Binary Content as Attachments

The current release enables you to send binary content between the client and server in MTOM (SOAP Message Transmission Optimization Mechanism) format. Using MTOM, binary content can be sent as a MIME attachment, which reduces the transmission size on the wire. The binary is semantically part of the XML document. This is an advantage over SWA (SOAP Messages with Attachments), in that it enables you to apply operations such as WS-Security signature on the message.

See Also:

"Working with Attachments containing Binary Data" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on how OracleAS Web Services supports MIME and DIME attachments.

Message Delivery Quality of Service

The current release provides an OASIS Web Service Reliability (WS-Reliability) implementation with guaranteed at-least-once message delivery, duplicate message elimination (at most once delivery), exactly once message delivery (guaranteed delivery and duplicate elimination), and message ordering within groups of messages.

See Also:

"Ensuring Web Service Reliability" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on reliability.

JMS Transport as an Alternative to HTTP

As an alternative to HTTP, the current release enables the use of JMS queues as a transport for SOAP messages. JMS provides a higher level of reliable message delivery for SOAP messages.

Where HTTP is required as the transport, the current release continues to support the ability to put and get SOAP messages from JMS queues and topics, and adds the functionality of correlating messages processed in this manner.

See Also:

- Chapter 9, "Assembling Web Services with JMS Destinations" for more information on using JMS to develop Web services.
- "Using JMS as a Web Service Transport" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on using JMS as a transport mechanism for Web services.

Web Services Provider Support

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.

See Also:

"Using Web Service Providers" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on Web service providers.

Web Services Invocation Framework for Describing WSDL Programming Artifacts

The Web Services Invocation Framework (WSIF) provides a general purpose, extensible mechanism to describe programmatic artifacts using WSDL and a framework to invoke those artifacts using their native protocols. The current release supports an initial implementation of WSIF.

See Also:

"Using Web Services Invocation Framework" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on WSIF.

SOAP Message Auditing and Logging

The current release provides the ability to record inbound and outbound SOAP messages into logging and auditing files. Entire messages or parts of messages can be logged by subquerying through the use of Xpath statements.

See Also:

"Auditing and Logging Messages" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on logging and auditing messages.

Oracle BPEL

BPEL (Business Process Execution Language) is a standard published by the Organization of the Advancement of Structure Information Standards (OASIS) and backed by major vendors including Oracle. BPEL describes an XML syntax for describing a business process through a "composition of invocations of (other) Business Processes—typically Web services". A BPEL process definition, therefore, is an XML document composed according to an XSD that is maintained by OASIS.

A business process, in BPEL terms, is composed of invocations (call-outs to Web services), receptions (call-ins from external services), decision points with simple conditional logic and parallel flows or sequences, that in turn consist of invocations, decision points, and so on. Variables can be defined as part of a business process. They can be assigned values from parameters passed to the business process at startup, passed along in invocations, given values from the results of invocations, and returned as the final result of the business process. A business process itself is also implemented as a Web service—in the BPEL (runtime) engine.

The Oracle implementation of BPEL uses WSIF technology to directly invoke business processes. In certain cases, the actual invocation from BPEL can be a direct call to a Java class without the Web service overhead of marshalling and unmarshalling SOAP messages.

See Also:

The following Web site provides more information on Oracle BPEL:

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

Oracle Web Services Manager

Oracle Web Services Manager is a comprehensive solution for managing service oriented architectures. It allows IT managements to centrally define policies that govern Web services operations such as access policy, logging policy, and content validation, and then wrap these policies around services, with no modification to existing web services required. Also, Oracle Web Services Manager collects monitoring statistics to ensure service levels and security, and displays them in a web dashboard. As a result, Oracle Web Services Manager brings enterprises better control and visibility over their web services.

See Also:

The following Web site provides more information on Oracle Web Services Manager:

http://www.oracle.com/technology/products/webservices_manager/index.html

Compatibility with Previous Versions of OracleAS Web Services

The 10.1.3.1 release of OracleAS Web Services is fully compatible with the 10.1.3 release.

Applications designed to run with versions 9.0.4 or 10.1.2 of the Oracle Application Server can be used with version 10.1.3.1. While you can still deploy your 9.0.4 or 10.1.2 Web Services in the 10.1.3.1 Application Server, you will not be able to see them in the management console.

Redeploying Applications on OracleAS Web Services 10.1.3.1

For backward compatibility, Oracle Application Server 10*g* Release 3 (10.1.3.1) includes the underlying software required to run 10*g* Release 2 (10.1.2) Web services. As a result, Web services applications designed to run with Oracle Application Server 10*g* (9.0.4) and 10*g* Release 2 (10.1.2) can be used without modification with Release 3.

However, there are significant advantages to re-creating your Web services for 10*g* Release 3 (10.1.3.1). For example, you can take advantage of the all the new features such as quality of service (QOS), a standard-based development model (JAX-RPC), and JMX-based management.

See Also:

Oracle Application Server Upgrade Guide provides more information on redeploying your existing Web service applications to OracleAS Web Services 10.1.3.1.

Deprecated Features

- The version 10.1.2 Web services stack is being deprecated. However, it is still supported in version 10.1.3.1.
- The `ws.debug` system property and its related behavior have been deprecated.
- The `oracle.webservices.ClientConstants.WSM_INTERCEPTOR_PIPELINE_CONFIG` property has been deprecated and replaced by `oracle.webservices.ClientConstants.CLIENT_CONFIG`. For more information on this property, see "How to Pass a Management Configuration to a Client Programmatically" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- The `oracle.webservices.OracleStub` properties `ENCODING_STYLE_PROPERTY` and `OPERATION_STYLE_PROPERTY` have been deprecated.

Clustered Environments and High Availability

To use OracleAS Web Services in a clustered environment, you must install the service on every machine in the cluster. Application Server Control enables you to easily deploy a service to a group of instances within an Oracle Application Server cluster.

If you install a standard, stateless Web service, then a clustered deployment will result in multiple instances that can process requests, but cannot share state.

If you install a stateful Web service, then those instances will share state. If the service is not installed on all machines in a cluster, then the cluster dispatcher might dispatch a service request to a machine that does not have the service, resulting in an error on the service invocation.

To support a clustered environment, for stateful Java Web services with serializable Java classes, the WebServicesAssembler adds a `<distributable>` tag in the `web.xml` of the Web service[1]s generated J2EE EAR file.

See Also:

- "Deploying to OC4J Instances Within a Cluster" in the *Oracle Containers for J2EE Deployment Guide* for more information on deploying to a group of instances.
- "Java Classes and Stateful Web Services" on page 7-9 for information on assembling stateful Web services.

OC4J in a Standalone Versus Oracle Application Server Environment

During development, it is typical to use OC4J by itself, outside an OracleAS Web Services environment. We refer to this as standalone OC4J (or, sometimes, as unmanaged OC4J). In this scenario, OC4J can use its own Web listener and is not managed by any external Oracle Application Server processes.

In contrast, a full Oracle Application Server environment (sometimes referred to as managed OC4J), includes the use of Oracle HTTP Server as the Web listener, and the Oracle Process Manager and Notification Server (OPMN) to manage the environment.

See Also:

- *Oracle Containers for J2EE Configuration and Administration Guide* for additional information about Oracle Application Server versus standalone environments and about the use of Oracle HTTP Server and OPMN with OC4J.
- *Oracle HTTP Server Administrator's Guide* for general information about the Oracle HTTP Server and the related `mod_OC4J` module. (Connection to the OC4J servlet container from Oracle HTTP Server is through this module.)
- *Oracle Process Manager and Notification Server Administrator's Guide* for general information about OPMN.

2

Development and Documentation Roadmap

The following sections provide a suggested roadmap through the documentation that takes you through the steps of developing a Web service.

- Setting Up Your Environment
- Best Coding Practices
- Assembling Web Service Artifacts
- Deploying the Web Service
- Testing the Deployed Web Service
- Assembling a Web Service Client
- Adding Quality of Service Features to a Web Service
- Adding Advanced Functionality to a Web Service
- Alternative Web Service Strategies
- Reference Chapters and Appendixes

Setting Up Your Environment

The following chapter describes how to set up your environment to use the functionality provided by OracleAS Web Services.

- Chapter 4, "Getting Started"

Best Coding Practices

The chapters in this section describe issues that you should consider as you design your Web service.

The following chapter describes the varieties of message formats that you can employ in OracleAS Web Services. It describes the advantages and disadvantages of each message format and suggests which format to use based on the client functionality.

- Chapter 5, "OracleAS Web Services Messages"

The following chapter in the *Oracle Application Server Advanced Web Services Developer's Guide* identifies some of the common areas where interoperability problems can occur. It provides design suggestions and programming guidelines that increase the interoperability of your Web service with applications on different platforms.

- "Ensuring Interoperable Web Services"

Assembling Web Service Artifacts

OracleAS Web Services enables you to assemble Web service artifacts bottom up starting from Java classes, EJBs, JMS destinations, database resources, or source files which employ J2SE 5.0 JDK Web Service Annotations. You can also assemble the artifacts top down starting from a WSDL. In OracleAS Web Services, you use the WebServicesAssembler tool to perform the assembly. Chapter 18, "Using WebServicesAssembler" provides a reference guide to the tool.

The following chapters describe how to use the WebServicesAssembler tool to perform the different types of Web Service assembly supported by OracleAS Web Services.

- Chapter 6, "Assembling a Web Service from WSDL"
- Chapter 7, "Assembling a Web Service with Java Classes"
- Chapter 8, "Assembling a Web Service with EJBs"
- Chapter 9, "Assembling Web Services with JMS Destinations"
- Chapter 10, "Assembling Database Web Services"
- Chapter 11, "Assembling Web Services with Annotations"

Deploying the Web Service

While the WebServicesAssembler tool does not perform deployment, it does package the Web service into a deployable EAR or WAR file. Deploying this file is very similar to deploying any other EAR or WAR file into a running instance of OC4J. OC4J provides a separate book that describes how to perform deployment.

- *Oracle Containers for J2EE Deployment Guide*

The following chapter provides additional information about the packaging format and the files required for deployment. The chapter also briefly describes the deployment support offered by the Oracle JDeveloper and Application Server Control tools.

- Chapter 19, "Packaging and Deploying Web Services"

Testing the Deployed Web Service

The following chapter describes the Web Service Test Page. This page lets you test whether deployment was successful without the need to write any code.

- Chapter 13, "Testing Web Services Deployment"

Assembling a Web Service Client

The following chapters describe how to use WebServicesAssembler to assemble a Web service client for the J2SE and J2EE platforms.

- Chapter 14, "Assembling a J2EE Web Service Client"
- Chapter 15, "Assembling a J2SE Web Service Client"

Adding Quality of Service Features to a Web Service

OracleAS Web Services support quality of service features, such as security, reliability, message logging, and auditing. The following chapters describe how to implement these features; they can be managed by other tools such as Oracle JDeveloper and Application Server Control. The following chapters appear in the *Oracle Application Server Advanced Web Services Developer's Guide*.

- "Managing Web Services"

- "Ensuring Web Services Security"

  This chapter provides only an overview of the contents of the *Oracle Application Server Web Services Security Guide*. The Security Guide describes the Web Services implementation of message-level security.
- "Ensuring Web Service Reliability"
- "Auditing and Logging Messages"

Adding Advanced Functionality to a Web Service

The following chapters describe additional features that can enhance the performance and functionality of your Web service.

- Chapter 16, "Using JAX-RPC Handlers"
- Chapter 17, "Processing SOAP Headers"

See also the following chapters in the *Oracle Application Server Advanced Web Services Developer's Guide*.

- "Working with Message Attachments"
- "Custom Serialization of Java Value Types"

Alternative Web Service Strategies

The following chapters describe alternative modes of Web service implementation.

For example, you can write your own infrastructure to make Web service calls, create a client for non-SOAP protocols, or use a non-HTTP transport mechanism.

- Chapter 12, "Assembling REST Web Services"

See also the following chapters in the *Oracle Application Server Advanced Web Services Developer's Guide*.

- "Using JMS as a Web Service Transport"
- "Using Web Service Invocation Framework"
- "Using Web Service Providers"

Reference Chapters and Appendixes

The following chapters and appendixes provide information to supplement the implementation and development tasks described in this book.

- Chapter 1, "Web Services Overview"
- Chapter 3, "OracleAS Web Services Architecture and Life Cycle"
- Chapter 18, "Using WebServicesAssembler"
- Appendix A, "Web Service Client APIs and JARs"
- Appendix B, "Oracle Implementation of the WSDL 1.1 API"
- Appendix C, "oracle-webservices.xml Deployment Descriptor Schema"
- Appendix D, "service-ref-mapping Schema"
- Appendix E, "Error Message Prefixes"

See also the following reference chapters and appendixes in the *Oracle Application Server Advanced Web Services Developer's Guide*.

- "Understanding the WSMGMT Schema"

- "OracleAS Web Services Client Schema"
- "OracleAS Web Services Reliability Schema"
- "OracleAS Web Services Auditing Schema"
- "OracleAS Web Services Tracing Schema"
- "JAX-RPC Mapping File Descriptor"
- "Web Service MBeans"
- "Mapping Java Types to XML and WSDL Types"

3

OracleAS Web Services Architecture and Life Cycle

This chapter provides an overview of the components that comprise Oracle Application Server Web Services and publishable service artifacts. These components are defined by the Java API for XML-Based RPC (JAX-RPC). This API enables Java technology developers to build Web applications and Web services incorporating XML-based RPC functionality according to the SOAP (Simple Object Access Protocol) 1.1 specification.

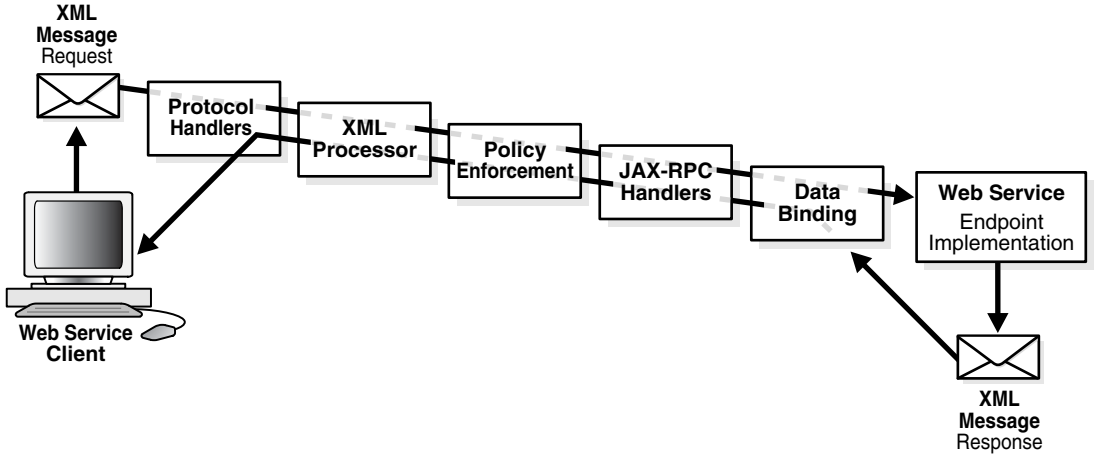
Architecture

The OracleAS Web Services stack is designed with three primary goals in mind: performance, interoperability, and manageability. This section describes how the Web services runtime is structured to provide enterprise-quality infrastructure for SOAP-based communication endpoints.

Processing Components

Each step in the processing of a Web services request is represented by a logical component in the runtime infrastructure. As an XML message is delivered to the system, it flows through the following layers before being delivered to an endpoint implementation: protocol handlers, XML processor, policy enforcement pipeline, JAX-RPC handlers, data binding, and the endpoint implementation. Response messages flow through the same infrastructure following a reverse path.

Figure 3–1 How XML Messages Flow From Client to Service



This section describes the purpose of each of these processing layers. Where appropriate, pointers to other sections are provided for more detailed information on system functionality and configuration.

Protocol Handlers

A protocol handler provides the entry point to the Web services infrastructure. The protocol handler is used to send and receive SOAP messages over a transport protocol. The Web services infrastructure can be configured to send and receive messages over HTTP or JMS.

If the messages are sent over HTTP, the OracleAS Web Services stack uses the Oracle HTTP Client libraries for sending Web services messages to services and the OC4J Servlet engine for receiving Web services messages sent by clients. All of the capabilities and management infrastructure of the Oracle servlet environment are available to Web services. For example, a Web service can be accessed with an encrypted data stream using HTTP or HTTPS.

JMS transport may be configured to work with different JMS providers. It is integrated into the application server by using the JMS infrastructure provided with the Oracle Application Server. JMS is often used to gain the full quality of service features of a message bus during a SOAP message exchange.

XML Processing

Once the SOAP message is retrieved from the transport layer, it is converted into an XML message representation that is compatible with the SOAP with Attachments API for Java (SAAJ). The SAAJ message is constructed using Oracle's optimized XML parsing technologies for performance and efficient memory utilization. This message is the basis for the JAX-RPC compliant SOAP processing infrastructure provided with OC4J. Once instantiated, the SAAJ message is delivered to the next layer of the processing stack.

Some portions of the SOAP request may not be XML. For example, a SOAP message may be sent with attachments, which are used to package non-XML content along with a SOAP message. The SAAJ implementation also deals with these attachments.

Though the default processing of XML messages assumes the message payload is encoded in a SOAP 1.1 or SOAP 1.2 compliant envelope, the OracleAS Web Services stack can also be configured to accept and dispatch XML messages over HTTP directly without using SOAP. This allows developers to create applications that integrate directly with existing HTTP infrastructure that is not aware of the SOAP protocol. Applications can be built to conform to the REST architecture style, using HTTP and URLs to define the messages that describe the system.

When configured to use XML-over-HTTP messaging, the infrastructure determines if a message contains an application message directly or a SOAP envelope as the top level element of the payload. If the message is "raw" XML, the processing layer will wrap the message in a SOAP envelope with no headers. This SOAP envelope can then be processed through the rest of the Web services processing elements and delivered to the endpoint implementation for processing.

The XML message is next processed by the policy enforcement mechanisms of the Web services stack.

Policy Enforcement

The OracleAS Web Services stack can be configured with additional information to enable a management chain that is responsible for enforcing runtime management policies. These policies include Web services management features like WS-Reliability,

WS-Security, auditing, and logging capabilities."Managing Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide* provides more information on the setup of these features. The *Oracle Application Server Web Services Security Guide* provides more information on setting up security. Enabled policies can be included in the WSDL document associated with a service to support automated configuration of client interceptor pipelines.

One of the protocols that the OracleAS Web Services stack supports is the WS-Reliability standard, which provides delivery guarantees for SOAP messages. The reliability infrastructure supports additional capabilities that allow the system infrastructure to send and receive asynchronous acknowledgment messages that conform to the WS-Reliability protocol. This is supported as an extension to the Oracle client infrastructure, which is available when using a JAX-RPC Stub or Call object.

The interceptor chain can also be configured to delegate to the OracleAS Web Services Management agent pipeline. This provides pre-integrated support for the OracleAS Web Services Management product and capabilities that support management of policies for Web services across a data center.

The management interceptors provide a runtime infrastructure for systems services that are provided in OC4J. Application-specific interceptors are supported in conformance with the JAX-RPC 1.1 Handler API.

JAX-RPC Handlers

Handlers are configured to process application-specific SOAP headers according to specific roles or actor attributes. The handlers have access to a SAAJ representation of the SOAP message and can perform operations on any level of the SOAP message.

Together, the interceptors and the handlers are used to enforce the SOAP processing model. This allows Web services endpoints to selectively process SOAP headers that are intended for a particular node. SOAP messages can then be passed along to other nodes in the system and SOAP headers are processed as required.

See Also:

Chapter 16, "Using JAX-RPC Handlers" and Chapter 17, "Processing SOAP Headers" for information on how to use the Handler API.

Data Binding

In many applications, portions of the XML payload are converted to Java objects that are used by the application framework. This capability is often called data binding, where portions of XML data are bound to members of a Java class hierarchy. The process of data binding is driven through the serialization framework, which manages the conversion from XML to Java. The serialization framework is extensible and allows developers to define custom type mapping from XML data to Java objects.

The OC4J runtime features a special "provider" implementation that is optimized for processing SAAJ messages. When the provider is used, no data binding is performed. Providers can be used to implement applications that work directly with XML payload in a SOAP message.

Endpoint Implementation

After passing through the preceding four layers of the Web services stack, the endpoint implementation containing the application business logic is invoked. The endpoint can be a regular Java class, an Enterprise Java Bean, or a provider. The endpoint can also be a JMS queue when a JMS endpoint configuration is enabled.

In the OracleAS Web Services container, there are very few requirements that a Java class must conform to in order to be exposed as a Web service. If the endpoint implementation requires more complex interaction with the container throughout its life cycle, it can implement a JAX-RPC `ServiceLifecycle` interface, which provides more information about the Web services requests to the endpoint during initialization of the service and while handling requests.

Java classes may also be augmented with Web services specific annotations. These annotations can be used to provide additional configuration information specifying what methods are exposed as Web services operations, what protocols can be used to access the service, and so on.

See Also:

Chapter 11, "Assembling Web Services with Annotations" for more information about OracleAS Web Services support for J2SE 5.0 JDK Web Service Annotations.

Java Management Extensions (JMX)

While the interceptor pipeline is used to enforce management policies, systems management capabilities are exposed in the OracleAS Web Services stack by using the Java Management Extensions (JMX) standard. JMX allows administrators to gather important metrics on the health of a running OC4J system and to change the configuration of a running system. JMX metrics and operations are available in the Web services console in the Application Server Control tool.

Development Tools

Another key feature of the OracleAS Web Services implementation is the development tools that allow for the development and deployment of endpoint implementations. Web services can be developed using either command line tools or a Java Integrated Development Environment (IDE).

WebServicesAssembler (WSA), the command line tool, is used to generate artifacts required to deploy a Web service from WSDL or endpoint implementation classes. It is useful for automating the creation of Web services in a scripted environment since, in addition to command line, it exposes its functionality as Ant tasks.

Oracle JDeveloper, Oracle's full-featured Java IDE, can be used for end-to-end development of Web services. Developers can build Java classes or EJBs, expose them as Web services, automatically deploy them to an instance of the Oracle Application Server, and immediately test the running Web service. Alternatively, Oracle JDeveloper can be used to drive the creation of Web services from WSDL descriptions. Like WebServicesAssembler, Oracle JDeveloper also is Ant-aware. You can use this tool to build and run Ant scripts for assembling the client and for assembling and deploying the service.

See Also:

Chapter 18, "Using WebServicesAssembler" for more information on how to use the webServicesAssembler tool to assemble web service artifacts.

Web Services Development Life Cycle

This section describes the stages of Web service development.

1. Create the Implementation

2. Assemble the Web Service
3. Assemble the Client
4. Deploy the Web Service
5. Test the Web Service
6. Perform Post Deployment Tasks

Create the Implementation

Create the implementation that you want to expose as a Web service. OracleAS Web Services allows a variety of artifacts to be exposed as a Web service, including:

- Java classes
- Enterprise Java Beans (EJBs)
- JMS queues or topics
- PL/SQL procedures
- SQL Statements
- Oracle Advanced Queues
- Java classes in the database
- CORBA servant objects

You can develop these artifacts using any tool or IDE. Oracle Oracle JDeveloper enables you to create Java classes, JMS queues and topics, PL/SQL procedures, CORBA servant objects, and EJBs.

Assemble the Web Service

OracleAS Web Services provides a variety of commands that let you assemble a Web service by using either a top down (starting with a WSDL) or a bottom up (starting with Java classes, EJBs, database artifacts, or JMS queues) approach. These commands can be issued on the command line or they can be written as tasks in an Ant script.

Since Oracle JDeveloper is fully Ant-aware, you can use this tool to build and run the Ant scripts. Oracle JDeveloper also has design-time wizards which symmetrically mirror several of the Web service generation commands. Using these wizards can speed your development process and save you the steps of creating the build scripts.

Most current Java IDEs (such as Eclipse) are also Ant-aware. You can use any of these IDEs to run the Ant scripts if you choose not to use Oracle JDeveloper.

If you need to invoke other OracleAS Web Services commands, for example, to generate a WSDL or to add quality of service features, you can invoke them on the command line or with Ant tasks. While there are no design-time wizards to support these commands, the Ant tasks can be run directly in Oracle JDeveloper.

See Also:

"Web Service Assembly Commands" on page 18-4 for more information on the commands that assemble Web services.

Assemble the Client

OracleAS Web Services provides commands that can be used to assemble J2SE and J2EE client code.

Oracle JDeveloper supports OC4J J2SE Web service clients by allowing developers to create Java stubs from Web service WSDL descriptions. These stubs can then be used to access existing Web services.

Deploy the Web Service

Web services deployment can be performed either with Java-language commands or with Ant tasks. The Ant tasks can reside in a build file or they can be issued directly from Oracle JDeveloper. Oracle JDeveloper also has a deployment wizard which configures the deployable EAR file and deploys it.

Deployment can also be performed with Application Server Control. However, this tool cannot be used to configure the EAR file or change its contents.

Test the Web Service

The OracleAS Web Services stack provides a Web Service Test Page for each deployed Web service. By entering a service endpoint address in a Web browser, you can access the operations that the Web service exposes. Interactive pages let you invoke the operations for values that you enter.

The WebServicesAssembler command line tool can generate tests suitable for the JUnit framework for every method in the Web service. If you use the WebServicesAssembler tool to generate JUnit tests for the `assemble`, `plsqlAssemble`, and `genProxy` commands, then Oracle JDeveloper can import them by default.

Perform Post Deployment Tasks

There are a number of tasks you can perform post-deployment. Some of these involve fine-tuning the performance of the Web service, such as changing security and logging policies. Others involve larger changes to the Web service, such as changing a key store configuration. You can make some of these changes dynamically; for others, you must redeploy the Web service. To perform these tasks, use Oracle Application Server Control. For more information on these tasks, see the Application Server Control on-line help.

4

Getting Started

The `README.txt` file that is included at the top level of the OC4J distribution provides instructions for setting up and running OC4J. This chapter serves as an addendum to the `README`. It provides information that is specific to setting up Oracle Application Server Web Services in your environment.

This chapter provides the following sections.

- Supported Platforms
- How to Install OC4J
- How to Set Up Your Environment for OracleAS Web Services
- How to Set Up Ant for WebServicesAssembler
- Database Requirements

Supported Platforms

OracleAS Web Services is supported on the following platforms.

- Redhat Linux
- Solaris Sparc 32- and 64-bit
- Microsoft Windows: Windows XP, Windows 2000, Windows Server 2003

How to Install OC4J

Follow the instructions in the `README.txt` file for installing and running OC4J. The `README.txt` file can be found at the top-level of the OC4J distribution.

How to Set Up Your Environment for OracleAS Web Services

This section lists the software you must install and environment variables you must define to use OracleAS Web Services.

- Java2 Standard Edition (J2SE) JDK version 1.4.1 or later
- Java2 Standard Edition (J2SE) JDK version 5.0 or later, if you are using the JDK 5.0 Web Service Annotations feature
- Jakarta Ant 1.5.x or 1.6.5 (recommended) is required if you will be using Ant tasks to assemble Web services. Jakarta Ant 1.6.5 is included in the OracleAS Web Services distribution.

  - If you want to have your own installation, you can obtain Ant from the following Web address.

    `http://jakarta.apache.org/ant/index.html`

  - If you are using your own installation of Ant, see "How to Set Up Ant for WebServicesAssembler" on page 4-3 for instructions on setting up Ant to assemble Web services.

  - If you installed the version of Ant that is distributed with Oracle Application Server, then Ant tasks for OracleAS Web Services will already be configured. Follow the instructions under "How to Set Up Ant 1.6.5 Distributed with Oracle Application Server" to appropriately modify your environment and build files.

- Define the following environment variables:

  - *ORACLE_HOME*—define this variable to point to the OC4J installation directory.

  - *JAVA_HOME*—define this variable to point to the J2SE SDK installation directory. OracleAS Web Services expects *JAVA_HOME*`/bin/java` to point to the Java VM and *JAVA_HOME*`/bin/javac` to point to the Java compiler executables.

- Add the following files or paths to the classpath:

  - To make assembly of Web services and the processing of XML files more convenient, add the paths to the WebServicesAssembler JAR, *ORACLE_HOME*`/webservices/lib/wsa.jar`, and the XML parser JAR, *ORACLE_HOME*`/lib/xmlparserv2.jar`, to the classpath.

  - If you want to assemble and compile your Web Services programs using Ant, appropriate `CLASSPATH` settings can be found in the Ant scripts accompanying the Web Services example code. If you want to invoke the WebServicesAssembler and compile generated code on the command line, you may want to add the full complement of libraries to your `CLASSPATH`.

    See "Sample Classpath Commands" on page A-6 for a sample Windows platform `set CLASSPATH` command for all of the OracleAS Web Services client JAR files. The classpath on the UNIX platform would be set in a similar manner.

  - OracleAS Web Services provides a separate library, `wsclient_extended.jar`, for running Web services clients in a J2SE environment. This library includes everything that would be required by OracleAS Web Services clients in a J2SE environment and should simplify the packaging and distribution of your J2SE client applications. You can find the `wsclient_extended.jar` file at the following address on the Oracle Technology Network Web site.

    `http://download.oracle.com/otn/java/oc4j/10131/wsclient_extended.zip`

    If you installed the OC4J companion CD, then the `wsclient_extended.jar` file can also be found in the *ORACLE_HOME*`/webservices/lib` directory.

    See "Simplifying the Classpath with wsclient_extended.jar" on page A-2 for more information on `wsclient_extended.jar` file.

- If you will be enabling reliable messaging between the client and server, you must run SQL scripts that will install tables for both the client and server.

"Installing SQL Tables for the Client and Server" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on how to find and run the SQL scripts.

- You will need an installed, running Oracle database if you are using reliable messaging or assembling database Web services.

  See "Database Requirements" on page 4-6 for more information on this topic.

How to Set Up Ant for WebServicesAssembler

The WebServicesAssembler tool assists in assembling OracleAS Web Services. It enables you to generate the artifacts required to develop and deploy Web services, regardless of whether you are creating the service using the top down or bottom up approach. WebServicesAssembler commands can be called either from the command line or from Ant tasks.

This section describes how to set up your environment and build script files to call WebServicesAssembler commands from Ant tasks.You can use an Ant installation you have previously installed or use the Ant that is found in *ORACLE_HOME*`/ant`. The following sections describe how to set up Ant, depending on the version you have installed.

Note: All of the Ant task examples in this book assume that you are using Ant version 1.6.5 or later. These versions let you use task namespaces. Hence, all of the Ant tags and subtags corresponding to WebServicesAssembler commands are prefixed with the `oracle:` namespace.

- How to Set Up Ant 1.6.5 Distributed with Oracle Application Server
- How to Set Up Ant 1.6.5 Using a Previous Installation of Ant
- How to Set Up Ant 1.5.2 Using a Previous Installation of Ant
- How to Set Up the "oracle:" Namespace Prefix for Ant Tasks

How to Set Up Ant 1.6.5 Distributed with Oracle Application Server

The following steps describe how to set up your environment and build files to use WebServicesAssembler with the Ant 1.6.5 installation found in *ORACLE_HOME*`/ant`. This is the version of Ant distributed with the Oracle Application Server.

1. Enter *ORACLE_HOME*`/ant/bin` at the front of your `PATH` variable.
2. Edit your build script (`build.xml`). Add the `antlib:oracle` namespace declaration for the imported Ant tasks. In the following example, *bottomup* is the name of your project.

   ```
   <project name="bottomup" default="all" basedir="."
   xmlns:oracle="antlib:oracle">
   ```

3. Add the `oracle:` namespace as a prefix to all WebServicesAssembler tags. For example:

   ```
   <oracle:assemble ....>
           <oracle:port ... />
   </oracle:assemble>
   <oracle:genProxy ..../>
   ```

4. (Optional) Copy the `ant-oracle.properties` files to the same directory as your build script.

   Although you can modify the properties file in the `j2ee/utilities` directory and reference it from your build scripts, it is better to maintain this file as a template.

5. (Optional) Edit the `ant-oracle.properties` file to reflect your installation environment.

6. (Optional) Edit the build script (`build.xml`). Reference the `ant-oracle.properties` file in the build script. For example:

   ```
   <property file="ant-oracle.properties"/>
   ```

7. (Optional) If you will be using the `junit` Ant task for reports, set the `ANT_OPTS` system property to the Xalan `TransformerFactoryImpl` class.

   - If you are using the JDK 1.5, set the `ANT_OPTS` property for `TransformerFactory` to `com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl`. For example:

     ```
     set ANT_
     OPTS=-Djavax.xml.transform.TransformerFactory=com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl
     ```

   - If you are using the JDK 1.4, set the `ANT_OPTS` property for `TransformerFactory` to `org.apache.xalan.processor.TransformerFactoryImpl`. For example:

     ```
     set ANT_
     OPTS=-Djavax.xml.transform.TransformerFactory=org.apache.xalan.processor.TransformerFactoryImpl
     ```

How to Set Up Ant 1.6.5 Using a Previous Installation of Ant

The following steps describe how to set up your environment and build files to use WebServicesAssembler with a previous installation of Ant 1.6.5 (or later).

1. Navigate to the directory *`ORACLE_HOME`*`/j2ee/utilities` and ensure that the following files are present:

   - `ant-oracle.properties`—this file enables you to designate the key properties for the execution of the Oracle Ant tasks.
   - `ant-oracle.xml`—this file enables you to use Oracle Ant tasks.

2. Copy the `ant-oracle.properties` and `ant-oracle.xml` files to the same directory as your build script (`build.xml`).

   Although you can modify the files in the `j2ee/utilities` directory and reference them from your build scripts, it is better to maintain the source files as templates. Also, if you leave the `ant-oracle.xml` file in its original location, then the import reference must be hard coded to specify the full path to the file (for example, `c:/oc4j/j2ee/utilities/ant-oracle.xml`).

3. Edit the `ant-oracle.properties` file to reflect your installation environment.

4. Edit the build script (`build.xml`).

   - Import the `ant-oracle.xml` file into the build script.

```
<!-- Import for OC4J ant integration. -->
<import file="ant-oracle.xml"/>
```

- Add the `antlib:oracle` namespace reference for the imported Ant tasks. In the following example, *`bottomup`* is the name of your project.

```
<project name="bottomup" default="all" basedir="."
xmlns:oracle="antlib:oracle">
```

5. Include the `oracle` namespace as a prefix to all WebServicesAssembler commands. For example:

```
<oracle:deploy ..../>
<oracle:genProxy ..../>
```

6. (Optional) If you will be using the `junit` Ant task for reports, set the `ANT_OPTS` system property to the Xalan `TransformerFactoryImpl` class.

   - If you are using the JDK 1.5, set the `ANT_OPTS` property for `TransformerFactory` to `com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl`. For example:

```
set ANT_
OPTS=-Djavax.xml.transform.TransformerFactory=com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl
```

   - If you are using the JDK 1.4, set the `ANT_OPTS` property for `TransformerFactory` to `org.apache.xalan.processor.TransformerFactoryImpl`. For example:

```
set ANT_
OPTS=-Djavax.xml.transform.TransformerFactory=org.apache.xalan.processor.TransformerFactoryImpl
```

How to Set Up Ant 1.5.2 Using a Previous Installation of Ant

The following steps describe how to set up your environment and build files to use WebServicesAssembler with a previous installation of Ant 1.5.2.

1. Ensure that your installations of Ant and the Java JDK are already included in the classpath environment variable.
2. Add the path to `wsa.jar` to the CLASSPATH environment variable. The path will typically be:

   `OC4J_Home``/webservices/lib/wsa.jar`

   In this example, *`OC4J_Home`* is the directory where you installed OC4J.
3. Add the following lines to any Ant build file that will invoke the Ant tasks.

```
<taskdef resource="orawsa.tasks" />
<typedef resource="orawsa.types" />
```

   These lines can appear anywhere in the build file before the first WebServicesAssembler task is called.
4. (Optional) If you will be using the `junit` Ant task for reports, set the `ANT_OPTS` system property to the Xalan `TransformerFactoryImpl` class.

- If you are using the JDK 1.5, set the `ANT_OPTS` property for `TransformerFactory` to `com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl`. For example:

```
set ANT_
OPTS=-Djavax.xml.transform.TransformerFactory=com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl
```

- If you are using the JDK 1.4, set the `ANT_OPTS` property for `TransformerFactory` to `org.apache.xalan.processor.TransformerFactoryImpl`. For example:

```
set ANT_
OPTS=-Djavax.xml.transform.TransformerFactory=org.apache.xalan.processor.TransformerFactoryImpl
```

How to Set Up the "oracle:" Namespace Prefix for Ant Tasks

Ant version 1.6.5 and higher requires the `oracle:` prefix at the beginning of all Ant tags and subtags for WebServicesAssembler commands. This prefix informs the Ant interpreter that this is an Oracle Ant task.

The `oracle:` prefix corresponds to the prefix that is found in the `project` tag at the beginning of the Ant `build.xml` file.

```
<project name="myproject" default="all" basedir="." xmlns:oracle="antlib:oracle">
```

If you do not want to use `oracle` as a prefix, you can change it to any valid XML `QName` prefix. For example, if you want to use `oracletags` as a prefix, then you must change the value of the `project` tag.

```
<project name="myproject" default="all" basedir="."
xmlns:oracletags="antlib:oracle">
```

If you make this change, then all Ant tags and subtags for WebServicesAssembler commands must start with `oracletags:`. For example:

```
<oracletags:assemble ...>
```

Database Requirements

You will need an installed running Oracle database (local or remote) if you will be performing any of the following tasks:

- assembling any of the database Web Services. Chapter 10, "Assembling Database Web Services" provides more information on the database Web services that can be run on OracleAS Web Services.
- enabling reliable messaging between the client and server. "Ensuring Web Service Reliability" in the *Oracle Application Server Advanced Web Services Developer's Guide* provides more information on reliable messaging.

5

OracleAS Web Services Messages

Oracle Application Server Web Services supports SOAP 1.1 and 1.2 messages. The format of the messages can be document or RPC style and literal or encoded use.

This chapter contains these sections:

- OracleAS Web Services Message Formats
- Working with SOAP Messages
- How to Convert XML Elements to SOAP Elements

OracleAS Web Services Message Formats

This section describes the message formats supported by the current release of OracleAS Web Services. It includes these topics:

- Understanding Message Formats
- Supported Message Formats
- How to Select Message Formats for a Web Service
- Changing Message Formats in a Service Implementation
- Message Format Recommendations

To understand the message formats supported by OracleAS Web Services, it is useful to understand the relationship between the Web Service Description Language (WSDL) 1.1 and the wire format. The wire format is the physical representation of a Simple Object Access Protocol (SOAP) message, or payload, for transmission. The message format is determined by the use and style attributes from the binding defined in the WSDL. The type of XML schema that defines the message part enhances the message format. The WSDL, then, can be thought of as a contract. By defining the various attributes in the WSDL, you affect the format of the message on the wire.

Any interoperability issues that arise are usually noticed at runtime in the wire format. Often, you can fix these by adjusting the WSDL and regenerating the Web service artifacts.

The relationship between the message format and the wire format is not one-to-one. For example, you can define a document-literal style Web service, Service A, with an XML schema that makes runtime SOAP messages look identical to messages produced by an RPC-literal style Web service, Service B. If you change the style and use (that is, the message format) of Service A to be "RPC" and "literal", then Service A will not be the same as Service B. The runtime SOAP messages would look completely different after the change, unless you also change the schema used in Service A.

Understanding Message Formats

The following sections briefly describe the message formats supported by OracleAS Web Services.

> **See Also:**
>
> The SOAP and WSDL specifications listed in "SOAP-Related Documents" on page xxiii and "WSDL-Related Documents" on page xxiv describe the full implications of using the `use` and `style` WSDL binding attributes.

RPC and Document Styles

A SOAP payload can be either RPC or document style. An RPC-style payload is usually used if there is a need to invoke a remote procedure or method call. With RPC style, the name of the top-most XML element in the SOAP body of the request is always the name of the WSDL operation. There is no ambiguity, because the names are unique in a given binding. The SOAP XML message typically consists of a method name and parameters, all represented in XML.

If the WSDL operation is overloaded, there must be a unique `SOAPAction` specified in the corresponding operation binding. Section 7 of the SOAP 1.1 specification describes the structure of an RPC-style SOAP body element (`<body>`).

The SOAP body of a document-style payload contains XML that does not have to conform to Section 7 of the SOAP 1.1 specification, but it uses an XML schema global element to define the payload of the message. That schema is defined within or imported into the WSDL's `type` section.

Literal and Encoded Uses

The SOAP client and server interprets the XML contents of the SOAP payload `<body>` element according to the rules specified by the `use` attribute of the WSDL's `binding` section. The client and server must agree on the same encoding rule to ensure that they can each correctly interpret the data.

For a literal use, the rules for encoding and interpreting the SOAP body of input and output messages are described entirely in terms of the schema.

For the encoded use, the `encodingStyle` attribute in the SOAP body identifies the rules for encoding and interpreting the message according to the SOAP specification:

- For SOAP 1.1, see "SOAP Encoding", Section 5 of the SOAP 1.1 specification: `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383512`
- For SOAP 1.2, see "SOAP Encoding", Section 3 of the SOAP 1.2 specification: `http://www.w3.org/TR/2003/REC-soap12-part2-20030624/#soapenc`

You can also use types defined in the SOAP encoding schema extension.

Supported Message Formats

The following sections describe the message formats supported by OracleAS Web Services.

- Document-Literal Message Format
- RPC-Encoded Message Format
- RPC-Literal Message Format

Note: Multiple message formats, such as RPC-encoded and document-literal, are not supported in a single Web application.

Document-Literal Message Format

Document-literal is the default message format for OracleAS Web Services. The two common styles of document-literal operations are wrapped and bare.

- For a wrapped style, a schema definition of a wrapper element wraps the parameters belonging to a method. The messages are not SOAP encoded and do not use SOAP RPC conventions.
- For a bare style, the method must have only one parameter mapped to a SOAP body. If the method has multiple parameters, then only one can be mapped to the body part. The other parameters must be mapped to SOAP headers.

Document-literal complies with WS-I Basic Profile 1.0 and 1.1.

Each document-literal operation is uniquely identified from the `QName` of the top element of the input message. Document-literal with the wrapped style has the best interoperability with .NET Web services and is the preferred default message format for OracleAS Web Services.

See Also:

"Ensuring Interoperable Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide* for information on interoperability and message formats.

Advantages:

- Complies with WS-I.
- Provides support for interoperability.
- Natural format for SOAP fault handling.
- Format of the message part is described by the standard XML schema.

Disadvantages:

- Not fully backward compatible with older stacks.
- Does not support object graphs.

Sample Request Message with the Document-Literal Message Format Example 5–1 illustrates a request message in the document-literal message format. Note that the XML element part (`payloadDocument`) under the SOAP body (`env:Body`) must be a document instance of a global element defined in the WSDL's schema.

The example applies to SOAP 1.1 messages. To change the example to apply to SOAP 1.2 messages, change the value of `xmlns:env` to `http://www.w3.org/2003/05/soap-envelope`.

Example 5–1 Request Message with the Document-Literal Message Format

```
<env:Envelope
        xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:ns0="http://ws.oracle.com/doc-lit">
        <env:Body>
                <ns0:payloadDocument>
                        <ns0:name>Scott</ns0:name>
```

```
                        <ns0:data>Hello</ns0:data>
                    </ns0:payloadDocument>
            </env:Body>
</env:Envelope>
```

RPC-Encoded Message Format

The RPC-encoded message format uses the encoding rules defined in the SOAP specification:

- For SOAP 1.1, see "SOAP Encoding", Section 5 of the SOAP 1.1 specification.

  `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383512`

  The `encodingStyle` attribute for the SOAP 1.1 RPC-encoded message format is expressed with the following value.

  ```
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  ```

- For SOAP 1.2, see "SOAP Encoding", Section 3 of the SOAP 1.2 specification.

  `http://www.w3.org/TR/2003/REC-soap12-part2-20030624/#soapenc`

  The `encodingStyle` attribute for the SOAP 1.2 RPC-encoded message format is expressed with the following value.

  ```
  encodingStyle="http://www.w3.org/2003/05/soap-encoding"
  ```

  See Also:

  "Ensuring Interoperable Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide* for information about the issues of interoperability and message formats.

Advantages:

- SOAP 1.1 encoding is backward compatible with older stacks, because this was one of the most common message formats for Web services.
- Supports object graphs (through `id` and `href` attributes).
- Provides additional Java type mapping support indicated by the namespace `http://www.oracle.com/webservices/internal`. For a list of supported data types, see Table 5–1 on page 5-7.

Disadvantages:

- Does not comply with the WS-I Basic Profile.
- More difficult to perform schema validation of the entire message payload.

Sample Messages with the RPC-Encoded Message Format Example 5–2 illustrates a request message that uses the RPC-encoded message format. Note that the tag name of the XML element part (`echoString`) sent under the SOAP body (`env:Body`) must be the same as the name of the corresponding WSDL operation. The `env:encodingStyle` attribute indicates the SOAP encoding style being used. Each XML element part (`stringParam`) under the operation element corresponds to a parameter. It must be an instance of `simpleType` or a global type definition. If it is a global type definition, it must be in the WSDL's schema or one of the SOAP encoding extension types.

RPC-encoded request messages to OracleAS Web Services (or RPC-encoded response messages to OracleAS Web Services-generated stubs) can be consumed without `xsi:type` attributes.

Example 5–2 and Example 5–3 apply to SOAP 1.1 messages. To change the examples to apply to SOAP 1.2 messages:

- Change the value of `xmlns:env` to `http://www.w3.org/2003/05/soap-envelope`
- Change the value of `env:encodingStyle` to `http://www.w3.org/2003/05/soap-encoding`

Example 5–2 RPC-Encoded Request Message

```
<env:Envelope
        xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:ns0="http://ws.oracle.com/rpc-enc">
        <env:Body>
                <ns0:echoString

env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
                        <stringParam>Hello</stringParam>
                </ns0:echoString>
        </env:Body>
</env:Envelope>
```

Example 5–3 illustrates a response message that uses the RPC-encoded message format. RPC-encoded response messages from OracleAS Web Services (or the RPC-encoded request messages from an OracleAS Web Services-generated stub) always contain `xsi:type` attributes.

Example 5–3 RPC-Encoded Response 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://ws.oracle.com/rpc-enc">
        <env:Body>
                <ns0:echoStringResponse

env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
                        <stringParam xsi:type="xsd:string">Hello</stringParam>
                </ns0:echoStringResponse>
        </env:Body>
</env:Envelope>
```

The xsi:type Attribute in RPC-Encoded Message Formats In many SOAP implementations, messages that use the RPC-encoded message format usually use the `xsi:type` attribute on each element in the message payload. This attribute helps object serialization and deserialization. The `xsi:type` attribute is optional in most cases. The `xsi:type` attribute is required only if the element is an instance of a derived type of an element type defined in the schema. For inbound SOAP messages, OracleAS Web Services accepts messages with or without the `xsi:type` attribute. For outbound SOAP messages in RPC-Encoded format, OracleAS Web Services always emits the `xsi:type` attribute.

Example 5–4 and Example 5–5 apply to SOAP 1.1 messages. To change the examples to apply to SOAP 1.2 messages:

- Change the value of `xmlns:env` to `http://www.w3.org/2003/05/soap-envelope`

- Change the value of `env:encodingStyle` to `http://www.w3.org/2003/05/soap-encoding`

Example 5–4 illustrates a request message for the `echo` operation in RPC-encoded message format. Note that the code sample does not contain the `xsi:type="xsd:string"` attribute.

Example 5–4 RPC-Encoded Request Message Without the xsi:type Attribute

```
<env:Envelope
        xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:ns0="http://ws.oracle.com/rpc-enc">
   <env:Body>
      <ns0:echo env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
         <param>some string</param>
      </ns0:echo>
   </env:Body>
</env:Envelope>
```

Example 5–5 illustrates the same request message in RPC-encoded format with the `xsi:type` attribute.

Example 5–5 RPC-Encoded Request Message With the xsi:type Attribute

```
<env:Envelope
        xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:ns0="http://ws.oracle.com/rpc-enc">
   <env:Body>
      <ns0:echo env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
         <param xsi:type="xsd:string">some string</param>
      </ns0:echo>
   </env:Body>
</env:Envelope>
```

Oracle Specific Type Support For a complete list of the supported types, see Chapters 4 and 5 of the JAX-RPC 1.1 specification. The JAX-RPC 1.1 specification is available from:

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

For completeness, Table 5–1 describes all of the Java types and Oracle proprietary types supported by the OracleAS Web Services RPC-encoded message format. The Java type mapping support for the types is indicated by the following OracleAS Web Services-specific namespace:

`http://www.oracle.com/webservices/internal`

This namespace accommodates nonstandard XML schema definitions for some standard Java types, such as `Collection`, that are not supported by JAX-RPC 1.1.

> **Note:** The `java.util.Collection` family and `java.util.Map` family are also supported for the literal use (that is, document-literal and RPC-literal). The schema types for these Java types are defined under a different namespace. See "Mapping Java Types to XML and WSDL Types" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on how these Java types are supported.

Table 5–1 Java Types Supported by RPC-Encoded Proprietary Message Format

Java Type	Java Classes	Mapping Details
Java primitive types	boolean, byte, double, float, int, long, short	See "How OracleAS Web Services Maps Java Primitive Types to XML Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*
java.lang Object types	Boolean, Byte, Double, Float, Integer, Long, Short, String	See "How OracleAS Web Services Maps Java Types to XML Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*
basic Java types	java.math.BigDecimal java.math.BigInteger java.xml.QName java.util.Calendar java.util.Date java.net.URI	See "How OracleAS Web Services Maps Java Types to XML Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*
Java array types	One-dimensional arrays with elements of supported type	See "How OracleAS Web Services Maps Java Types to XML Types" and "How OracleAS Web Services Maps Java Collection Classes to XML Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*
Java value types	Java Beans with properties of supported types	See "How OracleAS Web Services Supports Java Value Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*
attachments **Note:** attachments are not interoperable	java.awt.Image javax.mail.internet.MimeMultipart javax.xml.transform.Source javax.activation.DataHandler	See "Working with Message Attachments" in the *Oracle Application Server Advanced Web Services Developer's Guide*
java.util.Collection	java.util.Collection java.util.List java.util.Set java.util.Vector java.util.Stack java.util.LinkedList java.util.ArrayList java.util.HashSet java.util.TreeSet	See "How OracleAS Web Services Maps Java Collection Classes to XML Types" and "Additional Information about Oracle Proprietary XML Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*

Table 5–1 (Cont.) Java Types Supported by RPC-Encoded Proprietary Message Format

Java Type	Java Classes	Mapping Details
java.util.Map	java.util.Map java.util.HashMap java.util.TreeMap java.util.Hashtable java.util.Properties	See "How OracleAS Web Services Maps Java Collection Classes to XML Types" and "Additional Information about Oracle Proprietary XML Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*

If you want to use any value types which are not built-in (for example, a user-defined `MyBean` type) as items in a `Map` or `Collection`, then you must use the value of the `valueTypeClassName` argument to declare these types to the WebServicesAssembler tool when generating the WSDL.

```
java -jar wsa.jar -genWsdl
                  -valueTypeClassName hello.MyBean
                  -valueTypeClassName hello.MyFoo...
```

In this command:

- `genWsdl`—Generates a WSDL based on a Java interface. See "genWsdl" on page 18-28.
- `valueTypeClassName`—Specifies the fully-qualified class name of the JAX-RPC value type which is used for `java.util.Collection` and `java.util.Map`. See "valueTypeClassName" on page 18-69.

This allows the generated WSDL to include the schema definitions for these value types. The runtime can then correctly generate the corresponding serialized values. All WebServicesAssembler commands and Ant tasks that assemble Web services bottom up (from Java classes, EJBs, database resources, and so on) support the `valueTypeClassName` argument.

See Also:

"Custom Serialization of Java Value Types" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on using value type classes that do not conform to the JAX-RPC value type requirements, or that cannot be handled by the default JAX-RPC serialization mechanism.

Restrictions on RPC-Encoded Format OracleAS Web Services does not support the combination of RPC-encoded message formats and `databinding=false`.

RPC-Literal Message Format

RPC-literal message format complies with WS-I Basic Profile 1.0 and 1.1. This format uses the RPC style of message payload structure but supports the literal way of describing the types passed by a procedure. In this case, literal means that there exists a schema for every parameter type but not for the payload of the message body itself.

See Also:

"Ensuring Interoperable Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide* for information on interoperability and message formats.

Advantages:

- Complies with WS-I.
- Format of the message part is described by the standard XML schema.

Disadvantages:

- Support for RPC-literal is not consistent across all vendors.
- Does not support object graphs.
- Is not backward compatible with older stacks.
- It is not possible to represent null values for Java method parameters when mapping to WSDL message parts.

Sample Request Message with the RPC-Literal Message Format Example 5–6 illustrates a request message coded for the RPC-literal message format. Note that the tag name of the XML element part (`echoString`) under the SOAP body (`env:Body`) must be identical to the name of the corresponding WSDL operation. Each XML element part (`stringParam`) under the operation element corresponds to a parameter and must be an instance of `simpleType` or a global type definition. If it is a global type definition, then it must be in the WSDL's schema.

The example applies to SOAP 1.1 messages. For SOAP 1.2 messages, change the value of `xmlns:env` to `http://www.w3.org/2003/05/soap-envelope`.

Example 5–6 RPC-Literal Request Message

```
<env:Envelope
        xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:ns0="http://ws.oracle.com/rpc-lit">
        <env:Body>
                <ns0:echoString>
                        <stringParam>Hello</stringParam>
                </ns0:echoString>
        </env:Body>
</env:Envelope>
```

How to Select Message Formats for a Web Service

The WebServicesAssembler tool provides arguments that let you control the message format used by a Web service. These arguments let you specify whether the message format is RPC or document (wrapped or bare), encoded or literal.

- style
- use

The following WebServicesAssembler commands allow you to use the `use` and `style` arguments to specify the message format for your Web service. You can find more information on these commands in Chapter 18, "Using WebServicesAssembler".

- aqAssemble
- assemble
- corbaAssemble
- dbJavaAssemble
- ejbAssemble
- genWsdl

- plsqlAssemble
- sqlAssemble

The following example uses the `assemble` command to assemble the Web service server components. The `style` and `use` arguments specify that the message format used is RPC-literal:

```
java -jar wsa.jar -assemble
                  -appName $(app.name)
                  -serviceName HelloServiceWSIF
                  -uri $(app.name)
                  -interfaceName oracle.demo.hello.HelloInterface
                  -className oracle.demo.helllo.HelloImpl
                  -input $(service.classes.dir)
                  -output build
                  -ear dist/$(app.name).ear
                  -style rpc
                  -use literal
```

In this command:

- `assemble`—Generates a Web service from Java files bottom up. The command creates all of the files required to create a deployable archive. See "assemble" on page 18-7.
- `appName`—Specifies the name of an application. Usually, this name is used as a base value for other arguments like `context` and `uri`. See "appName" on page 18-42.
- `serviceName`—Specifies the service name. See "serviceName" on page 18-50.
- `uri`—Specifies the URI to use for the Web service. See "uri" on page 18-63.
- `interfaceName`—Specifies the name of a Java class (including the package name) that contains the service endpoint interface (SEI). See "interfaceName" on page 18-47.
- `classname`—Specifies the name of the class (including the package name) that is the implementation class for the Web service. See "className" on page 18-43.
- `input`—Specifies the directory or JAR containing the classes that should be copied to `WEB-INF/classes`. This argument will be added to the classpath used by the WebServicesAssembler. See "input" on page 18-46
- `output`—Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See "output" on page 18-48.
- `ear`—Specifies the name and location of the generated EAR. See "ear" on page 18-44.
- `style`—For bottom up Web service assembly, this argument specifies the `style` attribute of the message format in the generated WSDL. See "style" on page 18-68.
- `use`—For bottom up Web service assembly, this argument specifies the `use` attribute of the message format in the generated WSDL. See "use" on page 18-68.

See Also:

Chapter 18, "Using WebServicesAssembler" for more information on the WebServicesAssembler tool and its functionality.

Changing Message Formats in a Service Implementation

You can expose Java service endpoint implementations in RPC-encoded, RPC-literal, or document-literal format. However, if the service endpoint implementation uses the `attachment` data types listed in the Table 5–1, then you can use only RPC-encoded. If none of the `attachment` types are used, then you can use RPC-literal or document-literal format to achieve better interoperability. Each of the formats display similar performance in marshaling between Java objects and XML.

Message Format Recommendations

This section provides some general guidelines for choosing a message format when designing your Web service. When choosing a message format, consider the functionality requirements of the client you want to support. Table 5–2 describes some possible client functionality requirements and suggested message formats.

Table 5–2 Message Format Suggestions, Based on Client Functionality

Client Functionality	Suggested Message Format
Axis 1.2 client	Use either RPC-encoded or document-literal message formats
BPEL processes	Use document style with the literal use.
call-out from an Oracle release 10.1 database	JAX-RPC supports all formats. You may want to avoid RPC-literal database call-out if you need to represent `NULL` values.
call-out from an Oracle release 9.2 database	Callout is based on `soap.jar`; use RPC-encoded.
expose graph and preserve object identity	Use RPC-encoded. Although RPC-encoded is the easiest way, it can be achieved in document-literal if the model (schema) is well-designed.
OmniPortlet	Use either RPC-encoded or document-literal. The Omniportlet APIs (wizard) does not support RPC-literal.
Oracle XML Query Service (XQS) integration	Use either RPC or document style with the literal use. Oracle XML Query Service does not support SOAP encoding.
WS-I compliance	Use either RPC or document style with the literal use.
XML Schema integration	Reuse the schema definition to describe the SOAP message part of the WSDL operation. This integration can be achieved with either RPC or document style with the literal use.

Working with SOAP Messages

OracleAS Web Services supports SOAP 1.1 and 1.2 messages both programmatically and by using the WebServicesAssembler tool to assemble Web services bottom up and top down.

- Understanding the OraSAAJ APIs
- How to Use the OraSAAJ APIs to Manipulate SOAP Messages
- How to Convert XML Elements to SOAP Elements
- How to Use SOAP 1.2 Formatted Messages in Bottom Up Web Service Assembly
- How to Use SOAP 1.2 Formatted Messages in Top Down Web Service Assembly

The SOAP with Attachments API for Java (SAAJ) version 1.2 is the programmatic model for representing and working with a SOAP message. The standard SAAJ APIs support SOAP 1.1. You can find more information on SAAJ at the following Web address:

http://java.sun.com/webservices/saaj/index.jsp

The Oracle extension of the SAAJ 1.2 API (OraSAAJ) allows a Web service to work with SOAP 1.2 messages.

Understanding the OraSAAJ APIs

OracleAS Web Services support SAAJ 1.2, which is a specification for modeling SOAP messages with attachments in Java objects. However, the standard SAAJ 1.2 APIs support only SOAP 1.1 messages. To provide programmatic support for SOAP 1.2 messages, OracleAS Web Services includes the `oracle.webservices.soap` package. The interfaces in this package allow you to work with and add information to SOAP 1.2 message objects.

The classes in this package, `VersionedMessageFactory` and `VersionedSOAPFactory`, are extensions to standard SAAJ classes `MessageFactory` and `SOAPFactory`. The methods in the `VersionedMessageFactory` and `VersionedSOAPFactory` classes contain an extra parameter that lets you specify a SOAP message version when using the standard SAAJ APIs.

Table 5–3 Interfaces and Classes in the OraSAAJ API

Interface/Class Name	Description
Body12 interface	Represents a SOAP 1.2 message `Body` object.
Fault12 interface	Provides methods to add SOAP 1.2 `FaultCode` and `FaultReason` elements to a SOAP 1.2 `Fault` element.
FaultCode12 interface	Provides methods to add SOAP 1.2 `FaultValue` and `FaultSubcode` elements to a SOAP 1.2 `FaultCode` element.
FaultReason12 interface	Provides methods to add a SOAP 1.2 `FaultText` element to a SOAP 1.2 `FaultReason` element.
FaultSubcode12 interface	This interface is an extension to the `FaultCode12` interface. It is a marker interface for compiler-enforced strong typing.
FaultText12 interface	Provides methods for adding the text node content and locale information to a `FaultText` element.
FaultValue12 interface	Provides methods to set the fault code on a `FaultValue12` element.
OracleSOAPElement interface	An extension of the `java.xml.soap.SOAPElement` interface. Provides methods for working with the data handlers for messages with attachments.
SOAPVersion interface	Provides constants representing the SOAP versions that are available to the platform.
VersionedMessageFactory class	Provides methods for creating a SOAP message. These methods mimic the standard SAAJ `MessageFactory` class, except that they contain an extra parameter for specifying a SOAP message version. One method creates an empty message with standard MIME headers. The other method creates a message based on an input stream and a specified MIME header.

Table 5–3 (Cont.) Interfaces and Classes in the OraSAAJ API

Interface/Class Name	Description
VersionedSOAPFactory class	Provides methods for creating SOAP elements. These methods mimic the standard SAAJ `SOAPFactory` class, except that they contain an extra parameter for specifying a SOAP message version.

See Also:

The *Oracle Application Server Web Services Java API Reference* for more information on the classes and methods in the `oracle.webservices.soap` package.

How to Use the OraSAAJ APIs to Manipulate SOAP Messages

The OraSAAJ extensions can be used from a `javax.xml.rpc.handler.Handler`. In most cases, the standard `javax.xml.soap.*` classes can be used to manipulate SOAP 1.2 SAAJ messages. However, if you want to use the functionality provided by SOAP 1.2, you must use the OraSAAJ APIs.

Example 5–7 illustrates how the standard `javax.xml.soap.*` classes and the OraSAAJ classes can be used together. The example code creates a SOAP 1.2 message from scratch. The `VersionedMessageFactory` method returns a `javax.xml.soap.SoapMessage` object. This enables you to use the standard `javax.xml.soap.*` methods such as `getBody` and `getEnvelope` on the message.

The `addFault` method adds a SOAP 1.2 fault to the message. To send SOAP 1.2 faults, you must use the OraSAAJ `Fault12`, `FaultCode12`, `FaultValue12`, and `FaultReason12` APIs. This is because SOAP 1.2 faults contain more information than SOAP 1.1 faults.

Example 5–7 Working with the SAAJ and OraSAAJ APIs

```
public boolean handleResponse(MessageContext context) {
    ...
    // create a SOAP 1.2 message from scratch.
    // Note the use of VersionedMessageFactory to get a SOAPMessage
    // for a specific version of soap
    SOAPMessage message =
((VersionedMessageFactory)MessageFactory.newInstance()).createVersionedMessage(ora
cle.webservices.soap.SOAPVersion.SOAP_1_2);
    // Now standard APIs can be used.
    SOAPBody body = message.getSOAPPart().getEnvelope().getBody();
    // However, if you need to send a fault, you must
    // use Oracle specific APIs, because SOAP 1.2
    // faults contain more information than SOAP 1.1 faults.
    // Note the use of Fault12, FaultCode12, and FaultReason12
    SOAPFault fault = body.addFault();
      Fault12 soapFault = (Fault12) fault;
      FaultCode12 faultCode = soapFault.addCode();
      FaultValue12 faultValue = faultCode.addFaultValue();
      QNameAdapter faultCodeQName = new QNameAdapter("http://my.foo.com/",
                                                     "myFaultCode",
                                                     "foo");
      faultValue.setFaultCode(faultCodeQName);
      FaultReason12 faultReason = soapFault.addReason();
      faultReason.addFaultText().setValue("An unknown error occurred");
    ...
```

```
}
```

How to Work with Messages with Attachments Containing Binary Data

The `oracle.webservices` package provides an `OracleSOAPElement` interface that contains methods that enable you to work with messages with attachments.

See Also:

"Working with Attachments Containing Binary Data" in the *Oracle Application Server Advanced Web Services Developer's Guide* provides more information on the `OracleSOAPElement` interface and how to use its methods to get and set attachments with binary data.

How to Convert XML Elements to SOAP Elements

The `oracle.webservices` package provides a `toSOAPElement` method in the `SOAPUtil` class to convert XML elements (`org.w3c.dom.Element`) to SOAP elements (`javax.xml.soap.SOAPElement`). Note that `SOAPElement` implements `dom.Element`, and not vice versa.

Example 5–8 illustrates a code sample that creates an XML document, converts it to a SOAP element, and prints it to the standard output.

Example 5–8 Converting an XML Element to a SOAP Element

```
...
try {
   DOMParser parser = new DOMParser();
   parser.parse(new StringReader(
      "<Flds:CustomerGroup xmlns:Flds=\"http://foo.com/foo.xsd\">
      "<Flds:Customer>xyz</Flds:Customer>
      "</Flds:CustomerGroup>"));

   SOAPElement se = SOAPUtil.toSOAPElement(
      parser.getDocument().getDocumentElement());
   ((XMLElement)se).print(System.out);
} catch (Exception ex) {
   ex.printStackTrace();
}
...
```

How to Use SOAP 1.2 Formatted Messages in Bottom Up Web Service Assembly

To support SOAP version 1.2 messages in bottom up Web service generation, WebServicesAssembler provides a `soapVersion` argument. Values can be "`1.1`", "`1.2`", or "`1.1,1.2`". Default value is "`1.1`".

The "`1.1,1.2`" value means that WebServicesAssembler will create two ports with two bindings. One port and binding will support version 1.1; the other port and binding will support version 1.2. Each port must be bound to a different URL. That is, you cannot support both versions concurrently with the same URL address.

See Also:

"soapVersion" on page 18-66 for more information on the WebServicesAssembler `soapVersion` argument.

How to Use SOAP 1.2 Formatted Messages in Top Down Web Service Assembly

To support SOAP 1.2 messages in top down Web services development, you must supply a WSDL with a SOAP 1.2 binding. A WSDL with a SOAP 1.2 binding contains a set of URIs specific to SOAP 1.2. These URIs are listed in Table 5–4.

Table 5–4 URIs for SOAP 1.2 Messages

URI	Description
`http://schemas.xmlsoap.org/wsdl/soap12`	The namespace of the SOAP 1.2 binding element that goes into the WSDL `binding` element.
`http://www.w3.org/2003/05/soap-encoding`	For a SOAP 1.2 message, indicates the encoding rules that the contents of the containing element follows. For more information, see the SOAP Version 1.2 Part 2 Recommendation at: `http://www.w3.org/TR/2003/REC-soap12-part2-20030624/`

`http://www.w3.org/2003/05/soap/bindings/HTTP`
and
`http://schemas.xmlsoap.org/soap/http`
	Describes HTTP transport for SOAP 1.2. Both URIs are accepted by the OracleAS Web Services stack, but the `schemas.xmlsoap.org` URI is more interoperable than the `www.w3.org` URI. Therefore, the `schemas.xmlsoap.org` URI is used when the WSDL is generated for bottom up Web Service assembly.

Example 5–9 displays a WSDL that supports SOAP 1.2 messages. The URIs and elements that are needed to support SOAP 1.2 are displayed in **bold** font.

Example 5–9 Sample WSDL with SOAP 1.2 Binding

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
     name="Rpclitbottomup"
     targetNamespace="http://www.oracle.ws/rpcliteral"
     xmlns="http://schemas.xmlsoap.org/wsdl/"
     xmlns:tns="http://www.oracle.ws/rpcliteral"
     xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
     xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
     xmlns:tns0="http://www.oracle.ws/rpcliteral/schema"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    >
    <types>
        <schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.oracle.ws/rpcliteral/schema"
             elementFormDefault="qualified"
xmlns:tns="http://www.oracle.ws/rpcliteral/schema"
             xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <complexType name="HelloMessage">
                <sequence>
                    <element name="longValue" type="long"/>
                    <element name="age" type="int"/>
                    <element name="greeting" type="string" nillable="true"/>
                    <element name="name" type="string" nillable="true"/>
                    <element name="id" type="decimal" nillable="true"/>
                </sequence>
            </complexType>
        </schema>
    </types>
```

```
    <message name="HelloInterface_hello">
        <part name="msg" type="tns0:HelloMessage"/>
    </message>
    <message name="HelloInterface_helloResponse">
        <part name="result" type="tns0:HelloMessage"/>
    </message>
    <portType name="HelloInterface">
        <operation name="hello" parameterOrder="msg">
            <input message="tns:HelloInterface_hello"/>
            <output message="tns:HelloInterface_helloResponse"/>
        </operation>
    </portType>
    <binding name="HelloInterfacePortBinding" type="tns:HelloInterface">
        <soap12:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="hello">
            <soap12:operation soapAction="http://www.oracle.ws/rpcliteral/hello"
soapActionRequired="false"/>
            <input>
                <soap12:body use="literal"
namespace="http://www.oracle.ws/rpcliteral" parts="msg"/>
            </input>
            <output>
                <soap12:body use="literal"
namespace="http://www.oracle.ws/rpcliteral" parts="result"/>
            </output>
        </operation>
    </binding>
    <service name="Rpclitbottomup">
        <port name="HelloInterfacePort" binding="tns:HelloInterfacePortBinding">
            <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
        </port>
    </service>
</definitions>
```

Limitations

See "OracleAS Web Services Messages" on page F-1.

Additional Information

For more information on:

- using the WebServicesAssembler tool to assemble Web services, see Chapter 18, "Using WebServicesAssembler".
- Web services interoperability, see "Ensuring Interoperable Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- message attachments, and using attachments in your Web service, see "Working with Message Attachments" in the *Oracle Application Server Advanced Web Services Developer's Guide*
- processing nonstandard data types, see "Custom Serialization of Java Value Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- the JAX-RPC mapping file, see "JAX-RPC Mapping File Descriptor" in the *Oracle Application Server Advanced Web Services Developer's Guide*.

- data types supported by OracleAS Web Services, see "Mapping Java Types to XML and WSDL Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.

6

Assembling a Web Service from WSDL

This chapter describes how to assemble a Web service, starting with a Web Service Description Language (WSDL) file. This is also known as top down Web service generation.

Understanding Top Down Assembly

In top down Web service assembly, you generate a service from an existing WSDL file that models the business processes you want to expose.

A development tool, such as WebServiceAssembler, uses the WSDL to generate the service endpoint interface for the service. You can then implement the service for any supported architecture, such as Java classes. After compiling the implementation, you generate the service and deploy it to the application server.

Assembling a Web service requires the WebServiceAssembler tool, and Java platform tools such as the `javac` compiler, that are found in the J2SE 1.4 SDK distribution.

These are the general steps for generating a Web service top down:

1. Generate the service endpoint interface.

   WebServicesAssembler can perform this step.

2. Implement and compile the services.

   The developer performs this step.

3. Assemble the services.

   WebServicesAssembler can perform this step. "Steps for Assembling the Web Service Top Down" on page 6-3 provides more detail on each of these steps.

See Also:

Chapter 5, "OracleAS Web Services Messages" for information about the message formats you can assign to a Web service.

Note: If you are assembling Web Services top down, WebServicesAssembler cannot consume WSDLs that contain the `xsd:choice` or `xsd:group` XML types. If you want to consume a WSDL that contains these XML types, set the WebServicesAssembler `dataBinding` argument to `false` and code the `SOAPElement` so that the payload conforms to the schema definition in the WSDL file.

How to Assemble a Web Service Top Down

This section contains the following subsections:

- Prerequisites
- Steps for Assembling the Web Service Top Down

Prerequisites

Generating a Web service top down with WebServiceAssembler requires you to specify only the WSDL and an output directory.

Before you generate Web services, consider these issues:

- WebServicesAssembler places some restrictions on the WSDL that you specify:
  - The WSDL should comply with Web Services-Interoperability (WS-I) Basic Profile 1.0. If it does not, WebServiceAssembler provides command-line arguments that enable you to work around many limitations in the WSDL.
  - Only one service element can be implemented. WebServicesAssembler enables you to generate the artifacts for only one service at a time. If more than one service is described in the WSDL, a command line argument, `serviceName`, enables you to specify the service you want to use.
  - The message format is specified in the WSDL. You cannot use WebServicesAssembler to change the message format in top down Web service development.
  - The WSDL cannot contain multiple message formats. Remove any ports from the WSDL that reference a binding with a message format that you do not want to use.
  - National Language Support (also known as "NLS" or "Globalization Support") characters that occur in names in the WSDL, such as in the name of a service, port type, operation, binding or port, are not supported. This may also result in errors during code generation or in the Web Services Test Page.
- Decide whether you want to use wrapped or unwrapped parameters. To control this, WebServicesAssembler provides the `unwrapParameters` command-line option.
- If you use nonstandard data types, ensure that the `oracle-webservices.xml` deployment descriptor defines how they will be handled. This file can be used to identify the name of the serialization class that converts the data between XML and Java, the name of the Java class that describes the Java representation of the data, and so on.

See Also:

- Chapter 18, "Using WebServicesAssembler" provides descriptions of WebServicesAssembler commands and arguments.
- "Mapping Java Types to XML and WSDL Types" in the *Oracle Application Server Advanced Web Services Developer's Guide* and Table 5–1 on page 5-7 for more information on using non-standard data types.
- Appendix C, "oracle-webservices.xml Deployment Descriptor Schema" provides more information on this file.
- "Custom Serialization of Java Value Types" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on working with nonstandard data types

Steps for Assembling the Web Service Top Down

The following steps illustrate how to assemble a Web service top down. The Web service will provide a logging facility that is defined by the WSDL stored at `wsdl/LoggingFacility.wsdl`.

1. Provide a WSDL from which the Web service will be generated as input to the WebServiceAssembler `genInterface` command.

   Command line:

   ```
   java -jar wsa.jar -genInterface
                     -output build/src/service
                     -wsdl wsdl/LoggingFacility.wsdl
                     -unwrapParameters false
                     -packageName oracle.demo.topdowndoclit.service
                     -mappingFileName type-mapping.xml
   ```

   Ant task:

   ```
   <oracle:genInterface wsdl="wsdl/LoggingFacility.wsdl"
                 output= "build/src/service"
                 packageName= "oracle.demo.topdowndoclit.service"
                 mappingFileName="type-mapping.xml"
                 dataBinding="true"
                 unwrapParameters="false"
   />
   ```

   In this command line and Ant task:

   - `genInterface`—Creates a service endpoint interface for each port type and a Java value type class (beans) for any complex type defined in a WSDL. It also creates a JAX-RPC mapping file that describes the mapping between the XML schema types and the Java value type classes. See "genInterface" on page 18-31.
   - `output`—Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See"output" on page 18-48.
   - `wsdl`—Specifies the absolute file path, relative file path, or URL to a WSDL document. See "wsdl" on page 18-65.
   - `unwrapParameters`—When set to `false` the generated service endpoint interface will be generated with wrappers around the input parameter and the return type. See "unwrapParameters" on page 18-69.

- `packageName`—Specifies the package name that will be used for generated classes if no package name is declared in the JAX-RPC mapping file. See "packageName" on page 18-48.
- `mappingFileName`—Specifies a file location that points to a JAX-RPC mapping file. See "mappingFileName" on page 18-47.
- `dataBinding`—If `true`, WebServicesAssembler will attempt to generate Java value types that follow the JAX-RPC default type mapping rules for every element in the schema. See "dataBinding" on page 18-69.

At a minimum, specify the name of the WSDL. For more information on the required and optional arguments to `genInterface` see"genInterface" on page 18-31.

The WebServiceAssembler tool generates a Java interface for every port type specified in the WSDL, and a Java Bean for each complex type. 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/topdowndoclit/service`. The value types are stored in the specified output directory, but the package name is based on the type namespace or values in the JAX-RPC mapping file, `type-mapping.xml`.

2. Compile the generated interface and type classes. For example:

```
javac build/src/service/*.java -destdir build/classes
```

3. Create the service endpoint implementation for your Web service.

   The Java service endpoint must have a method signature that matches every method in the generated Java interface. This example assumes that you are creating the service in the file `DocLitLoggerImpl.java`.

4. Compile the Java service endpoint.

   For example, you can use the same command as in Step 2 if the Java service endpoint interface source was generated in the same directory where the `Impl` class was saved. If it was not, then you must change the value of the `classpath` argument.

5. Assemble the service by running the WebServiceAssembler tool with the `topDownAssemble` command. In this example, `doclit_topdown` is an application name for the generated Web service. For example:

   Command line:

```
java -jar wsa.jar -topDownAssemble
            -appName doclit_topdown
            -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
```

   Ant task:

```
<oracle:topDownAssemble
```

```
            appName="doclit_topdown"
            wsdl="./wsdl/LoggingFacility.wsdl"
            unwrapParameters="false"
            input="build/classes/service "
            output="build"
            ear="dist/doclit_topdown.ear"
            mappingFileName="type-mapping.xml"
            packageName="oracle.demo.topdowndoclit.service"
            fetchWsdlImports="true"
            >
               <oracle:portType
                  className="oracle.demo.topdowndoclit.service.DocLitLoggerImpl"/>
   </oracle:topDownAssemble>
```

In this command and Ant task:

- `topDownAssemble`—Creates the required classes and deployment descriptors for a Web service based on a WSDL description. The files can be stored in either an EAR file, a WAR file, or a directory. See "topDownAssemble" on page 18-22.
- `appName`—Specifies the name of an application. Usually, this name is used as a base value for other arguments like `context` and `uri`. See "appName" on page 18-42.
- `wsdl`—Specifies the absolute file path, relative file path, or URL to a WSDL document. See "wsdl" on page 18-65.
- `unwrapParameters`—When set to `false` the generated service endpoint interface will be generated with wrappers around the input parameter and the return type. See "unwrapParameters" on page 18-69.
- `input`—Specifies the directory or JAR containing the classes that should be copied to `WEB-INF/classes`. This argument will be added to the classpath used by the WebServicesAssembler. See "input" on page 18-46.
- `output`—Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See "output" on page 18-48.
- `ear`—Specifies the name and location of the generated EAR. See "ear" on page 18-44.
- `mappingFileName`—Specifies a file location that points to a JAX-RPC mapping file. See "mappingFileName" on page 18-47.
- `packageName`—Specifies the package name that will be used for generated classes if no package name is declared in the JAX-RPC mapping file. See "packageName" on page 18-48.
- `fetchWsdlImports`—Indicates if you want to make a local copy of the WSDL and everything it imports. See "fetchWsdlImports" on page 18-64.

At a minimum, specify the name of the WSDL, the class name that implements the service, and the name of the output directory. The WebServiceAssembler tool outputs an EAR file, and a WAR file within the EAR. The WAR file contains the service artifacts, the implementation classes, the Web deployment descriptor (`web.xml`) and the JAX-RPC deployment descriptor (`webservices.xml`). For more information on the required and optional arguments to `topDownAssemble`, see "topDownAssemble" on page 18-22.

6. 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*. The following is a sample deployment command.

```
java -jar <oc4jHome>/j2ee/home/admin_client.jar deployer:oc4j:localhost:port
<user> <password>
            -deploy
            -file dist/doclit_topdown.ear
            -deploymentName doclit_topdown
            -bindWebApp default-web-site
```

The following list describes the parameters in this code example:

- *<oc4j_Home>*—the directory containing the OC4J installation.
- *<user>*—the user name for the OC4J instance. The user name is assigned at installation time.
- *<password>*—the password for the OC4J instance. The password is assigned at installation time.
- `doclit_topdown`—the name of the application.
- `default-web-site`—the Web site to which the application will be bound. This is usually `default-web-site`. To configure Web sites, see the `server.xml` file in *<oc4j_home>*`/j2ee/home/config`.

7. (Optional) Check that deployment succeeded. OracleAS Web Services provides a Web Service Test Page for each deployed Web service. See "How to Use the Web Services Test Page" on page 13-6 for information on accessing and using the Web Service Test Page.
8. Generate the client code:
   - For the J2SE environment, generate stubs (client proxies) for a J2SE Web service client by running the WebServicesAssembler tool with the `genProxy` command. For more information on generating and assembling client-side code for a J2SE Web service, see Chapter 15, "Assembling a J2SE Web Service Client".
   - For the J2EE environment, 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 for a J2EE Web service, see Chapter 14, "Assembling a J2EE Web Service Client".

   For example, the following command assembles client proxies (stubs) that can be used for a J2SE client. In this example, `doclit_topdown` is an application name for the generated Web service.

   Command line:

   ```
   java -jar wsa.jar -genProxy
                  -wsdl http://localhost:8888/doclit_topdown/doclit_topdown?WSDL
                  -unwrapParameters false
                  -output build/src/client
                  -packageName oracle.demo.topdowndoclit.stubs
                  -mappingFileName type-mapping.xml
   ```

   Ant task:

   ```
   <oracle:genProxy
       wsdl="http://localhost:8888/doclit_topdown/doclit_topdown?WSDL"
       unwrapParameters="false"
   ```

```
    output="build/src/client"
    packageName="oracle.demo.topdowndoclit.stubs"
    mappingFileName="type-mapping.xml"
/>
```

In this command and Ant task:

- `genProxy`—Creates a static proxy stub that can be used by a J2SE Web service client. See "genProxy" on page 18-34.
- `wsdl`—Specifies the absolute file path, relative file path, or URL to a WSDL document. See "wsdl" on page 18-65.
- `unwrapParameters`—When set to `false` the generated service endpoint interface will be generated with wrappers around the input parameter and the return type. See "unwrapParameters" on page 18-69.
- `output`—Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See "output" on page 18-48.
- `packageName`—Specifies the package name that will be used for generated classes if no package name is declared in the JAX-RPC mapping file. See "packageName" on page 18-48.
- `mappingFileName`—Specifies a file location that points to a JAX-RPC mapping file. See "mappingFileName" on page 18-47.

At a minimum, specify the name of the WSDL and the name of the output directory. The WebServiceAssembler tool generates a stub. A client application uses the stub to invoke operations on a remote service. For more information on the required and optional arguments to `genProxy`, see "genProxy" on page 18-34.

9. Compile and run the client.

   List the appropriate JARs on the classpath before compiling the client. Table A–2, " Classpath Components for a Client Using a Client-Side Proxy" 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 Table A–2. See "Setting the Classpath for a Web Service Proxy" on page A-2 for more information on `wsclient_extended.jar` and the client classpath.

Limitations

See "Assembling Web Services from a WSDL" on page F-5.

Additional Information

For more information on:

- using the Test Page to test Web service deployment, see Chapter 13, "Testing Web Services Deployment".
- building a J2EE client, see Chapter 14, "Assembling a J2EE Web Service Client".
- building a J2SE client, see Chapter 15, "Assembling a J2SE Web Service Client".
- JAX-RPC handlers, see Chapter 16, "Using JAX-RPC Handlers".
- using the WebServicesAssembler tool to assemble Web services, see Chapter 18, "Using WebServicesAssembler".

- packaging and deploying Web services, see Chapter 19, "Packaging and Deploying Web Services".
- Web services interoperability, see "Ensuring interoperable Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- using quality of service features in Web service clients, see "Managing Web Service" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- adding security to a Web service, see the *Oracle Application Server Web Services Security Guide*.
- adding reliability to a Web service, see "Ensuring Web Service Reliability" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- adding an auditing and logging configuration to a Web service, see "Auditing and Logging Messages" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- processing nonstandard data types, see "Custom Serialization of Java Value Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- the JAX-RPC mapping file, see "JAX-RPC Mapping File Descriptor" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- data types supported by OracleAS Web Services, see "Mapping Java Types to XML and WSDL Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- Oracle JDeveloper tool support for Web service development, see the Oracle JDeveloper on-line help.

7

Assembling a Web Service with Java Classes

This chapter describes how to assemble stateless and stateful Web services based on Java classes. The assembly is performed bottom up by WebServicesAssembler.

A stateless Web service does not carry any local state across calls. Stateless Web services interoperate with .NET or any vendor's Web services. The service can use either HTTP or JMS as the transport mechanism.

In contrast, a stateful Web service may carry state across calls, and the results of method invocations depend on the scope. The stateful Web services supported by Oracle Application Server Web Services is HTTP-based. Stateful OracleAS Web Services based on Java classes contain Oracle proprietary extensions and may not operate with other services unless the service provider makes available scopes with the same semantics.

This chapter has the following sections:

- Requirements for Writing Java Classes for Web Services
- Java Classes and Stateless Web Services
- Java Classes and Stateful Web Services

Requirements for Writing Java Classes for Web Services

To use JAX-RPC to create a Web service with Java files, you must provide a public interface that defines the remote methods that you want to expose as a service. The interface definition must also extend `java.rmi.Remote`, and its methods must throw a `java.rmi.RemoteException` object. The interface must also reside in a package.

You can also use the public interface to list the signatures for the public methods, or public methods with supported data types, that you want to make available to the Web service. That is, you can employ the interface to filter the methods that you want to expose.

The implementation of the interface must satisfy other requirements:

- The class must contain a default public constructor.
- The class methods must implement the methods of the service endpoint interface.
- Class methods must not be final.
- For stateful Web services, the class must implement `java.io.Serializable`.
- The class must reside in a package.
- All methods in the interface must throw `java.rmi.RemoteException`. In addition, methods can declare other, specific exceptions. These must extend

`java.lang.Exception` directly or indirectly but must not be a `RuntimeException`.

- Method parameters and return types must be JAX-RPC supported Java types.

  See "Mapping Java Types to XML and WSDL Types" in the *Oracle Application Server Advanced Web Services Developer's Guide* provides a list of supported Java types.

- Holder classes can be used as method parameters. These holder classes are either generated or derived from the `javax.xml.rpc.holders` package.
- The implementation class must not include public final static declarations.
- A service endpoint interface must not include a remote reference (a class that implements `RemoteInterface`) as either a parameter or a return type. A Java array or JAX-RPC value type must not include a remote reference as a contained element.

  For a description of all of the requirements on the interface, see the Enterprise Web Services 1.1 specification at the following Web address: `http://www.jcp.org/aboutJava/communityprocess/final/jsr109/index.html`

When a Web service client makes a service request, OC4J runs the corresponding method in that class. There are very few restrictions on what actions the Web service can perform. At a minimum, the Web services generate some data that is sent to the client or perform an action specified by a Web service request.

Note: OracleAS Web Services may be able to expose a Java class directly as a Web service even if you do not provide a service interface and if your class does not meet some of the conditions for a compliant class. To expose a Java class as a Web service in this case, you must use J2SE 5.0 Annotations such as `@WebService`. For more information, see Chapter 11, "Assembling Web Services with Annotations".

Also note that a Web service assembled under these conditions will not be conformant or portable as defined by the JAX-RPC specification.

See Also:

"How to Write Java Implementations for Stateless Web Services" on page 7-7 for information on writing Java classes for stateless Web services and "How to Write Java Implementations for Stateful Web Services" on page 7-13 for information on writing Java classes for stateful Web services.

Java Classes and Stateless Web Services

This section has the following subsections:

- Exposing Java Classes as a Stateless Web Service
- How to Assemble A Stateless Web Service with Java Classes
- How to Write Java Implementations for Stateless Web Services

Exposing Java Classes as a Stateless Web Service

You can use WebServicesAssembler to assemble Web services from Java classes that conform to the JAX-RPC 1.1 specification. Exposing Java classes as a Web service is convenient if you want a lightweight system and do not need the transactional capabilities that an EJB container offers.

WebServicesAssembler assembles the service bottom up. It starts with the Java classes you want to expose and generates a deployable EAR file containing the WSDL, the mapping files, the implementation files, and the deployment descriptors.

JAX-RPC requires you to provide a Java class that contains the methods you want to expose as a service and its interface.

A Web service based on Java classes can be invoked by a client written in Java, .NET, or any other programming language. The Web service can use either HTTP or JMS as the transport mechanism. The client can be based on static stub or Dynamic Invocation Interface (DII).

See Also:

"Requirements for Writing Java Classes for Web Services" on page 7-1 for more information on the requirements on the class and interface.

How to Assemble A Stateless Web Service with Java Classes

This section contains the following subsections:

- Prerequisites
- Steps for Assembling a Stateless Web Service with Java Classes

Prerequisites

Before you begin, provide the following files and information.

- Supply a compiled Java class and interface that contains the methods that you want to expose as a service. The class and its interface must conform to the JAX-RPC standards for a Web service.

  See "How to Write Java Implementations for Stateless Web Services" on page 7-7 if you are exposing Java classes as a stateless Web service. See "Exposing Java Classes as a Stateful Web Service" on page 7-9 if you are exposing Java classes as a stateful Web service.

- Decide whether you want WebServicesAssembler to only generate the service files or if you want it to package the files into a deployable archive. The `ear` argument packages the files into an archive. If you do not specify `ear`, then the files are stored in the directory specified by the `output` argument.

  See "ear" on page 18-44, "output" on page 18-48, and "war" on page 18-51 for more information on these arguments.

- If your Java classes need to work with any additional message processing components, for example to provide reliability and security features, you can specify message handlers.

  See Chapter 17, "Processing SOAP Headers" and "How to Configure Handlers in an Ant Task" on page 18-81 for more information on these topics.

- If the methods in the Java class use nonstandard data types, you must specify a custom serializer to process them.

If you are working with non-supported data types, see "Mapping Java Types to XML and WSDL Types" in the *Oracle Application Server Advanced Web Services Developer's Guide* and Table 5–1, " Java Types Supported by RPC-Encoded Proprietary Message Format" on page 5-7.

For information on custom serialization, see "Custom Serialization of Java Value Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.

For a list of supported data types, see the JAX-RPC 1.1 specification available from the following Web site:

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

Steps for Assembling a Stateless Web Service with Java Classes

The following steps describe how to use WebServicesAssembler to expose a stateless Web service from a Java class:

1. Provide the compiled Java class that you want to expose as a Web service and its compiled interface.

   This example uses the `HelloInterface` interface and the `HelloImpl` class. You can find code listings of these files "Steps for Assembling a Stateless Web Service with Java Classes" on page 7-4.

2. Generate the service artifacts by running the WebServicesAssembler with the `assemble` command. This example assumes that the interface and implementation classes are compiled to the `./build/classes/service` directory.

   Command line:

```
java -jar wsa.jar -assemble
    -appName hello
    -serviceName HelloService
    -interfaceName oracle.demo.hello.HelloInterface
    -className oracle.demo.hello.HelloImpl
    -input ./build/classes/service
    -output build
    -ear dist/hello.ear
    -uri HelloService
    -targetNamespace http://hello.demo.oracle
    -classpath ./build/classes/service
```

   Ant task:

```
<oracle:assemble appName="hello"
        serviceName="HelloService"
        input="./build/classes/service"
        output="build"
        ear="dist/hello.ear"
        targetNamespace="http://hello.demo.oracle"
        >
            <oracle:porttype
                interfaceName="oracle.demo.hello.HelloInterface"
                className="oracle.demo.hello.HelloImpl">
                <oracle:port uri="HelloService" />
            </oracle:porttype>
            <oracle:classpath>
                <pathelement location="build/classes/service"/>
            </oracle:classpath>
</oracle:assemble>
```

In this command and Ant task:

- `assemble`—Generates a Web service from Java files bottom up. The command creates all of the files required to create a deployable archive. See "assemble" on page 18-7.
- `appName`—Specifies the name of an application. Usually, this name is used as a base value for other arguments like `context` and `uri`. See "appName" on page 18-42.
- `serviceName`—Specifies the service name. See "serviceName" on page 18-50.
- `interfaceName`—Specifies the name of a Java class (including the package name) that contains the service endpoint interface (SEI). See "interfaceName" on page 18-47.
- `className`—Specifies the name of the class (including the package name) that is the implementation class for the Web service. See "className" on page 18-43.
- `input`—Specifies the directory or JAR containing the classes that should be copied to `WEB-INF/classes`. This argument will be added to the classpath used by the WebServicesAssembler. See "input" on page 18-46.
- `output`—Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See "output" on page 18-48.
- `ear`—Specifies the name and location of the generated EAR. See "ear" on page 18-44.
- `uri`—Specifies the URI to use for the Web service. See "uri" on page 18-63.
- `targetNamespace`—Specifies the target namespace to be used in the generated WSDL. The value can be either a conforming HTTP URL, a non-conforming HTTP URL, or even a URI. See "targetNamespace" on page 18-66.
- `classpath`—Specifies the classpath that contains any user-created classes given to WebServicesAssembler. One reason to specify this argument is if you have already created some value type classes or exceptions and you do not want WebServicesAssembler to overwrite them. See "classpath" on page 18-43.

The output of this command is an EAR file that contains the contents of a WAR file that can be deployed to an OC4J instance. The `dist` directory contains the J2EE Web services-compliant application EAR file, `hello.ear`. For more information on the required and optional arguments to `assemble`, see "assemble" on page 18-7.

3. 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/hello.ear
            -deploymentName hello
            -bindWebApp default-web-site
```

   The following list describes the variables used in this code example.

   - *<OC4J_Home>*—The directory containing the OC4J installation.

- *<user>*—The user name for the OC4J instance. The user name is assigned at installation time
- *<password>*—The password for the OC4J instance. The password is assigned at installation time
- `default-web-site`—The Web site to which the application will be bound. This is usually `default-web-site`. To configure Web sites, see the `server.xml` file in `<OC4J_HOME>/j2ee/home/config`.

4. (Optional) Check that deployment succeeded. OracleAS Web Services provides a Web Service Test Page for each deployed Web service. See "How to Use the Web Services Test Page" on page 13-6 for information on accessing and using the Web Service Test Page.
5. Generate the client-side code:
   - For the J2SE environment, generate stubs (client proxies) for a J2SE Web service client by running the WebServicesAssembler tool with the `genProxy` command. For more information on generating and assembling client-side code for the J2SE environment, see Chapter 15, "Assembling a J2SE Web Service Client".
   - For the J2EE environment, generate a Service Endpoint Interface and a 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 Chapter 14, "Assembling a J2EE Web Service Client".

   For example, the following command generates stubs that can be used for a J2SE client:

   Command line:

   ```
   java -jar wsa.jar -genProxy
                     -output build/src/client/
                     -wsdl http://localhost:8888/hello/HelloService?WSDL
                     -packageName oracle.demo.hello
   ```

   Ant task:

   ```
   <oracle:genProxy  wsdl="http://localhost:8888/hello/HelloService?WSDL"
             output="build/src/client"
             packageName="oracle.demo.hello"
   />
   ```

   In this command line and Ant task:

   - `genProxy`—Creates a static proxy stub that can be used by a J2SE Web service client. See "genProxy" on page 18-34.
   - `output`—Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See "output" on page 18-48.
   - `wsdl`—Specifies the absolute file path, relative file path, or URL to a WSDL document. See "wsdl" on page 18-65.
   - `packageName`—Specifies the package name that will be used for generated classes if no package name is declared in the JAX-RPC mapping file. See "packageName" on page 18-48.

   This command generates the client proxies and stores them in the directory `build/src/client`. The client application uses the stub to invoke operations on

a remote service. For more information on the required and optional arguments to `genProxy`, see "genProxy" on page 18-34.

6. Write the client application.
7. Compile and run the client.

   List the appropriate JARs on the classpath before compiling the client. Table A–2, " Classpath Components for a Client Using a Client-Side Proxy" 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 Table A–2. See "Setting the Classpath for a Web Service Proxy" on page A-2 for more information on `wsclient_extended.jar` and the client classpath.

How to Write Java Implementations for Stateless Web Services

OracleAS Web Services supports stateless implementations for Java classes running as Web services. For a stateless Java implementation, OracleAS Web Services creates multiple instances of the Java class in a pool; any one of the instances can be used to service a request. After servicing the request, the object is returned to the pool for use by a subsequent request.

You can assemble Java classes for stateless Web services only for SOAP/HTTP endpoints and not for SOAP/JMS endpoints.

To develop a stateless Java Web service, define the Java interface and its implementation. This is described in the following sections:

- How to Define a Java Interface
- How to Define a Java Class

> **Note:** OracleAS Web Services may be able to expose a Java class directly as a Web service even if you do not provide a service interface and if your class does not meet some of the conditions for a compliant class. To expose a Java class as a Web service in this case, you must use J2SE 5.0 Annotations such as `@WebService`. For more information, see Chapter 11, "Assembling Web Services with Annotations".
>
> Also note that a Web service assembled under these conditions will not be conformant or portable as per the JAX-RPC specification.

How to Define a Java Interface

Example 7–1 displays the `HelloInterface.java` interface for the stateless Web service. To comply with the JAX-RPC 1.1 specification, the interface must reside in a package. It must also extend `java.rmi.Remote`, and its methods must throw a `java.rmi.RemoteException` object.

Example 7–1 Defining an Interface for a Stateless Web Service

```
package oracle.demo.hello;

import java.rmi.RemoteException;
import java.rmi.Remote;

public interface HelloInterface extends Remote {
    public String sayHello(String name) throws RemoteException;
```

```
}
```

How to Define a Java Class

Create a Java class by implementing the methods in the interface that you want to expose as a Web service. A Java class for a Web service usually defines one or more public methods. To comply with the JAX-RPC 1.1 specification, the implementation class must reside in a package. It must also import `java.rmi.Remote` and `java.rmi.RemoteException`.

Example 7–2 displays the public class, `HelloImpl`. This class defines a public method, `sayHello`, that returns the string `"Hello` *name*`!"` where *name* is an input value.

Example 7–2 Defining a Public Class for a Stateless Web Service

```
package oracle.demo.hello;

import java.rmi.RemoteException;
import java.rmi.Remote;

public class HelloImpl {
     public HelloImpl() {
   }
   public String sayHello(String name) {
     return ("Hello " + name + "!");
   }
}
```

Notice that Java class Web service implementations must include a public constructor that takes no arguments.

When an error occurs while running a method on the Java class, it throws a `RemoteException`. In response to the exception, OracleAS Web Services returns a Web service (SOAP) fault. Use standard J2EE and OC4J administration facilities to view errors for a Web service that uses Java classes for its implementation.

When you create a Java class containing methods that implement a Web service, the methods, parameters, and return values must use supported types or nonstandard types supported by OracleAS Web Services. For a list of the supported data types, see the JAX-RPC 1.1 specification available from the following Web site:

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

> **See Also:**
>
> "Mapping Java Types to XML and WSDL Types" in the *Oracle Application Server Advanced Web Services Developer's Guide* and Table 5–1 on page 5-7 for lists of supported data types and supported nonstandard types.

If methods, parameters, and return values use unsupported types, then you must handle them in either of the following ways.

- Use the interface class to limit the exposed methods to only those using JAX-RPC supported types and the supported non-standard types.
- Use the custom serializer to map unsupported types.

See Also:

"Custom Serialization of Java Value Types" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on working with unsupported types.

Java Classes and Stateful Web Services

This section has the following subsections:

- Exposing Java Classes as a Stateful Web Service
- How to Assemble a Stateful Web Service
- How to Write Java Implementations for Stateful Web Services

Exposing Java Classes as a Stateful Web Service

OC4J supports stateful Web services based on Java classes. The Java object that implements the service persists for the duration of the HTTP session. To maintain state, these services contain Oracle proprietary extensions. Because of these extensions, you should not consider stateful OracleAS Web Services to be interoperable unless the service provider makes available scopes with the same semantics.

The stateful Web services supported by OracleAS Web Services is HTTP session-based.

How to Assemble a Stateful Web Service

This section contains the following subsections:

- Prerequisites
- Steps for Assembling a Stateful Web Service

Prerequisites

The prerequisites for generating a stateful Web service from Java classes are identical to those described for a stateless Web service. For a description of the information and files you must provide, see "Prerequisites" on page 7-3.

Steps for Assembling a Stateful Web Service

The following instructions describe how to use WebServicesAssembler to create a stateful Web service from a Java class. The Java object that implements the service persists for the duration of the HTTP session.

1. Provide the Java class that you want to expose as a Web service and its interface.
2. Generate the service artifacts by running the WebServicesAssembler with the `assemble` command. For more information on the required and optional arguments to `assemble`, see "assemble" on page 18-7.

   For example:

   Command line:

```
java -jar wsa.jar -assemble
    -appName counter
    -serviceName counterService
    -interfaceName oracle.demo.count.CounterInterface
    -className oracle.demo.count.CounterImpl
    -input build/classes/service
```

```
    -output build
    -ear dist/counter.ear
    -recoverable true
    -timeout 30
    -uri counterService
```

Ant task:

```
<oracle:assemble appName="counter"
            serviceName="counterService"
            input="build/classes/service"
            output="build"
            ear="dist/service.ear"
            recoverable="true"
            timeout="30"
        >
        <oracle:porttype
            interfaceName="oracle.demo.count.CounterInterface"
            className="oracle.demo.count.CounterImpl">
            <oracle:port uri="counterService" />
        </oracle:porttype>
        <oracle:classpath>
            <pathelement path="${wsdemo.common.class.path}"/>
            <pathelement location="build/classes/client"/>
        </oracle:classpath>
    </oracle:assemble>
```

In this command and Ant task:

- `assemble`—Generates a Web service from Java files bottom up. The command creates all of the files required to create a deployable archive. See "assemble" on page 18-7.
- `appName`—Specifies the name of an application. Usually, this name is used as a base value for other arguments like `context` and `uri`. See "appName" on page 18-42.
- `serviceName`—Specifies the service name. See "serviceName" on page 18-50.
- `interfaceName`—Specifies the name of a Java class (including the package name) that contains the service endpoint interface (SEI). See "interfaceName" on page 18-47.
- `className`—Specifies the name of the class (including the package name) that is the implementation class for the Web service. See "className" on page 18-43.
- `input`—Specifies the directory or JAR containing the classes that should be copied to `WEB-INF/classes`. This argument will be added to the classpath used by the WebServicesAssembler. See "input" on page 18-46.
- `output`—Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See "output" on page 18-48.
- `ear`—Specifies the name and location of the generated EAR. See "ear" on page 18-44.
- `recoverable`—Indicates whether applications with session state are recoverable. This argument can be used only when the service is exposed as a stateful Web service with session scope. See "recoverable" on page 18-52.
- `timeout`—Specifies the number of seconds a session should last before it times out. See "timeout" on page 18-52.

- `uri`—Specifies the URI to use for the Web service. See "uri" on page 18-63.

This command generates all the files required to create a deployable archive. The output `build` directory contains separate directories for the EAR file and the Java classes. The `dist` directory contains the J2EE Web services-compliant application EAR file, `counter.ear`.

Note the `timeout` argument on the command line. In addition to indicating the number of seconds an HTTP session should last before it times out, it also implicitly sets the `session` argument to `true`. When `session` is `true`, the service instance is stored in an HTTP session.

The `recoverable` argument indicates that this stateful application needs to be distributable. Recoverable means that you want the service to be able to recover in a distributed environment if the node that you are interacting with goes down. This means that the Web service state must be also be distributable.

3. Deploy the service and bind the application.

   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*. 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/counter.ear
          -deploymentName counter
          -bindWebApp default-web-site
```

   The following list describes the parameters used in this code example.

   - *<OC4J_HOME>*—The directory containing the OC4J installation.
   - *<user>*—the user name for the OC4J instance. The user name is assigned at installation time.
   - *<password>*—the password for the OC4J instance. The password is assigned at installation time.
   - `default-web-site`— the Web site to which the application will be bound. This is usually `default-web-site`. To configure Web sites, see the `server.xml` file in *<OC4J_HOME>*`/j2ee/home/config`.

4. (Optional) Check that deployment succeeded.

   OracleAS Web Services provides a Web Service Test Page for each deployed Web service. See "How to Use the Web Services Test Page" on page 13-6 for information on accessing and using the Web Service Test Page.

5. Generate the client code:

   - For the J2SE environment, generate stubs (client proxies) by running the WebServicesAssembler tool with the `genProxy` command. For more information on generating and assembling client-side code for the J2SE environment, see Chapter 15, "Assembling a J2SE Web Service Client".
   - For the J2EE environment, generate a Service Endpoint Interface and a 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 Chapter 14, "Assembling a J2EE Web Service Client".

For example, the following command generates client proxies (stubs) that can be used for a J2SE client:

Command line:

```
java -jar wsa.jar -genProxy
                  -output build/src/client/
                  -wsdl http://localhost:8888/counter/counterService?WSDL
                  -packageName oracle.demo.count
```

Ant task:

```
<oracle:genProxy wsdl="http://stadp54.us.oracle.com:8888/counter/counter?WSDL"
    output="build/src/client"
    packageName="oracle.demo.count"/>
```

In this command and Ant task:

- `genProxy`—Creates a static proxy stub that can be used by a J2SE Web service client. See "genProxy" on page 18-34.
- `output`—Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See "output" on page 18-48.
- `wsdl`—Specifies the absolute file path, relative file path, or URL to a WSDL document. See "wsdl" on page 18-65.
- `packageName`—Specifies the package name that will be used for generated classes if no package name is declared in the JAX-RPC mapping file. See "packageName" on page 18-48.

This command generates the client proxies and stores them in the directory `build/src/client`. The client application uses the stub to invoke operations on a remote service. For more information on the required and optional arguments to `genProxy`, see "genProxy" on page 18-34.

Note: On systems such as Unix or Linux, the URL might need to be quoted (" ") on the command line.

---

6. Write the client application.

   Ensure that the client participates in the session by setting the `javax.xml.rpc.Call.SESSION_MAINTAIN_PROPERTY` runtime property to `true` either on the stub, the DII call, or the endpoint client instance.

   Instead of setting this property directly, OracleAS Web Services provides a helpful wrapper class with a `setMaintainSession(boolean)` method. Set this method to `true` to maintain sessions. The wrapper takes care of setting the property inside of the client. For example, in the client code, enter:

   ```
   CounterServicePortClient c = new CounterServicePortClient();
   //sets Maintain Session to true, as the endpoint is stateful.
   c.setMaintainSession(true);
   ```

7. Compile and run the client.

   List the appropriate JARs on the classpath before compiling the client. Table A–2, " Classpath Components for a Client Using a Client-Side Proxy" 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 Table A–2. See "Setting the Classpath for a Web Service Proxy" on page A-2 for more information on `wsclient_extended.jar` and the client classpath.

How to Write Java Implementations for Stateful Web Services

Java implementations for a stateful Web service must meet the same requirements as implementations for a stateless service.

> **See Also:**
>
> "Requirements for Writing Java Classes for Web Services", on page 7-1 for a description of the requirements for a Java implementation.

WebServicesAssembler enables you to define call, session, and endpoint scopes for a stateful Java implementation:

- `call` scope—When the WebServicesAssembler callScope argument is set to `true`, a new class instance is created for each call. The instance is garbage collected after each call. The default value of `callScope` is `false`.
- session scope—When the WebServicesAssembler session argument is set to `true`, the class instance is stored in an HTTP session. This applies only for HTTP transport. Session timeout can be tuned by the `timeout` argument.

  The default value for `session` is `false`. However, if the `timeout` argument is set with WebServicesAssembler, then `session` will automatically be set to `true`.
- endpoint scope—This is the default behavior and scope for a Web service application. The service endpoint implementation class instance is a singleton instance for each endpoint.

Developing a stateful Java Web service consists steps described in the following sections:

- How to Define a Java Interface
- How to Define a Java Class

How to Define a Java Interface

Example 7–3 displays the `CountInterface.java` interface for the stateful Web service. This example also shows that the service class does not have to implement the service endpoint interface directly. To comply with the JAX-RPC 1.1 specification, the interface must reside in a package. It must also extend `java.rmi.Remote` and its methods must throw a `java.rmi.RemoteException` object.

Example 7–3 Defining an Interface for a Stateful Web Service

```
package oracle.demo.count;

import java.rmi.RemoteException;
import java.rmi.Remote;

public interface CounterInterface extends Remote {
    // gets the current counter value
    public int getCurrentCounter() throws RemoteException;
}
```

How to Define a Java Class

Create a Java class by implementing the methods in the interface that you want to expose as a Web service. A Java class for a Web service usually defines one or more public methods. To comply with the JAX-RPC 1.1 specification, the implementation class must reside in a package. It must also import `java.rmi.Remote` and `java.rmi.RemoteException`.

Example 7–4 displays the public class, `CounterImpl`. The class initializes the count and defines the public method, `getCurrentCounter`.

Example 7–4 Defining a Public Class for a Stateful Web Service

```
package oracle.demo.count;

import java.rmi.RemoteException;
import java.rmi.Remote;

public class CounterImpl implements java.io.Serializable {
    private int counter = 0;

    public CounterImpl() {
    }

  public int getCurrentCounter() {
      System.out.println("Current counter value is: " + (++counter));
      return (counter);
   }
}
```

Packaging and Deploying Web Services

The packaging of Web services that expose Java classes is described in "Packaging for a Web Service Based on Java Classes" on page 19-2.

See Also:

Oracle Containers for J2EE Deployment Guide for a detailed description of Web module deployment.

Tool Support for Exposing Java Classes as Web Services

With Oracle JDeveloper, you can create, modify, and deploy J2EE-compliant Java class files as Web services. When you create Java classes in Oracle JDeveloper, you use modeling tools and wizards. The wizards can perform the following tasks.

- Import or create Java classes and interfaces in a project.
- Package and deploy the Java classes exposed as Web services.

See Also:

Oracle JDeveloper on-line help for more information on using Oracle JDeveloper to create Java classes and expose them as Web services.

Limitations

See "Assembling Web Services from Java Classes" on page F-6.

Additional Information

For more information on:

- using the Test Page to test Web service deployment, see Chapter 13, "Testing Web Services Deployment".
- building a J2EE client, see Chapter 14, "Assembling a J2EE Web Service Client".
- building a J2SE client, see Chapter 15, "Assembling a J2SE Web Service Client".
- JAX-RPC handlers, see Chapter 16, "Using JAX-RPC Handlers".
- using the WebServicesAssembler tool to assemble Web services, see Chapter 18, "Using WebServicesAssembler".
- packaging and deploying Web services, see Chapter 19, "Packaging and Deploying Web Services".
- JAR files that are needed to assemble a client, see Appendix A, "Web Service Client APIs and JARs".
- Web services interoperability, see "Ensuring Interoperable Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- using quality of service features in Web service clients, see "Managing Web Service" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- adding security to a Web service, see the *Oracle Application Server Web Services Security Guide*.
- adding reliability to a Web service, see "Ensuring Web Service Reliability" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- adding an auditing and logging configuration to a Web service, see "Auditing and Logging Messages" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- processing nonstandard data types, see "Custom Serialization of Java Value Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- the JAX-RPC mapping file, see "JAX-RPC Mapping File Descriptor" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- data types supported by OracleAS Web Services, see "Mapping Java Types to XML and WSDL Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- Oracle JDeveloper tool support for Web service development, see the Oracle JDeveloper on-line help.

8

Assembling a Web Service with EJBs

This chapter describes how to use the WebServicesAssembler tool to expose version 2.1 Enterprise Java Beans (EJBs) as Web services.

This chapter has the following sections.

- Exposing EJBs as Web Services
- How to Assemble a Web Service with EJBs
- How to Write EJBs for Web Services
- Packaging and Deploying Web Services that Expose EJBs
- How to Secure EJB-Based Web Services at the Transport-Level
- Tool Support for Exposing EJBs as a Web Service

Exposing EJBs as Web Services

You can use WebServicesAssembler to expose version 2.1 EJBs as Web services that conform to J2EE 1.4 standards. Many EJBs have been written to encapsulate business functions in the middle tier. If you use EJBs in your enterprise applications, you can expose them as Web services.

> **Note:** WebServicesAssembler cannot be used to expose version 3.0 EJBs as a Web service.

EJB components, by design, are meant for distributed computing and are well-suited for exposure as Web services. The EJB specification supports declarative transactions, thread management, and role-based security. You can leverage these benefits if you decide to use EJB components as Web services. EJBs exposed as Web services can still be accessed by traditional RMI EJB clients as well as by SOAP protocols. J2EE 1.4 allows exposing only stateless session beans as Web services.

By their nature, SOAP and Web services are stateless. Therefore, stateless session beans are an ideal medium for exposure as Web services. Stateless session beans can be used for checking someone's credit, charging a bank account, or placing an order. Session beans that implement a business function to be used by other remote applications are a perfect fit for exposure as Web services.

Writing an EJB Web service using JAX-RPC involves writing an EJB that implements a service and provides an interface for it. The EJB should contain the business logic that a client can invoke when it makes a Web service request.

An EJB Web service does not differ from any other Web service and can be invoked by a client written in Java, .NET, or any other programming language. The client of an EJB Web service can leverage static stubs, dynamic proxies, or Dynamic Invocation Interfaces (DII).

These are the general steps for exposing an EJB as a Web service.

1. Create the service endpoint interface for the stateless EJB component.

   The developer performs this step.

2. Assemble the service artifacts. This includes generating the WSDL and mapping files, and packaging the application into a deployable archive.

   WebServicesAssembler can be used to perform this step.

See Also:

"Steps for Assembling a Web Service from an EJB" for more detailed information on each of these steps.

Working with Version 2.0 EJBs

Although this chapter focuses on exposing version 2.1 EJBs as Web services, version 2.0 EJBs can also be exposed. All the functionality in Oracle Application Server Web Services for working with version 2.1 EJBs is also available for version 2.0 EJBs. OracleAS Web Services and WebServicesAssembler can detect version 2.0 EJBs and ensure that they are processed correctly. Its remote interface methods must define the methods to be exposed as a Web service.

See Also:

"How to Write EJBs for Web Services" on page 8-6 for more information on the J2EE Web Services requirements for EJBs.

Working with Version 3.0 EJBs

You cannot use the WebServicesAssembler tool to expose version 3.0 EJBs as a Web service. However, you can expose these EJBs by using the J2SE 5.0 Web Service Annotations, which is supported by OracleAS Web Services.

See Also:

Chapter 11, "Assembling Web Services with Annotations" and "Steps to Use Annotations to Assemble a Web Service from a Version 3.0 EJB" provides more information on using annotations to expose version 3.0 EJBs as Web services.

How to Assemble a Web Service with EJBs

This section contains the following subsections:

- Prerequisites
- Steps for Assembling a Web Service from an EJB

Prerequisites

Before you begin, provide the following files and information.

- Write an Enterprise Java Bean that contains the business functions that you want to expose and its interface. The EJB and its interface must conform to EJB 2.1 standards and the J2EE 1.4 standard for a Web service.

  See "How to Write EJBs for Web Services" on page 8-6 for more information on these requirements.

- Decide whether you want WebServicesAssembler to only generate the service files or generate and package the files into a deployable archive. The `ear` argument packages the files into an archive. If you do not specify `ear`, then the files are stored in a directory specified by the `output` argument.

  See "ear" on page 18-44, "output" on page 18-48, and "war" on page 18-48 for more information on output and packaging arguments.

- If the methods in the EJB use nonstandard data types, you must specify a custom serializer to process them.

  See "Mapping Java Types to XML and WSDL Types" in the *Oracle Application Server Advanced Web Services Developer's Guide* and in Table 5–1 on page 5-7 for more information on non-standard data types.

  See "Custom Serialization of Java Value Types" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on writing custom serializers.

- If your EJB needs to work with any additional message processing components, for example to process SOAP header information, you can specify message handlers.

  See Chapter 17, "Processing SOAP Headers" and "How to Configure Handlers in an Ant Task" on page 18-81 for more information on these topics.

Steps for Assembling a Web Service from an EJB

The following steps describe how to use WebServicesAssembler to expose a session bean as a Web service.

1. Write the EJB that you want to expose as a Web service and its service endpoint interface.

2. Inspect the `ejb-jar.xml` deployment descriptor.

   Enter a `<service-endpoint>` element and a value if it is not already in the file. This element identifies the service endpoint interface for this Web service. In the following `ejb-jar.xml` fragment the `<service-endpoint>` element is highlighted in bold.

```
<enterprise-beans>
    <session>
        <ejb-name>HelloServiceBean</ejb-name>
        <service-endpoint>oracle.demo.ejb.HelloServiceIntf</service-endpoint>
        <ejb-class>oracle.demo.ejb.HelloserviceBean</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Container</transaction-type>
    </session>
</enterprise-beans>
```

3. Generate the service artifacts by running the WebServicesAssembler tool with the `ejbAssemble` command. For example:

   Command line:

```
java -jar wsa.jar -ejbAssemble
                -appName helloServices-ejb
                -ear dist/helloServices-ejb.ear
                -output build
                -targetNamespace http://oracle.j2ee.ws/ejb/Hello
                -typeNamespace http://oracle.j2ee.ws/ejb/Hello/types
                -input dist/HelloServiceejb.jar
                -ejbName HelloServiceBean
```

Ant task:

```
<oracle:ejbAssemble appName="ejbApp"
          targetNamespace="http://oracle.j2ee.ws/ejb/Hello"
          ear="dist/ejbApp.ear"
          output="build"
          typeNamespace="http://oracle.j2ee.ws/ejb/Hello/types"
          input dist/HelloServiceejb.jar
          ejbName HelloServiceBean
/>
```

In this command and Ant task:

- `ejbAssemble`—Creates an EAR or EJB JAR that can expose an EJB as a Web service. You must specify a valid version 2.1 EJB JAR as input; the system will create a WSDL and the proprietary `oracle-webservices.xml` deployment descriptor. See "ejbAssemble" on page 18-14.
- `appName`—Specifies the name of an application. Usually, this name is used as a base value for other arguments like `context` and `uri`. See "appName" on page 18-42.
- `ear`—Specifies the name and location of the generated EAR. See "ear" on page 18-44.
- `output`—Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See "output" on page 18-48.
- `targetNamespace`—Specifies the target namespace to be used in the generated WSDL. The value can be either a conforming HTTP URL, a non-conforming HTTP URL, or even a URI. See "targetNamespace" on page 18-66.
- `typeNamespace`—Specifies the type namespace to be used in the schema types in the generated WSDL. The name that you specify will always be used and it will not be reversed. See "typeNamespace" on page 18-67.
- `input`—Specifies the directory or JAR containing the classes that should be copied to `WEB-INF/classes`. This argument will be added to the classpath used by the WebServicesAssembler. See "input" on page 18-46.
- `ejbName`—Specifies the name of the EJB to be exposed as a Web service. Note that this is not a class name; it is the unique name of the EJB that is specified in the `<ejb-name>` tag in the `ejb-jar.xml` file. See "ejbName" on page 18-45.

This command assembles the EJB 2.1 Web service by generating the WSDL and mapping files, and packaging the application into a deployable archive, `dist/helloServices-ejb.ear`. This archive contains the `helloService-ejb.jar`, which stores all of the service artifacts, such as the EJB implementation classes, the generated WSDL and mapping file, standard Web service descriptor file, `webservices.xml`, and the Oracle proprietary deployment descriptor file `oracle-webservices.xml`. For more information on the `ejbAssemble` command, see "ejbAssemble" on page 18-14.

4. Deploy the service and bind the application.

   EAR files are deployed into a running instance of OC4J. For more information on deployment, 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/ejbApp.ear
              -deploymentName ejbApp
              -bindWebApp default-web-site
   ```

   The following list describes the parameters in this code example.

   - *<OC4J_HOME>*—the directory containing the OC4J installation.
   - *<user>*—the user name for the OC4J instance. The user name is assigned at installation time.
   - *<password>*—the password for the OC4J instance. The password is assigned at installation time.
   - *ejbApp*—the name of the application.
   - `default-web-site`—the Web site to which the application will be bound. This is usually `default-web-site`. To configure Web sites, see the `server.xml` file in *<OC4J_HOME>*`/j2ee/home/config`.

   After deployment, this Web service's WSDL will be available at the following Web address. The values for *context-root* and *endpoint-address-uri* can be found in the `META-INF/oracle-webservices.xml` file.

   ```
   http://host:port/context-root/endpoint-address-uri
   ```

5. (Optional) Check that deployment succeeded.

   OracleAS Web Services provides a Web Service Test Page for each deployed Web service. See "How to Use the Web Services Test Page" on page 13-6 for information on accessing and using the Web Service Test Page.

6. Generate the client-side code:

   - For the J2SE environment, generate stubs (client proxies) for a J2SE Web service client by running the WebServicesAssembler tool with the `genProxy` command. For more information on generating and assembling a stub, see Chapter 15, "Assembling a J2SE Web Service Client".
   - For the J2EE environment, 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 a client for a J2EE Web service, see Chapter 14, "Assembling a J2EE Web Service Client".

   For example, the following command and Ant task generates stubs that can be used for a J2SE client. The client proxies are stored them in the directory `build/src/client`. The client application will use the stub to invoke operations on a remote service. For more information on the required and optional arguments to `genProxy`, see "genProxy" on page 18-34.

   Command line:

   ```
   java -jar wsa.jar -genProxy
                     -output build/src/client/
   ```

```
                    -wsdl http://localhost:8888/hello/HelloService?WSDL
                    -packageName oracle.demo.hello
```

Ant task:

```
<oracle:genProxy
          wsdl="http://localhost:8888/hello/HelloService?WSDL"
          output="build/src/client"
          packageName="oracle.demo.hello"
/>
```

In this command line and Ant task:

- `genProxy`—Creates a static proxy stub that can be used by a J2SE Web service client. See "genProxy" on page 18-34.
- `output`—Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See "output" on page 18-48.
- `wsdl`—Specifies the absolute file path, relative file path, or URL to a WSDL document. See "wsdl" on page 18-65.
- `packageName`—Specifies the package name that will be used for generated classes if no package name is declared in the JAX-RPC mapping file. See "packageName" on page 18-48.

7. Compile and run the client.

   List the appropriate JARs on the classpath before compiling the client. Table A–2, " Classpath Components for a Client Using a Client-Side Proxy" 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 Table A–2. See "Setting the Classpath for a Web Service Proxy" on page A-2 for more information on `wsclient_extended.jar` and the client classpath.

How to Write EJBs for Web Services

Writing EJB-based Web services using JAX-RPC involves writing an EJB that implements a service and providing an interface for it. The EJB should contain the business functions that OracleAS Web Services can invoke when a client makes a Web service request.

This section provides information on how to write an EJB Web service that returns a string, "`HELLO!! You just said:`*phrase*", where *phrase* is input from a client. The EJB Web service receives a client request with a single `String` parameter and generates a response that it returns to the Web service client.

Writing a J2EE 1.4-compliant EJB implementation for Web services consists of these tasks.

- How to Write an EJB Service Endpoint Interface
- How to Implement an EJB Interface for Web Services

How to Write an EJB Service Endpoint Interface

To use JAX-RPC to create a Web service with EJBs, you must write a public service endpoint interface to which the EJB must conform. The requirements for creating a service endpoint interface for a stateless session bean are summarized in Section 5.3.2.1

of the Enterprise Web Services 1.1 specification. The specification is available from the following Web site.

http://www.jcp.org/en/jsr/detail?id=921

The interface must extend `java.rmi.Remote`, and all methods must throw a `java.rmi.RemoteException`. You can use only Java primitives and classes that are JAX-RPC value types and nonstandard data types as parameters or return types for the EJB methods defined in the service endpoint interface.

Some examples of JAX-RPC value types are non-primitives such as `java.lang.String` or `java.lang.Double` and Java mappings of Multipurpose Internet Mail Extensions (MIME) types such as `java.awt.Image` or `javax.xml.transform.Source`.

You can use custom Java data types in the service endpoint interface, but you must also provide a serializer to process them.

See Also:

- "Mapping Java Types to XML and WSDL Types" in the *Oracle Application Server Advanced Web Services Developer's Guide* and in Table 5–1 on page 5-7 for descriptions of nonstandard data types.
- "Custom Serialization of Java Value Types" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on using custom data types and their serialization.

Example 8–1 illustrates a sample service endpoint interface.

Example 8–1 Sample Service Endpoint Interface

```
package oracle.demo.ejb;

import java.rmi.Remote;
import java.rmi.RemoteException;
/**
 * This is an Enterprise Java Bean Service Endpoint Interface
 */
public interface HelloServiceInf extends java.rmi.Remote {
    /**
     * @param phrase java.lang.String
     * @return java.lang.String
     * @throws String The exception description.
     */
    java.lang.String sayHello(java.lang.String phrase)
          throws java.rmi.RemoteException;
}
```

How to Implement an EJB Interface for Web Services

Create an Enterprise Java Bean by implementing the business functions that you want the Web service to expose.

The `HelloServiceBean` described in Example 8–2 is a sample session bean. The class defines a public method, `sayHello`, that returns `HELLO!! You just said:` *phrase*, where *phrase* was input from a client. In general, a Java bean for a Web service defines one or more public methods.

An Enterprise Java Bean, for the purposes of Web services, is any Java class that conforms to the following requirements:

- It must have a constructor that takes no arguments.
- All properties that you want to use must be exposed through accessors.

See Also:

"Mapping Java Types to XML and WSDL Types" in the *Oracle Application Server Advanced Web Services Developer's Guide* and in Table 5–1 on page 5-7 for descriptions of JAX-RPC supported data types for EJBs parameters and return types.

To comply with the JAX-RPC standard, all the methods in `HelloServiceBean` throw a `java.rmi.RemoteException`. They must also follow all the requirements of the version 2.1 EJB specification and Enterprise Web Services 1.1 specification.

Example 8–2 Sample HelloService Session Bean

```
package oracle.demo.ejb;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Properties;
import javax.ejb.*;

/**
 * This is a Session Bean Class.
 */
public class HelloServiceBean implements SessionBean {
    public String sayHello(String phrase) {
        return "HELLO!! You just said :" + phrase;
    }

    public void setSessionContext(javax.ejb.SessionContext ctx)
            throws java.rmi.RemoteException {
    }

    public void ejbActivate() throws java.rmi.RemoteException {
    }

    public void ejbCreate()
            throws javax.ejb.CreateException,
                   java.rmi.RemoteException {
    }
    public void ejbPassivate() throws java.rmi.RemoteException {
    }

    public void ejbRemove() throws java.rmi.RemoteException {
    }
}
```

Packaging and Deploying Web Services that Expose EJBs

The packaging structure of Web services that expose EJBs is described in "Packaging for a Web Service Based on EJBs" on page 19-3.

See Also:

Oracle Containers for J2EE Deployment Guide for a detailed description of the deployment of EJBs.

Note: WebServicesAssembler cannot detect whether the configuration of the 2.1 EJB is correct. This can be detected only during deployment and server-side code generation of the 2.1 EJB Web service.

How to Secure EJB-Based Web Services at the Transport-Level

You can use the `<ejb-transport-security-constraint>` and `<ejb-transport-login-config>` elements in the `oracle-webservices.xml` deployment descriptor to configure transport-level security constraints for a version 2.1 or 3.0 EJB. These elements are described in "Securing EJB-Based Web Services at the Transport Level" on page C-15.

See Also:

"Adding Transport-level Security for Web Services Based on EJBs" and "Accessing Web Services Secured on the Transport Level" in the *Oracle Application Server Web Services Security Guide* for more information on providing transport-level security for EJBs, and how to write clients to access Web services secured on the transport level.

Tool Support for Exposing EJBs as a Web Service

With Oracle JDeveloper, you can use modeling tools and wizards to create, modify, and deploy J2EE-compliant EJBs. The EJB wizards can be used to perform the following tasks.

- Create the Enterprise Bean class for several types of Enterprise JavaBeans, including stateless session beans.
- Generate the home interface needed to create an EJB object. The inclusion of the `ejbCreate()` method enables you to deploy the EJB to Oracle Applications Server immediately, without having to manually code the method.
- Enable a selection of home interface methods (and create a default method).
- Generate the remote interface.
- Enable a selection of remote interface methods.
- Deploy the EJB exposed as a Web service.

For more information on using Oracle JDeveloper to create EJBs and expose them as a Web service, see the Oracle JDeveloper on-line help.

Limitations

See "Assembling Web Services From EJBs" on page F-7.

Additional Information

For more information on:

- using the Test Page to test Web service deployment, see Chapter 13, "Testing Web Services Deployment".
- building a J2EE client, see Chapter 14, "Assembling a J2EE Web Service Client".
- building a J2SE client, see Chapter 15, "Assembling a J2SE Web Service Client".

- using JAX-RPC handlers, see Chapter 16, "Using JAX-RPC Handlers".
- using the WebServicesAssembler tool to assemble Web services, see Chapter 18, "Using WebServicesAssembler".
- packaging and deploying Web services, see Chapter 19, "Packaging and Deploying Web Services".
- JAR files that are needed to assemble a client, see Appendix A, "Web Service Client APIs and JARs".
- Web services interoperability, see "Ensuring Interoperable Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- using quality of service features in Web service clients, see "Managing Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- adding security to a Web service, see the *Oracle Application Server Web Services Security Guide*.
- adding transport-level security to Web services based on EJBs, see "Securing EJB-Based Web Services at the Transport Level" on page C-15.

  See also "Adding Transport-level Security to a Web Service" and "Accessing Web Services Secured on the Transport Level" in the *Oracle Application Server Web Services Security Guide*.
- adding reliability to a Web service, see "Ensuring Web Service Reliability" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- adding an auditing and logging configuration to a Web service, see "Auditing and Logging Messages" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- processing nonstandard data types, see "Custom Serialization of Java Value Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- the JAX-RPC mapping file, see "JAX-RPC Mapping File Descriptor" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- data types supported by OracleAS Web Services, see "Mapping Java Types to XML and WSDL Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- Oracle JDeveloper tool support for Web service development, see the Oracle JDeveloper on-line help.

9

Assembling Web Services with JMS Destinations

This chapter describes how to expose a JMS destination as a Web service. A JMS endpoint Web service exposes JMS destinations, either queues or topics, as document-literal style operations in the WSDL. The operation can be in either send or receive mode.

A JMS endpoint Web service can be considered to be a special case of the Java endpoint Web service. In a JMS endpoint Web service, the JMS endpoint implements Web service operations by sending and receiving JMS message objects.

Note: There are differences between a Web service based on queues (or Oracle Streams Advanced Queuing (AQ)) in the database and a Web service based on a JMS destination. The AQ Web service is based on a configuration of queues that reside in the database. The JMS destination Web service is based on the configuration of a JMS provider in the middle tier. The JMS queues reside in a backend data source. This data source could be a database, a file-based system, or some other data repository.

If you want to construct a Web service from a queue or an AQ in the database, see "How to Assemble a Web Service from an Oracle Streams AQ" on page 10-22.

If you want to use JMS as the transport mechanism to communicate with a Web service, see "Using JMS as a Web Service Transport" in the *Oracle Application Server Advanced Web Services Developer's Guide*.

Understanding JMS Endpoint Web Services

OracleAS Web Services enables you to create Web service endpoints that let you put messages on and take messages off JMS destinations. A JMS Web service endpoint is configured to transfer messages to and from a specific JMS destination or pair of destinations.

A JMS endpoint Web service can have the following operations.

- `send`—the XML payload (SOAP `body` element) is sent to the corresponding JMS destination. A send operation can be configured so that JMS message properties can be set on each sent message to indicate the JMS reply-to destination, priority, expiration, and so on.

- `receive`—a message is retrieved from the corresponding JMS destination, the content of the JMS message is used to create the SOAP response message body payload.
- both—a service can offer both operations.

A JMS endpoint Web service can be configured so that message-ID, correlation-ID, and reply-to-destination JMS message properties can be transmitted as SOAP headers. With this configuration, the message property headers and their types are explicitly declared on the generated WSDL and schema so that the Web service client can use them.

- If the destination is a JMS queue, then invoking the `send` operation means enqueue. Invoking the `receive` operation means dequeue.
- If the destination is a topic, then the `send` operation means publish and the `receive` operation means subscribe.

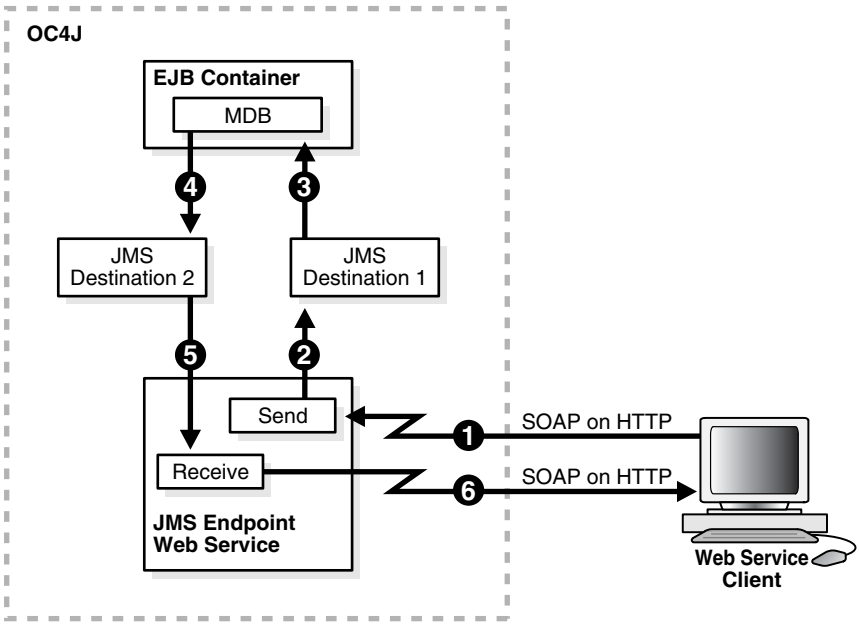
An individual JMS endpoint Web service can support just the `send` operation, just the `receive` operation, or both operations, as determined by the service developer.

JMS endpoint Web services use `javax.jms.ObjectMessage` as the JMS message type. As content, it can carry an instance of `javax.xml.soap.SOAPElement` or a `String` representation of an XML fragment.

The WSDL generated for a send JMS endpoint Web service follows the Web Service-Interoperability (WS-I) Basic Profile 1.0 and is interoperable.

Figure 9–1 shows an MDB-based JMS endpoint Web service application that, from the JMS endpoint Web service's view, handles both the message `send` and the message `receive` operations. The figure also includes an MDB that is configured to listen to a JMS destination.

Figure 9–1 MDB-Based JMS Endpoint Web Service



The following steps describe how the MDB-based JMS endpoint Web service application illustrated in Figure 9–1 works.

1. A Web service client sends a SOAP request to invoke the `send` operation on the JMS endpoint Web service.
2. The JMS endpoint Web service processes the incoming message and directs it to a JMS destination, JMS `Destination 1`.
3. The EJB container invokes the MDB listening on JMS `Destination 1`.
4. After processing the message an MDB produces a new message on JMS `Destination 2`. Producing and consuming messages could involve one or more MDBs. For example, a single MDB could be listing on JMS `Destination 1` and the same MDB could also send the message to JMS `Destination 2`.
5. (Arrows 5 and 6) A Web service client sends a SOAP request to perform a `receive` operation on the JMS endpoint Web service to retrieve a message. The JMS endpoint Web service consumes a message from the JMS destination, encloses it in a SOAP response message, and passes the outgoing SOAP response message to the client.

Steps for Assembling a JMS Endpoint Web Service

The following steps describe how to assemble a JMS Endpoint Web service with the WebServicesAssembler tool.

1. Generate the Web service EAR file by running the WebServicesAssembler with the `jmsAssemble` command or Ant task.

   For the `jmsAssemble` command, you must specify either the JNDI name of the JMS connection factory used to obtain a connection for the JMS `send` operation (`sendConnectionFactoryLocation`) or the JNDI name of the JMS connection factory to be used as the default reply-to of all `send` operation JMS messages (`replyToConnectionFactoryLocation`). For more information on the jmsAssemble command and its required and optional arguments, see "jmsAssemble" on page 18-16.

   In the following example, `jms/ws/mdb/theQueueConnectionFactory` is the JNDI name of the JMS connection factory used to produce connections to the JMS queue for the JMS `send` operation.

   - `jms/ws/mdb/theQueue`—the JNDI name of the JMS queue to which the `send` operation sends the SOAP message payload.
   - `jms/ws/mdb/logQueueConnectionFactory`—the JNDI name of the JMS connection factory to be used for the reply-to queue.
   - `jms/ws/mdb/logQueue`—the JNDI name of the JMS queue that will be set to each send message as the default reply-to destination. Because the `linkReceiveWithReplyTo` argument is enabled in this example, this reply-to destination is also used by the `receive` operation of the JMS endpoint Web service to retrieve messages.

   The J2EE EAR file produced by this command includes the JMS endpoint Web service configuration information, including the WSDL and the generated `web.xml` file.

   Command line:

```
java -jar wsa.jar -jmsAssemble
     -sendConnectionFactoryLocation jms/ws/mdb/theQueueConnectionFactory
     -sendQueueLocation jms/ws/mdb/theQueue
     -replyToConnectionFactoryLocation jms/ws/mdb/logQueueConnectionFactory
     -replyToQueueLocation jms/ws/mdb/logQueue
```

```
    -linkReceiveWithReplyTo true
    -targetNamespace http://oracle.j2ee.ws/jms-doc
    -typeNamespace http://oracle.j2ee.ws/jms-doc/types
    -serviceName JmsService
    -appName jms_service
    -context jms_service
    -input ./build/mdb_service.jar
    -uri JmsService
    -output ./dist
```

Ant task:

```
<oracle:jmsAssemble
        linkReceiveWithReplyTo="true"
        targetNamespace="http://oracle.j2ee.ws/jms-doc"
        typeNamespace="http://oracle.j2ee.ws/jms-doc/types"
        serviceName="JmsService"
        appName="jms_service"
        context="jms_service"
        input="./build/mdb_service.jar"
        uri="JmsService"
        output="./dist"
        sendConnectionFactoryLocation="jms/ws/mdb/theQueueConnectionFactory"
        sendQueueLocation="jms/ws/mdb/theQueue"

replyToConnectionFactoryLocation="jms/ws/mdb/logQueueConnectionFactory"
        replyToQueueLocation="jms/ws/mdb/logQueue"/>
```

In this command and Ant task:

- `jmsAssemble`—Exposes a JMS destination (queue or topic) as a Web service. See "jmsAssemble" on page 18-16.
- `sendConnectionFactoryLocation`—Specifies the JNDI name of the JMS `ConnectionFactory` used to obtain a connection for the JMS `send` operation. The type of `ConnectionFactory` must be consistent with the `send` destination. See "sendConnectionFactoryLocation" on page 18-60.
- `sendQueueLocation`—Specifies the JNDI name of the JMS queue to be used for the JMS `send` operation. No default value is provided. See "sendQueueLocation" on page 18-61.
- `replyToConnectionFactoryLocation`—Specifies the JNDI name of the JMS connection factory to be used as the default reply-to of all `send` operation JMS messages. The type of `ConnectionFactory` must be consistent with the reply-to destination. See "replyToConnectionFactoryLocation" on page 18-60.
- `replyToQueueLocation`—Specifies the JNDI name of the JMS queue to be used as the default reply-to of all `send` operation JMS messages. See "replyToQueueLocation" on page 18-60.
- `linkReceivedWithReplyTo`—Determines whether the `receive` operation will be linked to the reply-to destination. See "linkReceiveWithReplyTo" on page 18-59.
- `targetNamepace`—Specifies the target namespace to be used in the generated WSDL. The value can be either a conforming HTTP URL, a non-conforming HTTP URL, or even a URI. See "targetNamespace" on page 18-66.

- `typeNamespace`—Specifies the type namespace to be used in the schema types in the generated WSDL. The name that you specify will always be used and it will not be reversed. See "typeNamespace" on page 18-67.
- `serviceName`—Specifies the service name. See "serviceName" on page 18-50.
- `appName`—Specifies the name of an application. Usually, this name is used as a base value for other arguments like `context` and `uri`. See "appName" on page 18-42.
- `context`—Specifies the root context for the web application. See "context" on page 18-62.
- `input`—Specifies the directory or JAR containing the classes that should be copied to `WEB-INF/classes`. This argument will be added to the classpath used by the WebServicesAssembler. See "input" on page 18-46.
- `uri`—Specifies the URI to use for the Web service. See "uri" on page 18-63.
- `output`— Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See "output" on page 18-48.

2. Deploy all of the JMS destinations.
3. Deploy the service and bind the application.

   Deploy the EAR file in the standard manner into a running instance of OC4J. For more information on deploying EAR files, see Chapter 19, "Packaging and Deploying Web Services" and 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/jms_service.ear
              -deploymentName jms_service
              -bindWebApp default-web-site
   ```

   The following list describes the parameters in this code example.

   - *<oc4jHome>*—The directory containing the OC4J installation.
   - *<user>*—The user name for the OC4J instance. The user name is assigned at installation time.
   - *<password>*—The password for the OC4J instance. The password is assigned at installation time.
   - `default-web-site` —The Web site to which the application will be bound. This is usually `default-web-site`. To configure Web sites, see the `server.xml` file in *<OC4J_HOME>*`/j2ee/home/config`.

4. (Optional) Check that deployment succeeded.

   OracleAS Web Services provides a Web Service Test Page for each deployed Web service. See "How to Use the Web Services Test Page" on page 13-6 for information on accessing and using the Web Service Test Page.

5. Generate the client-side code.

   Generating a client-side proxy from a JMS endpoint Web service WSDL is the same as generating the proxy from any other WSDL. The JMS endpoint Web service WSDL is interoperable, in that it should be consumed by WS-I Basic Profile 1.0-compliant WSDL tools. For example, you can use a .NET WSDL tool to

generate C# client stubs to communicate with an Oracle JMS endpoint Web service.

- For the J2SE environment, generate stubs (client proxies) for a J2SE Web service client by running the WebServicesAssembler tool with the `genProxy` command. For more information on generating and assembling client-side code for the J2SE environment, see Chapter 15, "Assembling a J2SE Web Service Client".
- For the J2EE environment, generate a service endpoint interface and a 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 Chapter 14, "Assembling a J2EE Web Service Client".

For example, the following command generates stubs that can be used for a J2SE client:

Command line:

```
java -jar wsa.jar -genProxy
                  -output build/src/client/
                  -wsdl http://localhost:8888/hello/JmsService?WSDL
                  -packageName oracle.demo.jms_service
```

Ant task:

```
<oracle:genProxy  wsdl="http://localhost:8888/hello/JmsService?WSDL"
           output="build/src/client"
           packageName="oracle.demo.jms_service"
/>
```

In this command line and Ant task:

- `genProxy`—Creates a static proxy stub that can be used by a J2SE Web service client. See "genProxy" on page 18-34.
- `output`—Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See "output" on page 18-48.
- `wsdl`—Specifies the absolute file path, relative file path, or URL to a WSDL document. See "wsdl" on page 18-65.
- `packageName`—Specifies the package name that will be used for generated classes if no package name is declared in the JAX-RPC mapping file. See "packageName" on page 18-48.

This command generates the client proxies and stores them in the directory `build/src/client`. The client application uses the stub to invoke operations on a remote service. For more information on the required and optional arguments to `genProxy`, see "genProxy" on page 18-34.

6. Compile and run the client.

   List the appropriate JARs on the classpath before compiling the client. Table A–2, " Classpath Components for a Client Using a Client-Side Proxy" 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 Table A–2. See "Setting the Classpath for a Web Service Proxy" on page A-2 for more information on `wsclient_extended.jar` and the client classpath.

Message Processing and Reply Messages

The JMS endpoint Web service processes an incoming SOAP message and places the payload (the `body` element of the SOAP message) on a JMS destination. This section covers details that a developer needs to know to consume and process the JMS messages that originate from a JMS endpoint Web service.

The JMS message content associated with a JMS endpoint Web service can be either an instance of `javax.xml.soap.SOAPElement` (which is also a subclass of `org.w3c.dom.Element`), or `java.lang.String` which is the string representation of the XML payload. The JMS endpoint Web service may set certain JMS message header values before it places the message on a JMS destination. Depending on the values of optional configuration arguments specified when the JMS endpoint Web service is assembled, the JMS endpoint Web service sets the following JMS message headers.

```
JMSType
JMSReplyTo
JMSExpiration
JMSPriority
JMSDeliveryMode
```

The JMS endpoint Web service sets the `JMSReplyTo` header with the value specified by the `replyToTopicLocation` or the `replyToQueueLocation` argument (only one of these should be configured for any given JMS endpoint Web service). The value specified with the `replyToConnectionFactoryLocation` argument is set on the message as a standard string property. The property name is `OC4J_REPLY_TO_FACTORY_NAME`.

Example 9–1 provides a code segment that shows where the `onMessage()` method gets the `reply-to` information for a message generated from a JMS endpoint Web service `send` operation:

Example 9–1 Getting Reply-To Information for a Message Generated by a Send Operation

```
...
public void onMessage(Message inMessage) {
    // Do some processing
    ObjectMessage msg = null;
    String  factoryName;
    Destination dest;
    Element  el;
    try {
      // Message should be of type objectMessage
      if (inMessage instanceof ObjectMessage) {
        // retrieve the object
        msg = (ObjectMessage) inMessage;
        el = (Element)msg.getObject();
        System.out.println("MessageBean2::onMessage() => Message received: " );
        ((XMLElement)el).print(System.out);
        processElement(el);
        factoryName = inMessage.getStringProperty("OC4J_REPLY_TO_FACTORY_NAME");
        dest = inMessage.getJMSReplyTo();
...
```

Limitations

See "Assembling Web Services with JMS Destinations" on page F-7.

Additional Information

For more information on:

- using the Test Page to test Web service deployment, see Chapter 13, "Testing Web Services Deployment".
- building J2EE Web service clients, see Chapter 14, "Assembling a J2EE Web Service Client".
- building J2SE Web service clients, see Chapter 15, "Assembling a J2SE Web Service Client".
- JAX-RPC handlers, see Chapter 16, "Using JAX-RPC Handlers".
- using the WebServicesAssembler tool to assemble Web services, see Chapter 18, "Using WebServicesAssembler".
- packaging and deploying Web services, see Chapter 19, "Packaging and Deploying Web Services".
- JAR files that are needed to assemble a client, see Appendix A, "Web Service Client APIs and JARs".
- Web services interoperability, see "Ensuring Interoperable Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- using quality of service features in Web service clients, see "Managing Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- adding security to a Web service, see the *Oracle Application Server Web Services Security Guide*.
- adding reliability to a Web service, see "Ensuring Web Service Reliability" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- adding an auditing and logging configuration to a Web service, see "Auditing and Logging Messages" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- processing nonstandard data types, see "Custom Serialization of Java Value Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- the JAX-RPC mapping file, see "JAX-RPC Mapping File Descriptor" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- data types supported by OracleAS Web Services, see "Mapping Java Types to XML and WSDL Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- Oracle JDeveloper tool support for Web service development, see the Oracle JDeveloper on-line help.

10

Assembling Database Web Services

This chapter describes how you can develop Oracle Application Server Web Services that interact with an Oracle database. There are two ways in which OracleAS Web Services can interact with a database: call-in and call-out. Web service call-in refers to providing a Web service that provides access to a database resource. The Web service runs in an OC4J instance and accesses the database resource through JDBC.

Web services call-out refers to invoking an external Web service from inside the database. The invocation can be performed by SQL, PL/SQL, or Java code running in the database.

This chapter contains these sections.

- Understanding Database Web Services
- Type Mapping Between SQL and XML
- Developing Web Services that Expose Database Resources
- Developing a Web Service Client in the Database
- Tool Support for Web Services that Expose Database Resources

Understanding Database Web Services

Web service technology enables application-to-application interaction over the Web—regardless of platform, language, or data formats. The key ingredients, including XML, SOAP, WSDL, UDDI, WS-Security, and WS-Reliability have been adopted across the entire software industry. Web service technology usually refers to services implemented and deployed in middle-tier application servers. However, in heterogeneous and disconnected environments, there is an increasing need to access stored procedures, data and metadata, through Web service interfaces. Database Web service technology is a database approach to Web services. It works in two directions:

- accessing database resources as a Web service (database call-in)
- consuming external Web services from the database itself (database call-out)

Database Call-In

Turning the Oracle database into a Web service provider takes advantage of your investment in Java stored procedures, PL/SQL packages, Advanced Queues, pre-defined SQL queries and DML. Client applications can query and retrieve data from Oracle databases and invoke stored procedures using standard Web service protocols. There is no dependency on Oracle specific database connectivity protocols. Applications can employ any cached OC4J connection. This approach is very beneficial in heterogeneous, distributed, and non-connected environments.

Since database Web services are a part of OracleAS Web Services, they can participate in a consistent and uniform development and deployment environment. Messages exchanged between the Web service exposed database and the Web service client can take advantage of all of the management features provided by OracleAS Web Services, such as security, reliability, auditing and logging.

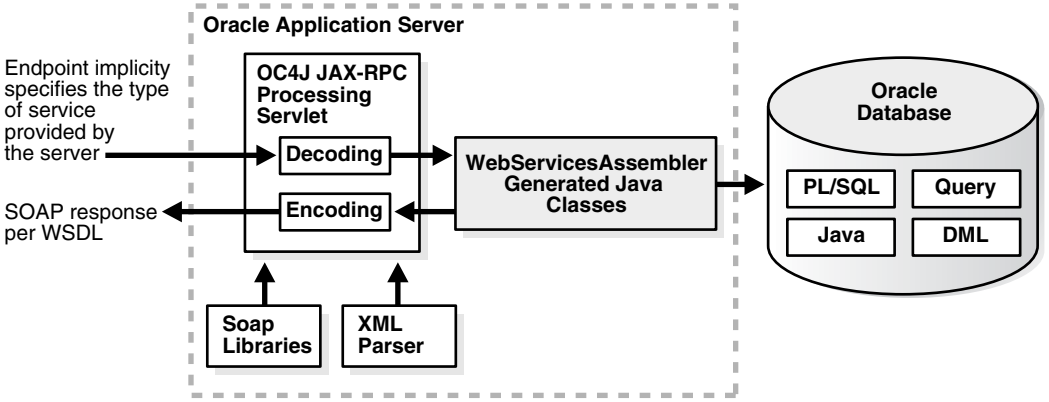
Figure 10–1 illustrates Web service call-in. The following steps describe the process.

1. A request for a type of database service arrives at the application server. The service endpoint implicitly specifies the type of service requested.
2. The OC4J JAX-RPC processing servlet references the SOAP libraries and XML parser to decode the request.
3. The servlet passes the request to the WebServicesAssembler-generated classes that correspond to the exposed database operations. WebServicesAssembler generates these classes by silently calling Oracle JPublisher. The generated classes can represent PL/SQL packages, queries, DML, AQ Streams, or Java classes in the database.
4. The database passes the response to the OC4J JAX-RPC processing servlet, which references the SOAP libraries and XML parser to encode it.
5. A SOAP response formed in accordance with the WSDL is returned to the client.

See Also:

"Developing Web Services that Expose Database Resources" on page 10-6 for more information on exposing PL/SQL packages, SQL queries, DML statements, Oracle AQ Streams, or server-side Java classes database operations as a Web service.

Figure 10–1 Web Service Calling-In to the Database



Database Call Out

You can extend a relational database's storage, indexing, and searching capabilities to include Web Services. By calling a Web service, the database can track, aggregate, refresh, and query dynamic data produced on-demand, such as stock prices, currency exchange rates, or weather information.

An example of using the database as a service consumer would be to call an external Web service from a predefined database job to obtain inventory information from multiple suppliers, then update your local inventory database. Another example is that of a Web Crawler: a database job can be scheduled to collate product and price information from a number of sources.

Figure 10–2 illustrates database call out.

- SQL and PL/SQL call specs—Invoke a Web service through a user-defined function call (generated through Oracle JPublisher) either directly within a SQL statement or view or through a variable.
- Dynamic Web service invocation using the `UTL_DBWS` PL/SQL package. A `Call` object can be dynamically created based on a WSDL and subsequently, Web services operations can be invoked.

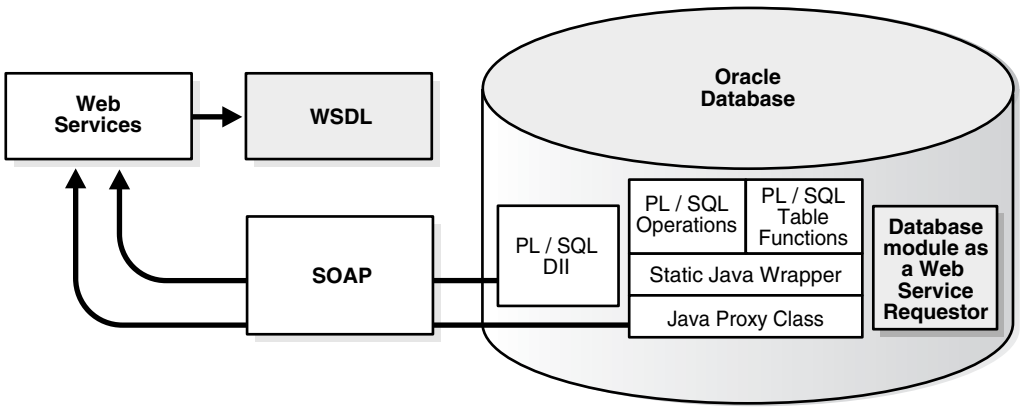
  Oracle Database PL/SQL Packages and Types Reference provides more information on using the `UTL_DBWS` PL/SQL package.
- Pure Java static proxy class—Use Oracle JPublisher to pre-generate a client proxy class which uses JAX-RPC. This method simplifies the Web service invocation as the location of the service is already known without needing to look up the service in the UDDI registry. The client proxy class does all of the work to construct the SOAP request, including marshalling and unmarshalling parameters.
- Pure Java using DII (dynamic invocation interface) over JAX-RPC—Dynamic invocation provides the ability to construct the SOAP request and access the service without the client proxy.

Which method to use depends on whether you want to execute from SQL or PL/SQL, from Java classes, or whether the service is known ahead of time (static invocation) or only at runtime (DII).

See Also:

"Developing a Web Service Client in the Database" on page 10-36 for more information about the support OracleAS Web Services offers for PL/SQL and Java call-outs from the database.

Figure 10–2 Calling Web Services From Within the Database



Type Mapping Between SQL and XML

The following sections describe the type mappings between SQL and XML for call-ins and call-outs when the Web service is known ahead of time (static invocation).

- SQL to XML Type Mappings for Web Service Call-Ins
- XML to SQL Type Mapping for Web Service Call-Outs

When the Web service is known at runtime you can use only the Dynamic Invocation Interface (DII) or the `UTL_DBWS` PL/SQL package. For more information on using the JAX-RPC DII, see the API at the following Web address.

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

See Also:

Oracle Database PL/SQL Packages and Types Reference for more information on using the `UTL_DBWS` package.

SQL to XML Type Mappings for Web Service Call-Ins

In a database Web service call-in, a SQL operation, such as a PL/SQL stored procedure or a SQL statement, is mapped into one or more Web service operations. The parameters to the SQL operation are mapped from SQL types into XML types.

Note: The reason there may be more than one operation is because OracleAS Web Services may be providing additional data representation choices for the SQL values in XML, such as different representations of SQL result sets.

Table 10–1 illustrates the SQL to XML mappings for Web service call-ins. The first column lists the SQL types. The second column of the table, **XML Type (Literal)**, shows SQL-to-XML type mappings for the default `literal` value of the `use` attribute. The third column, **XML Type (Encoded)**, shows the mappings for the `encoded` value of the `use` attribute. The `literal` and `encoded` values refer to the rules for encoding the body of a SOAP message.

See Also:

"Literal and Encoded Uses" on page 5-2 for more information on the rules for encoding the body of a SOAP message.

Table 10–1 SQL-to-XML Type Mappings for Web Services Call-Ins

SQL Type	XML Type (Literal)	XML Type (Encoded)
INT	int	int
INTEGER	int	int
FLOAT	double	double
NUMBER	decimal	decimal
VARCHAR2	string	string
DATE	dateTime	dateTime
TIMESTAMP	dateTime	dateTime
BLOB	byte[]	byte[]
CLOB	String	String
LONG	String	String
RAW	byte[]	byte[]
Primitive PL/SQL indexby table	Array	Array
PL/SQL Boolean	boolean	boolean

Table 10–1 (Cont.) SQL-to-XML Type Mappings for Web Services Call-Ins

SQL Type	XML Type (Literal)	XML Type (Encoded)
PL/SQL indexby table	complexType	complexType
PL/SQL record	complexType	complexType
REF CURSOR (*name*Beans)	Array	Array
REF CURSOR (*name*XML)	any	text_xml
REF CURSOR (*name*XMLRowSet)	swaRef	text_xml
SQL object	complexType	complexType
SQL table	complexType	complexType
SYS.XMLTYPE	any	text_xml

Note: If National Language Support (also known as "NLS" or "Globalization Support") characters are used in a `SQL SYS.XMLTYPE` value, they may not be properly handled.

A query or a PL/SQL function returning `REF CURSOR` will be mapped into three methods, *name*`Beans`, *name*`XMLRowSet`, and *name*`XML`, where *name* is the name of the query or the PL/SQL function.

- *name*`Beans`—this method returns an array, where each element is an instance of an XSD complex type that represents one row in the cursor. A complex type subelement corresponds to a column in that row.
- *name*`XMLRowSet`—this method returns a `swaRef` or `text_xml` response that contains an `OracleWebRowSet` instance in XML format. "Working with MIME Attachments" in the *Oracle Application Server Advanced Web Services Developer's Guide* provides more information on the `swaRef` MIME format.
- *name*`XML`—this method returns an XML `any` or `text_xml` response that contains an Oracle `XDB` row set.

Both `OUT` and `IN OUT` PL/SQL parameters are mapped to `IN OUT` parameters in the WSDL file.

See Also:

- "How to Assemble a Web Service from a SQL Query or DML Statement" on page 10-15 for examples of returning a result set which is similar to `REF CURSOR`.
- "How to Access PL/SQL IN and IN OUT Parameters from Client Code" on page 10-13 for an example of how PL/SQL parameters are generated into the WSDL and then accessed from client code.

Note that Table 10–1 provides two different mappings: one for literal and another for encoded `use`. The default mapping is literal. From a database Web service's perspective, there is no special reason why encoded should be used. The mapping for encoded is provided in case you encounter scenarios which call for the encoded `use`

setting. All of the descriptions in this chapter assume that you will be using the literal `use` setting unless otherwise specified.

Changing the SQL to XML Mapping for Numeric Types

Table 10–1 defines SQL to XML type mappings used for call-ins. The mappings for the numeric types is determined by how Oracle JPublisher maps SQL types to Java types. By default, the WebServicesAssembler tool uses the Oracle JPublisher option `-numbertypes=objectjdbc`. As a result, the XML types corresponding to the SQL numeric types are all declared nillable in the generated WSDL file.

To change the Oracle JPublisher mappings, and hence change the XML types, you can use the WebServicesAssembler `jpubProp` argument. For example, if you specify the following argument in the database WebServicesAssembler Ant task, then the XML types generated for SQL numeric types will not be declared nillable.

```
jpubProp="numbertypes=jdbc"
```

On the other hand, if you specify either `oracle` or `bigdecimal` as the target of the `numbertypes` option, then the XML types generated for SQL numeric types will all be `decimal` and nillable.

XML to SQL Type Mapping for Web Service Call-Outs

In database Web services call-outs, XML types are mapped into SQL types. Table 10–2 lists the XML-to-SQL type mappings used in call-outs.

Table 10–2 XML-to-SQL Type Mappings for Web Services Call-Outs

XML Type	SQL Type
int	NUMBER
float	NUMBER
double	NUMBER
decimal	NUMBER
dateTime	DATE
String	VARCHAR2
byte[]	RAW
complexType	SQL OBJECT
Array	SQL TABLE
text_xml	XMLType

Developing Web Services that Expose Database Resources

This section describes how to develop Web services implemented as PL/SQL stored procedures, SQL statements, Oracle Streams AQ queues, and server-side Java classes.

- How to Use Life Cycle for Web Service Call-in
- WebServicesAssembler Support for Web Service Call-in
- How to Assemble a Web Service from a PL/SQL Package
- How to Assemble a Web Service from a SQL Query or DML Statement
- How to Assemble a Web Service from an Oracle Streams AQ

- How to Assemble a Server-Side Java Class as a Web Service

How to Use Life Cycle for Web Service Call-in

Creating a database Web service call-in application is a bottom up process. In many cases, you will want to reuse existing database applications (such as PL/SQL packages or Java applications) or operational scripts (such as SQL query, DML, or AQ). You can also populate the database with the resources to be exposed as a Web service. Web service call-ins typically follow these steps:

1. Determine which database resources to expose and make them available.

   For example, you can provide the resources in any of the following ways.

   - load the PL/SQL package into the database
   - create the schema used by SQL query or DML statement Web service
   - load the Java class into the database, for a database server-side Java Web service

2. Run the WebServicesAssembler tool to assemble the Web service, based on the specified resources.

   Note that you could also use Oracle JDeveloper to assemble the Web service.

3. If the Web service assembly generates a PL/SQL wrapper, load it into the database.

   PL/SQL Web service assembly may generate a wrapper if a PL/SQL record or `INDEX BY` table type is included in the PL/SQL package. You must load the wrapper into the database. Server-side Java call-in assembly always generates a PL/SQL wrapper. WebServicesAssembler will load the wrapper automatically if the `sysUser` argument is set. For more information, see sysUser on page 18-57.

   Note that you could also use Oracle JDeveloper to load the PL/SQL wrapper into the database.

4. Configure the OC4J data sources to ensure that the Oracle JPublisher-generated Java classes that constitute the Web service implementation can connect to the database and the resource it is exposing.

   Add a data source entry in the J2EE data source file, so that the Web service application can connect to the database.

5. Deploy the Web service application into a running instance of OC4J.

6. (Optional) Check that deployment succeeded. OracleAS Web Services provides a Web Service Test Page for each deployed Web service.

   See "How to Use the Web Services Test Page" on page 13-6 for information on accessing and using the Web Service Test Page.

7. Use the WebServicesAssembler tool to generate the Web services client proxy and incorporate it into your client application.

8. Invoke the Web service using the client proxy.

WebServicesAssembler Support for Web Service Call-in

For all Web service call-in types, the WebServicesAssembler tool requires the following arguments.

- appName—the application name

- dataSource—the data source JNDI name; used at runtime
- dbConnection—the database connection URL; used at code generation time
- dbUser—the database user and password; used at code generation time

The `appName` argument specifies the Web service application name. The `dataSource` argument defines the data source's JNDI location for the database being accessed. At runtime, the Web service code accesses the database through that data source.

The WebServicesAssembler tool uses the `dbConnection` and `dbUser` arguments to define the connection to the database during Web service creation. The values of these arguments are used at code generation time, not at runtime. The `dbConnection` argument is used to access the information on the resource that is to be exposed. For example, it accesses information about the PL/SQL package, the schema, or the query.

The database used at Web service creation time and the one used at runtime do not have to be the same database. However, both should include the schema that contains the database resources being exposed.

At runtime, database Web services obtain the database connection from the data source. The database Web service deals with connection loss by reconnecting to the database. When the connection is in an invalid state, the Web service will attempt to reestablish the connection. If the Web service fails to re-connect, it will return a fault. The next time the client invokes the Web service, the Web service will attempt to connect to the database. Therefore, it is possible for the Web service to return a fault at one moment, but succeed later, due to connection failure handling.

The following arguments are optional and can be used in all Web service call-in scenarios.

- context—root context for the web application
- debug—displays detailed diagnostic messages
- ear—name and location of the generated EAR
- jpubProp—specifies Oracle JPublisher options to fine-tune Oracle JPublisher code generation
- output—location for storing generated files
- portName—the name of the port in the generated WSDL
- serviceName—local part of the service name in the generated WSDL
- style—the `style` part of the message format used in the generated WSDL
- uri—URI to use for the Web service in the deployment descriptors
- use—the `use` part of the message format used in the generated WSDL

The common prerequisite for all call-in types is that the database is populated with the resource to be exposed. The WebServicesAssembler employs Oracle JPublisher to generate Java code to access database resources. The `jpubProp` argument, which can appear more than once on the command line or in an Ant task, lets you pass options to Oracle JPublisher.

See Also:

Oracle Database JPublisher User's Guide for the list of Oracle JPublisher options and for information on how it maps PL/SQL, SQL types, SQL statements, and server-side Java into client-side Java wrappers.

How to Assemble a Web Service from a PL/SQL Package

Use the `plsqlAssemble` command to assemble Web services from a PL/SQL stored procedure. In the generated Web service, each Web service operation corresponds to a PL/SQL stored procedure.

This section has the following subsections:

- Prerequisites
- Steps for Assembling a Web Service from a PL/SQL Package
- Sample PL/SQL Package
- How PL/SQL Function Names are Mapped to Web Service Operation Names
- How to Access PL/SQL IN and IN OUT Parameters from Client Code
- How to Access SQL XMLType Types from Client Code

Prerequisites

Before you begin, provide the following files and information.

- The PL/SQL package that you want to expose as a Web service

  See Also:

  "Sample PL/SQL Package" on page 10-12 for the stored procedure that is used in this example.

- A name for the Web service application
- A JNDI location for the JDBC data source
- The JDBC database connection URL, and the username and password

Steps for Assembling a Web Service from a PL/SQL Package

The following steps describe how to assemble a Web service for the PL/SQL package `echo_plsql`.

1. Provide the PL/SQL package and the information described in the Prerequisites section as input to the WebServicesAssembler `plsqlAssemble` command.

   Command line:

   ```
   java -jar wsa.jar
       -plsqlAssemble
       -appName Echo
       -sql echo_plsql
       -dataSource jdbc/OracleManagedDS
       -dbConnection jdbc:oracle:thin:@stacd15:1521:lsqlj
       -dbUser scott/tiger
       -style rpc
       -use encoded
   ```

   Ant task:

   ```
   <oracle:plsqlAssemble
       dbUser="scott/tiger"
       sql="echo_plsql"
       dbConnection="jdbc:oracle:thin:@stacd15:1521:1sqlj"
       dataSource="jdbc/OracleManagedDS"
       appName="EchoPlsql"
   ```

```
    style="rpc"
    use="encoded"
/>
```

In this command and Ant task:

- `plsqlAssemble`—Generates Web services from a PL/SQL package containing stored procedures and functions. To use this command, you must connect to a database. See "plsqlAssemble" on page 18-17.
- `appName`—Specifies the name of an application. Usually, this name is used as a base value for other arguments like `context` and `uri`. See "appName" on page 18-42.
- `sql`—Specifies a PL/SQL package name. See "sql" on page 18-55.
- `dataSource`—Specifies the JNDI location of the data source used by the Web services at runtime. See "dataSource" on page 18-54.
- `dbConnection`—Specifies the JDBC URL for the database. See "dbConnection" on page 18-54.
- `dbUser`—Specifies the database schema and password in the form of *user*/*password*. See "dbUser" on page 18-55.
- `style`—For bottom up Web service assembly, this argument specifies the `style` attribute of the message format in the generated WSDL. See "style" on page 18-68.
- `use`—For bottom up Web service assembly, this argument specifies the `use` attribute of the message format in the generated WSDL. See "use" on page 18-68.

The arguments that can be used with the `plsqlAssemble` command are described in "plsqlAssemble" on page 18-17.

By default, WebServicesAssembler generates services using document-wrapped style. However, JAX-RPC clients that use document-wrapped style do not support `IN OUT` parameters directly. Instead, WebServicesAssembler packages `IN` and `IN OUT` parameters separately. Since the PL/SQL package used in this example contains `IN OUT` parameters, the `plsqlAssemble` command includes the `-style rpc` argument. For more information on parameters and different document styles, see "How to Access PL/SQL IN and IN OUT Parameters from Client Code" on page 10-13.

The command generates a Web service application, `EchoPlsql.ear`, and optionally, the following PL/SQL scripts.

- `Echo_plsql_wrapper.sql`—the PL/SQL wrapper generated to support PL/SQL `record` and `INDEX BY` table.
- `Echo_plsql_dropper.sql`—the PL/SQL script to tear down the types and packages created by the wrapper script.

2. Install any PL/SQL wrappers created during Web service generation into the database.

   Not all PL/SQL Web services assembly generates PL/SQL wrappers. If it does, you must load them into the appropriate user schema in the database before running the Web service.

   The wrappers can be loaded automatically or manually. To load the wrappers automatically, add the following line to the `plsqlAssemble` command:

`-jpubProp plsqload` (for the command line), or

`jpubprop="plsqlload"` (for an Ant task)

To manually load the wrapper package after Web service assembly, use SQL*PLUS. The following command line provides a sample SQL*PLUS command to load a wrapper package.

```
SQL>@Echo_plsql_wrapper.sql
```

3. Deploy the service into a running instance of OC4J and bind the application.

   The data source referenced by the `-dataSource` argument in Step 1 must be set up in this OC4J instance.

   The following command lines provide sample deployment and bind commands.

   ```
   % java -jar <J2EE_HOME>/admin_client.jar deployer:oc4j:localhost:port admin
   welcome -deploy -file dist/echo.ear -deploymentName echo

   % java -jar <J2EE_HOME>/admin_client.jar deployer:oc4j:localhost:port admin
   welcome -bindWebApp plsql plsql-web default-web-site /plsql
   ```

   In this example, <*J2EE_HOME*> is the directory where J2EE is installed.

   For more information on deploying EAR files, see the *Oracle Containers for J2EE Deployment Guide*.

4. (Optional) Check that deployment succeeded.

   OracleAS Web Services provides a Web Service Test Page for each deployed Web service. See "How to Use the Web Services Test Page" on page 13-6 for information on accessing and using the Web Service Test Page.

5. Generate the client-side code.

   - For the J2SE environment, generate stubs (client proxies) for a J2SE Web service client by running the WebServicesAssembler tool with the `genProxy` command. See Chapter 15, "Assembling a J2SE Web Service Client" for more information on generating and assembling client-side code for the J2SE environment.
   - For the J2EE environment, 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. See Chapter 14, "Assembling a J2EE Web Service Client" for more information on generating and assembling client-side code.

   For example, the following command uses the `genProxy` command to generate a J2SE client in the `build/classes/client` directory.

   ```
   % java -jar wsa.jar -genProxy
                       -wsdl http://locahost:8888/plsql/echo?WSDL
                       -output build/src/client
                       -mappingFileName ./mapping.xml
                       -packageName oracle.demo.db.plsql.stub
                       -unwrapParameters true
   ```

   In this command:

   - `genProxy`—Creates a static proxy stub that can be used by a J2SE Web service client. See "genProxy" on page 18-34.

- `wsdl`—Specifies the absolute file path, relative file path, or URL to a WSDL document. See "wsdl" on page 18-65.
- `output`—Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See "output" on page 18-48.
- `mappingFileName`—Specifies a file location that points to a JAX-RPC mapping file. See "mappingFileName" on page 18-47.
- `packageName`—Specifies the package name that will be used for generated classes if no package name is declared in the JAX-RPC mapping file. See "packageName" on page 18-48.
- `unwrapParameters`—This argument can be set only for document-literal operations and will be ignored for other message formats. When `unwrapParameters` is set to `true`, which is the default, the return type and response type will be unwrapped. This is usually easier to use, especially if the types are simple. See "unwrapParameters" on page 18-69.

6. Compile and run the client.

   List the appropriate JARs on the classpath before compiling the client. Table A–2, " Classpath Components for a Client Using a Client-Side Proxy" 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 Table A–2. See "Setting the Classpath for a Web Service Proxy" on page A-2 for more information on `wsclient_extended.jar` and the client classpath.

   The following command line provides sample compile and run commands.

```
% javac -classpath path:
        <ORACLE_HOME>/webservices/lib/wsclient_extended.jar:
        :<ORACLE_HOME>/webservices/lib/jaxrpc-api.jar EchoClient.java

% java -classpath path:
       <ORACLE_HOME>/webservices/lib/wsclient_extended.jar:
       <ORACLE_HOME>/webservices/lib/jaxrpc-api.jar:
       <J2EE_HOME>/lib/jax-qname-namespace.jar:
       <J2EE_HOME>/lib/activation.jar:
       <J2EE_HOME>/lib/mail.jar:
       <J2EE_HOME>/lib/http_client.jar:
       <ORACLE_HOME>/lib/xmlparserv2.jar EchoClient
```

   In this example, <*J2EE_HOME*> is the directory where J2EE is installed, <*ORACLE_HOME*> is the directory where OracleAS Web Services is installed.

Sample PL/SQL Package

Example 10–1 illustrates a PL/SQL package in the database that can be exposed as a Web service. The package contains procedures and functions that exercise various SQL and PL/SQL data types.

Example 10–1 Sample PL/SQL Package

```
create or replace type address as object(
     street varchar2(30), city varchar2(30), state varchar2(2), zip varchar2(5));
create or replace type employee as object (eid int, efirstname varchar(30),
     elastname varchar(30), addr address, salary float);
create table employees (eid int, emp employee);
create table employee_espp (eid int, status int);
```

```
create table employee_accounts (eid int, accounts sys.xmltype);
create table employee_biodata (eid int, biodata CLOB);
create table employee_image (eid int, image BLOB);
create table employee_hiredate(eid int, hiredate TIMESTAMP);

create or replace package echo_plsql as
procedure set_object (emp IN employee);
function get_object1(id IN int) return employee;
function get_object2(id IN int) return address;
function hold_varchar(id IN int, firstname OUT varchar2, lastname OUT varchar2)
return float;
procedure set_boolean(id IN int , status IN boolean);
function get_boolean(id IN int) return boolean;
procedure hold_float_inout(id IN int, newsalary IN OUT float);
procedure clear_object (id IN int);
procedure set_clob (id int, biodata IN CLOB);
function get_clob(id IN int) return CLOB;
procedure set_blob(id int, image IN BLOB);
function get_blob(id IN int) return BLOB;
procedure set_xmltype(id IN number, accounts sys.xmltype);
function get_xmltype(id IN number) return sys.xmltype;
procedure set_date(id IN int, hiredate IN TIMESTAMP);
function get_date(id IN int) return TIMESTAMP;
TYPE rec is RECORD (emp_id int, manager_id int);
TYPE index_tbl is TABLE OF rec INDEX BY BINARY_INTEGER;
function echo_rec(mrec rec) return rec;
function echo_index_tbl(mtbl index_tbl) return index_tbl;
end echo_plsql;
```

How PL/SQL Function Names are Mapped to Web Service Operation Names

WebServicesAssembler maps PL/SQL functions or procedures into Web service operations, often with adjusted names. Typically, the underscore in a PL/SQL name is removed and the letter following the underscore is capitalized. For example, notice the PL/SQL function `echo_index_tbl` in Example 10–1. This function is mapped into the Web service operation `echoIndexTbl`. The WSDL fragment in Example 10–2 shows how the PL/SQL function `echo_index_tbl` is expressed as the `echoIndexTbl` Web service operation.

Example 10–2 WSDL Fragment, Illustrating the Mapping of a PL/SQL Function

```
<operation name="echoIndexTbl" parameterOrder="EchobaseIndexTblBase_1">
<input message="tns:Echo_echoIndexTbl"/>
<output message="tns:Echo_echoIndexTblResponse"/>
</operation>
```

How to Access PL/SQL IN and IN OUT Parameters from Client Code

The PL/SQL parameters `OUT` and `IN OUT` in Example 10–1 on page 10-12 are represented as XML `IN OUT` parameters, as shown by the holder parameters of `holdVarchar`. The entries in the WSDL fragment in Example 10–3 illustrate the `holdVarchar` operation. The second and third parameters appear in both the input and output messages, which indicates that both parameters are IN OUT parameters.

Example 10–3 WSDL Fragment, Illustrating IN OUT Parameters

```
<operation name="holdVarchar"
   parameterOrder="Integer_1 String_2 String_3">
  <input message="tns:Echo_holdVarchar"/>
  <output message="tns:Echo_holdVarcharResponse"/>
```

```
</operation>
<message name="Echo_holdVarchar">
<part name="Integer_1" type="xsd:int"/>
<part name="String_2" type="xsd:string"/>
<part name="String_3" type="xsd:string"/>
</message>
<message name="Echo_holdVarcharResponse">
<part name="result" type="ns1:double"/>
<part name="String_2" type="xsd:string"/>
<part name="String_3" type="xsd:string"/>
</message>
```

To access the `IN OUT` parameters in JAX-RPC client code, you must use JAX-RPC holders. For example, the code in Example 10–4 retrieves the returned values as `firstName.value` and `lastName.value`, where `firstName` and `lastName` are both String holders. The actual values in the holders are accessed by the member `value`, as shown in the `println` statement.

Example 10–4 Accessing IN OUT Parameters in Client Code by Using JAX-RPC Holders

```
System.out.println("holdVarchar");
StringHolder firstName = new StringHolder("Tom");
StringHolder lastName = new StringHolder("Gordon");
System.out.println("Holder returned:  empid="
+ ci.holdVarchar(id, firstName, lastName)
+ ", name="
+ firstName.value
+ "."
+ lastName.value);
```

Note that the `plsqlAssemble` command line specified `-style rpc`. The RPC style supports holders. The default document-wrapped style does not support holders.

If the Web service had been created with the default document-wrapped style, then a different `holdVarchar` signature would have been generated. The `OUT` arguments would be captured as attributes on the return value.

The following WSDL segment shows the `holdVarchar` operation in the document-wrapped style. In the return type, `EchoUser_holdVarchar_Out`, the attributes `lastnameOut` and `firstnameOut` record the `OUT` value of the PL/SQL parameters `firstname` and `lastname`.

Example 10–5 WSDL Fragment, Illustrating IN OUT Parameters Handled in Document-Wrapped Style

```
<operation name="holdVarchar" parameterOrder="Integer_1">
            <input message="tns:Echo_holdVarchar"/>
            <output message="tns:Echo_holdVarcharResponse"/>
 </operation>

<message name="Echo_holdVarchar">
  <part name="Integer_1" type="xsd:int"/>
</message>
<message name="Echo_holdVarcharResponse">
  <part name="result" type="tns:Echo_holdVarchar_Out"/>
</message>

<complexType name="Echo_holdVarchar_Out">
     <sequence>
         <element name="return" type="double" nillable="true"/>
```

```
        <element name="lastnameOut" type="string" nillable="true"/>
         <element name="firstnameOut" type="string" nillable="true"/>
     </sequence>
</complexType>
```

How to Access SQL XMLType Types from Client Code

The SQL `XMLType` in Example 10–1 on page 10-12 is mapped into the XML `any` type. The `getXmltype` operation in the WSDL fragment in Example 10–6 illustrates this mapping.

Example 10–6 WSDL Fragment, Illustrating the Mapping of SQL XMLType into text_xml

```
<message name="Echo_getXmltypeResponse">
<part name="result" type="ns2:any"/>
</message>
<operation name="getXmltype" parameterOrder="BigDecimal_1">
<input message="tns:Echo_getXmltype"/>
<output message="tns:Echo_getXmltypeResponse"/>
</operation>
```

WebServicesAssembler generates a proxy that maps XML `any` to the Java type `org.w3c.org.dom.Element`. Therefore, a Java client accesses a SQL `XMLType` instance as an `Element` instance.

How to Assemble a Web Service from a SQL Query or DML Statement

Use the `sqlAssemble` command to generate Web services from a SQL statement. The statement can include SQL queries and DML (Data Manipulation Language) statements.

Unlike PL/SQL Web services generation, SQL statement assembly does not generate PL/SQL wrappers. PL/SQL wrappers are generated only to handle PL/SQL `record` or `INDEX BY` table types. These types cannot be used in a SQL statement.

This section contains the following subsections:

- Prerequisites
- Steps for Assembling a Web Service from a SQL Statement or Query
- Sample SQL Statements
- How SQL Queries are Mapped to Web Service Operations
- How DML Operations are Mapped to Web Service Operations

Prerequisites

Before you begin, provide the following files and information.

- The SQL statements or queries. Example 10–7, "Sample SQL Statements" on page 10-18 illustrates the SQL statements used in the following example.

  Multiple `sqlstatement` arguments can be specified on the command line or Ant task. For information on the format of the `sqlstatement` argument, see "sqlstatement" on page 18-56.
- A name for the Web service application.
- The JNDI location of the JDBC data source. This information is used at runtime.
- The JDBC database connection URL. This information is used at compile time.

- The name and password of the schema which the query or statement is based on. This information is used at compile time.

Steps for Assembling a Web Service from a SQL Statement or Query

The following steps use the `sqlAssemble` command to assemble a Web service for the queries and statements on the SCOTT schema.

1. Provide the SQL statements or query, the name and password for the database that they are based on, and the other information described in the Prerequisites section as input to the WebServicesAssembler `sqlAssemble` command.

   For example, the following command generates the Web service application `query.ear`.

   Command line:

```
java -jar wsa.jar -sqlAssemble
   -appName query
   -dataSource jdbc/OracleManagedDS
   -sqlstatement "getEmpCount=select ename, sal from emp where
                     sal>:{mysal NUMBER}"
   -sqlstatement "getEmpBySal=select ename, sal from emp where
                     sal>:{mysal NUMBER}"
   -sqlstatement "updateEmp=update emp SET sal=sal+500 where
                     ename=:{myname VARCHAR}"
   -dbConnection jdbc:oracle:thin:@stacd15:1521:lsqlj
   -dbUser scott/tiger
```

   Ant task:

```
<oracle:sqlAssemble
        appName="query"
        dataSource="jdbc/OracleManagedDS"
        dbConnection="jdbc:oracle:thin:@dsunrde22:1521:sqlj"
        dbUser="scott/tiger">
       <sqlstatement="getEmpCount=select ename, sal from emp where sal>:{mysal
NUMBER}"/>
       <sqlstatement="getEmpBySal=select ename, sal from emp where sal>:{mysal
NUMBER}"/>
        <sqlstatement="updateEmp=update emp SET sal=sal+500 where
ename=:{myname VARCHAR}"/>
/>
```

   In this command and Ant task:

   - `sqlAssemble`—Generates Web services from SQL statements, including SQL queries and DMLs (Data Manipulation Language). To use this command, you must connect to a database. See "sqlAssemble" on page 18-19.
   - `appName`—Specifies the name of an application. Usually, this name is used as a base value for other arguments like `context` and `uri`. See "appName" on page 18-42.
   - `dataSource`—Specifies the JNDI location of the data source used by the Web services at runtime. See "dataSource" on page 18-54.
   - `dbConnection`—Specifies the JDBC URL for the database. See "dbConnection" on page 18-54.
   - `dbUser`—Specifies the database schema and password in the form of *user/password*. See "dbUser" on page 18-55.

- `sqlStatement`—Specifies the DML statement or SQL query to be published as a Web service. See "sqlstatement" on page 18-56.

2. Deploy the service into a running instance of OC4J and bind the application.

   The data source referenced in the `dataSource` argument must have been set up for this OC4J instance.

   The following command lines provide sample deployment and bind commands.

   ```
   % java -jar <J2EE_HOME>/admin_client.jar deployer:oc4j:localhost:port admin
   welcome -deploy -file dist/query.ear -deploymentName query

   % java -jar <J2EE_HOME>/admin_client.jar deployer:oc4j:localhost:port admin
   welcome -bindWebApp plsql plsql-web default-web-site /query
   ```

   In these sample command lines, <*J2EE_HOME*> is the directory where J2EE is installed.

   For more information on deploying EAR files, see the *Oracle Containers for J2EE Deployment Guide*.

3. (Optional) Check that deployment succeeded. OracleAS Web Services provides a Web Service Test Page for each deployed Web service. See "How to Use the Web Services Test Page" on page 13-6 for information on accessing and using the Web Service Test Page.

4. Generate the client-side code.

   - For the J2SE environment, generate stubs (client proxies) for a J2SE Web service client by running the WebServicesAssembler tool with the `genProxy` command. For more information on generating and assembling client-side code for the J2SE environment, see Chapter 15, "Assembling a J2SE Web Service Client".
   - For the J2EE environment, 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 Chapter 14, "Assembling a J2EE Web Service Client".

   For example, the following command uses `genProxy` to generate code for a J2SE client.

   ```
   % java -jar wsa.jar -genProxy
                       -wsdl http://locahost:8888/query/query?WSDL
                       -output build/src/client
                       -mappingFileName ./mapping.xml
                       -packageName oracle.demo.db.query.stub
                       -unwrapParameters true
   ```

   The command generates the client in the `build/src/client` directory.

   In this command:

   - `genProxy`—Creates a static proxy stub that can be used by a J2SE Web service client. See "genProxy" on page 18-34.
   - `wsdl`—Specifies the absolute file path, relative file path, or URL to a WSDL document. See "wsdl" on page 18-65.
   - `output`—Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See "output" on page 18-48.

- `mappingFileName`—Specifies a file location that points to a JAX-RPC mapping file. See "mappingFileName" on page 18-47.
- `packageName`—Specifies the package name that will be used for generated classes if no package name is declared in the JAX-RPC mapping file. See "packageName" on page 18-48.
- `unwrapParameters`—This argument can be set only for document-literal operations and will be ignored for other message formats. When `unwrapParameters` is set to `true`, which is the default, the return type and response type will be unwrapped. This is usually easier to use, especially if the types are simple. See "unwrapParameters" on page 18-69.

5. Compile and run the client.

   List the appropriate JARs on the classpath before compiling the client. Table A–2, " Classpath Components for a Client Using a Client-Side Proxy" 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 Table A–2. See "Setting the Classpath for a Web Service Proxy" on page A-2 for more information on `wsclient_extended.jar` and the client classpath.

   The following command lines provide sample compile and run commands.

```
% javac -classpath path:
     <ORACLE_HOME>/webservices/lib/wsclient_extended.jar:
     :<ORACLE_HOME>/webservices/lib/jaxrpc-api.jar QueryClient.java

% java -classpath path
     <ORACLE_HOME>/webservices/lib/wsclient_extended.jar:
     <ORACLE_HOME>/webservices/lib/jaxrpc-api.jar:
     <J2EE_HOME>/lib/jax-qname-namespace.jar:
     <J2EE_HOME>/lib/activation.jar:
     <J2EE_HOME>/lib/mail.jar:
     <J2EE_HOME>/lib/http_client.jar:
     <ORACLE_HOME>/webservices/lib/commons-logging.jar:
     <ORACLE_HOME>/lib/xmlparserv2.jar QueryClient
```

In this example, <*J2EE_HOME*> is the directory where J2EE is installed; <*ORACLE_HOME*> is the directory where the OC4J is installed.

Sample SQL Statements

Example 10–7 illustrates the definitions of the SQL statements that are exposed as a Web service.

Example 10–7 Sample SQL Statements

```
getEmpCount=select ename, sal from emp where sal>:{mysal NUMBER}
getEmpBySal=select ename, sal from emp where sal>:{mysal NUMBER}
updateEmp=update emp SET sal=sal+500 where  ename=:{myname VARCHAR}
```

How SQL Queries are Mapped to Web Service Operations

A SQL query, when exposed as a Web service, is mapped to three service operations. For example, the `getEmpBySal` query in Example 10–7 generates these service operations.

- `getEmpBySal`**Beans**—returns an array. The array element is an object type with attributes corresponding to the columns in the row of the query result.
- `getEmpBySal`**XMLRowSet**—returns an XML document with the query result in the `WebRowSet` format.
- `getEmpBySal`**XML**—returns an XML document with the query result in Oracle `XDB` rowset format.

Providing three operations from one query is a convenience. The return values differ only in format. Note the naming convention is to attach `Beans`, `XMLRowSet`, and `XML` to the original query name.

The WSDL fragment in Example 10–8 describes the return types of the three operations in the WSDL file.

Example 10–8 WSDL Fragment, Illustrating Service Operations for a SQL Query

```
<complexType name="getEmpBySalBeansResponse">
<sequence>
<element name="result" type="tns:Query_getEmpBySalRowUser" nillable="true"
     minOccurs="0" maxOccurs="unbounded"/>
</sequence>
</complexType>
<complexType name="Query_getEmpBySalRowUser">
<sequence>
<element name="ename" type="string" nillable="true"/>
<element name="sal" type="decimal" nillable="true"/>
</sequence>
</complexType>

<complexType name="getEmpBySalXMLRowSetResponse">
<sequence>
<element name="result" type="ns1:swaRef" nillable="true"/>
</sequence>
</complexType>

<complexType name="getEmpBySalXMLResponse">
<sequence>
<element name="result" type="xsd:any" nillable="true"/>
</sequence>
</complexType>
```

Two of the methods, `getEmpBySalXMLRowSetResponse` and `getEmpBySalXMLResponse`, have parameters of `swaRef` type. For this type, the assembler generates `javax.xml.soap.AttachmentPart` in the client proxy. Example 10–9 illustrates the client code to access the returned query result.

Example 10–9 Accessing Returned Query Results from a swaRef Type in Client Code

```
import oracle.jdbc.pool.OracleDataSource;
import oracle.jdbc.rowset.OracleWebRowSet;
import javax.xml.soap.AttachmentPart;
import org.w3c.org.Element;
import javax.xml.transform.dom.*;
import java.io.*;
   ...

/* Access the query result as Oracle XDB RowSet */
Element element = eme.getEmpBySalXML(new BigDecimal(500));
DOMSource doms = new javax.xml.transform.doc.DOMSource(element);
```

```
buf = new jav.io.ByteArrayOutputStream();
StreamResult streamr = new StreamResult(buf);
trnas.transform(doms, streamr);
System.out.println(buf, toString());


/* Access the query result as Oracle WebRowSet */
ap = eme.getEmpBySalXMLRowSet(new BigDecimal(500));
source = (Source) ap.getContent();
trans = TransformerFactory.newInstance().newTransformer();
buf = new ByteArrayOutputStream();
streamr = new StreamResult(buf);
trans.transform(source, streamr);
InputStream istream = new ByteArrayInputStream(buf.toString().getBytes());
OracleWebRowSet rowset = new OracleWebRowSet();
System.setProperty("http.proxyHost", "www-proxy.us.oracle.com");
System.setProperty("http.proxyPort", "80");
         System.setProperty("javax.xml.parsers.DocumentBuilderFactory",
"oracle.xml.jaxp.JXDocumentBuilderFactory");
rowset.readXml(new InputStreamReader(istream));
rowset.writeXml(new PrintWriter(System.out));
```

The code in Example 10–9 emits the query result in two formats: Oracle XDB row set (`ROWSET`) and Oracle Web row set (`OracleWebRowSet`). Example 10–10 shows the query result as an Oracle XDB row set. Example 10–11 prints the result in `WebRowSet` format. In practice, you can access the variable rowset `OracleWebRowSet` instance in Example 10–9 using `oracle.jdbc.rowset.OracleWebRowSet` APIs.

See Also:

Oracle Database JDBC Developer's Guide and Reference for more information on these data types.

Example 10–10 Query Results as an Oracle XDB Row Set

```
<ROWSET>
<ROW num="1">
<ENAME>SMITH</ENAME><SAL>800</SAL>
</ROW>
<ROW num="2">
<ENAME>ALLEN</ENAME><SAL>1600</SAL>
</ROW>
<ROW num="3">
<ENAME>WARD</ENAME><SAL>1250</SAL>
</ROW>
</ROWSET>
```

Example 10–11 Query Results as a JDBC Web Row Set

```
<?xml version="1.0" encoding="UTF-8"?>
 <webRowSet xmlns="http://java.sun.com/xml/ns/jdbc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/jdbc
http://java.sun.com/xml/ns/jdbc/webrowset.xsd">
   <properties>
<command></command>
<concurrency>1007</concurrency>
<datasource></datasource>
<escape-processing>true</escape-processing>
<fetch-direction>1002</fetch-direction>
<fetch-size>10</fetch-size>
```

```
<isolation-level>2</isolation-level>
<key-columns>
</key-columns>
<map>
</map>
<max-field-size>0</max-field-size>
<max-rows>0</max-rows>
<query-timeout>0</query-timeout>
<read-only>false</read-only><rowset-type>1005</rowset-type>
<show-deleted>false</show-deleted>
<table-name></table-name>
<url>jdbc:oracle:thin:@stacd15:1521:lsqlj1</url>
<sync-provider>

<sync-provider-name>com.sun.rowset.providers.RIOptimisticProvider</sync-provider-n
ame>
  <sync-provider-vendor>Sun Microsystems Inc.</sync-provider-vendor>
  <sync-provider-version>1.0</sync-provider-version>
  <sync-provider-grade>2</sync-provider-grade>
  <data-source-lock>1</data-source-lock>
</sync-provider>
   </properties>
   <metadata>
<column-count>2</column-count>
<column-definition>
  <column-index>1</column-index>
  <auto-increment>false</auto-increment>
  <case-sensitive>true</case-sensitive>
  <currency>false</currency>
  <nullable>1</nullable>
  <signed>true</signed>
  <searchable>true</searchable>
  <column-display-size>10</column-display-size>
  <column-label>ENAME</column-label>
  <column-name>ENAME</column-name>
  <schema-name></schema-name>
  <column-precision>0</column-precision>
  <column-scale>0</column-scale>
  <table-name></table-name>
  <catalog-name></catalog-name>
  <column-type>12</column-type>
  <column-type-name>VARCHAR2</column-type-name>
              </column-definition>

<column-definition>
  <column-index>2</column-index>
  <auto-increment>false</auto-increment>
  <case-sensitive>true</case-sensitive>
  <currency>false</currency>
  <nullable>1</nullable>
  <signed>true</signed>
  <searchable>true</searchable>
  <column-display-size>10</column-display-size>
  <column-label>SAL</column-label>
  <column-name>SAL</column-name>
  <schema-name></schema-name>
  <column-precision>0</column-precision>
  <column-scale>0</column-scale>
  <table-name></table-name>
  <catalog-name></catalog-name>
```

```
  <column-type>2</column-type>
  <column-type-name>NUMBER</column-type-name>
</column-definition>
   </metadata>
   <data>
<currentRow>
  <columnValue>SMITH</columnValue>
              <columnValue>800</columnValue>
</currentRow>
<currentRow>
  <columnValue>ALLEN</columnValue>
            <columnValue>1600</columnValue>
     </currentRow>
     <currentRow>
       <columnValue>WARD</columnValue>
             <columnValue>1250</columnValue>
     </currentRow>
   </data>
 </webRowSet>
```

How DML Operations are Mapped to Web Service Operations

A DML statement is an `UPDATE`, `DELETE`, or `INSERT` SQL statement. The `sqlAssemble` command can expose DML statements into operations in a Web Service. DML operations are automatically committed when successful and automatically rolled back when they are not.

Example 10–12 illustrates a WSDL fragment that exposes the DML statement in Example 10–7:

```
updateEmp=update emp SET sal=sal+500 where ename=:{myname
VARCHAR}
```

The DML statement is exposed as two operations. In this example, `updateEmp` executes the statement; `updateEmpiS` executes it in batch mode. The batched operation takes an array for each original DML argument. Each element in the array is used for one execution in the batch. The result of a batched operation reflects the total number of rows updated by the batch.

Example 10–12 WSDL Fragment, Illustrating the Response Type of a DML Operation

```
    <message name="SqlStmts_updateEmp">
      <part name="salary" type="xsd:string"/>
  </message>
  <message name="SqlStmts_updateSchemaResponse">
      <part name="result" type="xsd:int"/>
  </message>
  <message name="SqlStmts_updateEmpiS">
      <part name="salary" type="tns:ArrayOfstring"/>
  </message>
  <message name="SqlStmts_updateEmpiSResponse">
      <part name="result" type="xsd:int"/>
  </message>
```

How to Assemble a Web Service from an Oracle Streams AQ

Oracle Streams Advanced Queuing is an asynchronous messaging system provided by Oracle databases. By exposing an Advanced Queue (AQ) as a Web service, the client can send a message to a receiver inside the database, or eventually, to another client of the same Web service.

The WebServicesAssembler tool can generate a Web service from an AQ existing in a database. An AQ can have a single consumer or multiple consumers. A single consumer is often referred to as a queue. A multiple consumer AQ is often referred to as a topic. Each Oracle Streams AQ belongs to a queue table, which defines the payload type of all its AQs, and whether the AQs support only queues or topics. The generated Java code employs the Oracle Streams AQ JMS APIs.

Example 10–14 on page 10-27 and Example 10–15 on page 10-29 illustrate the Web service operations that the WebServicesAssembler exposes for a queue and a topic, respectively. The assembler creates the operations based on Oracle Streams AQ and AQ JMS APIs.

This section has the following subsections:

- Prerequisites
- Steps for Assembling a Web Service from an Oracle AQ
- How to Access an AQ Queue Exposed as a Web Service from Client Code
- How to Access an Oracle AQ Queue with JMS
- Sample AQ Queue and Topic Declaration
- Sample Web Service for a Queue Generated by WebServicesAssembler
- Sample Web Service for a Topic Generated by WebServicesAssembler

See Also:

Oracle Streams Advanced Queuing Java API Reference for information regarding Oracle Streams AQ and AQ JMS APIs.

Prerequisites

Before you begin, provide the following files and information.

- A database connection URL to the database where the AQ resides. As an alternative to JDBC, WebServicesAssembler gives you the flexibility of using a JMS queue instance to access an Oracle AQ. For more information, see "How to Access an Oracle AQ Queue with JMS" on page 10-31.
- The name of the schema where the AQ resides and the user name and password to access it. This is used at compile time.
- The name of the queue or topic that you want to expose as a Web service. You can publish only a single queue or topic to be exposed by a Web service. See "Sample AQ Queue and Topic Declaration" on page 10-26 for a sample queue and topic.
- A name for the Web service application.
- The data source JNDI name. This information is used at runtime.

Steps for Assembling a Web Service from an Oracle AQ

The following steps describe how to use WebServicesAssembler to assemble a Web service from an Oracle AQ queue.

1. Provide the files and other information described in the Prerequisites section as input to WebServicesAssembler `-aqAssemble` command.

   For example, the following command creates a Web service application with the `queue.ear` file generated in the current directory. The WebServicesAssembler tool generates Java files to access the queue at runtime. "Sample AQ Queue and Topic Declaration" on page 10-26 illustrates the AQ `sample_queue` declaration.

Command line:

```
java -jar $ORACLE_HOME/webservices/lib/wsa.jar
  -aqAssemble
  -appName queue
  -dataSource jdbc/OracleManagedDS
  -portName assembleQueuePort
  -sql sample_queue
  -dbConnection jdbc:oracle:thin:@stacd15:1521:lsqlj
  -dbUser scott/tiger
```

Ant task:

```
<aqAssemble
   appName="queue"
   dataSource="jdbc/OracleManagedDS"
   sql="sample_queue"
   portName="assembleQueuePort"
   dbConnection="jdbc:oracle:thin:@stacd15:1521:lsqlj"
   dbUser="scott/tiger"
/>
```

You can publish the sample topic in the declaration, `sample_topic`, in the same way as `sample_queue` (but in a different WebServicesAssembler invocation). The only difference would be the values for `sql` and `appName` arguments in the `aqAssemble` command.

In this command and Ant task:

- `aqAssemble`—Generate Web services from an advanced queue in the database. To use this command, you must connect to a database. See "aqAssemble" on page 18-5.
- `appName`—Specifies the name of an application. Usually, this name is used as a base value for other arguments like `context` and `uri`. See "appName" on page 18-42.
- `dataSource`—Specifies the JNDI location of the data source used by the Web services at runtime. See "dataSource" on page 18-54.
- portName—See "portName" on page 18-48.
- `sql`—Specifies a PL/SQL package name. See "sql" on page 18-55.
- `dbConnection`—Specifies the JDBC URL for the database. See "dbConnection" on page 18-54.
- `dbUser`—Specifies the database schema and password in the form of *user*/*password*. See "dbUser" on page 18-55.

2. Deploy the service into a running instance of OC4J and bind the application.

   This step assumes that the AQ has been set up as a data source in the OC4J instance.

   The following command lines provide sample deployment and bind commands.

```
% java -jar <J2EE_HOME>/admin_client.jar deployer:oc4j:localhost:port admin
welcome -deploy -file dist/queue.ear -deploymentName queue

% java -jar <J2EE_HOME>/admin_client.jar deployer:oc4j:localhost:port admin
welcome -bindWebApp queue queue-web default-web-site /queue
```

   In this example, <*J2EE_HOME*> is the directory where J2EE is installed.

For more information on deploying EAR files, see the *Oracle Containers for J2EE Deployment Guide*.

3. (Optional) Check that deployment succeeded.

   OracleAS Web Services provides a Web Service Test Page for each deployed Web service. See "How to Use the Web Services Test Page" on page 13-6 for information on accessing and using the Web Service Test Page.

4. Generate the client-side code.

   - For the J2SE environment, generate stubs (client proxies) for a J2SE Web service client by running the WebServicesAssembler tool with the `genProxy` command. For more information on generating and assembling client-side code for the J2SE environment, see Chapter 15, "Assembling a J2SE Web Service Client".
   - For the J2EE environment, 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 Chapter 14, "Assembling a J2EE Web Service Client".

   For example, the following command uses `genProxy` to generate code for a J2SE client.

```
% java -jar wsa.jar -genProxy
                    -wsdl http://locahost:8888/queue/queue?WSDL
                    -output build/src/client
                    -mappingFileName ./mapping.xml
                    -packageName oracle.demo.db.queue.stub
                    -unwrapParameters true
```

   In this command:

   - `genProxy`—Creates a static proxy stub that can be used by a J2SE Web service client. See "genProxy" on page 18-34.
   - `wsdl`—Specifies the absolute file path, relative file path, or URL to a WSDL document. See "wsdl" on page 18-65.
   - `output`—Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See"output" on page 18-48.
   - `mappingFileName`—Specifies a file location that points to a JAX-RPC mapping file. See "mappingFileName" on page 18-47.
   - `packageName`—Specifies the package name that will be used for generated classes if no package name is declared in the JAX-RPC mapping file. See "packageName" on page 18-48.
   - `unwrapParameters`—This argument can be set only for document-literal operations and will be ignored for other message formats. When `unwrapParameters` is set to `true`, which is the default, the return type and response type will be unwrapped. This is usually easier to use, especially if the types are simple. See "unwrapParameters" on page 18-69.

5. Compile and run the client.

   List the appropriate JARs on the classpath before compiling the client. Table A–2, " Classpath Components for a Client Using a Client-Side Proxy" 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 Table A–2. See "Setting the Classpath for a Web Service Proxy" on page A-2 for more information on `wsclient_extended.jar` and the client classpath.

The following command lines provide sample compile and run commands.

```
% javac -classpath path
<ORACLE_HOME>/webservices/lib/wsclient_extended.jar:
:<ORACLE_HOME>/webservices/lib/jaxrpc-api.jar QueueClient.java

% java -classpath path
<ORACLE_HOME>/webservices/lib/wsclient_extended.jar:
<ORACLE_HOME>/webservices/lib/jaxrpc-api.jar:
<J2EE_HOME>/lib/jax-qname-namespace.jar:
<J2EE_HOME>/lib/activation.jar:<J2EE_HOME>/lib/mail.jar:
<J2EE_HOME>/lib/http_client.jar:
<ORACLE_HOME>/webservices/lib/commons-logging.jar:
<ORACLE_HOME>/lib/xmlparserv2.jar QueueClient
```

In this example, <*J2EE_HOME*> is the directory where J2EE is installed; <*ORACLE_HOME*> is the directory where OracleAS Web Services is installed.

Sample AQ Queue and Topic Declaration

The PL/SQL script in Example 10–13 defines a queue, `sample_queue`, and a topic, `sample_topic`. The queue payload type is `queue_message`, a SQL object type. The topic payload type is `topic_message`, also a SQL object type.

Example 10–13 Sample Queue and Topic Declaration

```
create type scott.queue_message as object (
   Subject               VARCHAR2(30),
   Text                  VARCHAR2(80));
create type scott.topic_message as object (
   Subject               VARCHAR2(30),
   Text                  VARCHAR2(80));
BEGIN
  dbms_aqadm.create_queue_table (
          Queue_table            => 'scott.queue_queue_table',
          Queue_payload_type     => 'scott.queue_message');
  dbms_aqadm.create_queue(
    queue_name  => 'scott.sample_queue',
    queue_table => 'scott.queue_queue_table' );
  dbms_aqadm.start_queue(queue_name => 'scott.sample_queue');

  dbms_aqadm.create_queue_table (
    Queue_table            => 'scott.topic_queue_table',
    Multiple_consumers => TRUE,
    Queue_payload_type     => 'scott.topic_message');
  dbms_aqadm.create_queue(
    queue_name  => 'scott.sample_topic',
    queue_table => 'scott.topic_queue_table' );
  dbms_aqadm.start_queue(queue_name => 'scott.sample_topic');
END;
/
```

Sample Web Service for a Queue Generated by WebServicesAssembler

For the queue described in "Sample AQ Queue and Topic Declaration", Example 10–14 lists the Web service operations exposed by WebServicesAssembler.

In this example, the `send` operation enqueues a payload to the queue. The payload type is the complex type `tns:QueueMessageUser`, which corresponds to the SQL type `QUEUE_MESSAGE`, as shown by `<send/>` element.

The `receive` operation returns a payload from the queue. The `<receiveResponse/>` element shows that the type of the returned payload is `tns:QueueMessage`. The operation blocks until a message becomes available.

The `receiveNoWait` operation returns a payload from the queue. If no message is available in the queue, the operation returns null without waiting.

The `receive2` operation has two arguments.

- `selector` of type `xsd:string`
- `noWait` of type `xsd:boolean`

The `selector` is a filter condition specified in the AQ convention. It allows the `receive` operation to return only messages that satisfies that condition. For example, the `JMSPriority < 3 AND PRICE < 300` selector exposes only messages with priority 3 or higher, and the attribute `PRICE` is lower than 300. If the parameter `noWait` is `true`, the operation does not block.

Example 10–14 Web Service Operations Exposed for a Queue

```
<operation name="receive">
  <input message="tns:SampleQueue_receive" />
  <output message="tns:SampleQueue_receiveResponse" />
  </operation>
<operation name="receive2">
  <input message="tns:SampleQueue_receive2" />
  <output message="tns:SampleQueue_receive2Response" />
  </operation>
<operation name="receiveNoWait">
  <input message="tns:SampleQueue_receiveNoWait" />
  <output message="tns:SampleQueue_receiveNoWaitResponse" />
  </operation>
<operation name="send">
  <input message="tns:SampleQueue_send" />
  <output message="tns:SampleQueue_sendResponse" />
  </operation>

<complexType name="receive">
  <sequence />
  </complexType>
<complexType name="receiveResponse">
<sequence>
  <element name="result" type="tns:QueueMessageUser" nillable="true" />
  </sequence>
  </complexType>
<complexType name="QueueMessageUser">
<sequence>
  <element name="text" type="string" nillable="true" />
  <element name="subject" type="string" nillable="true" />
  </sequence>
  </complexType>
<complexType name="receive2">
<sequence>
  <element name="String_1" type="string" nillable="true" />
  <element name="boolean_2" type="boolean" />
  </sequence>
  </complexType>
```

```
<complexType name="receive2Response">
<sequence>
  <element name="result" type="tns:QueueMessageUser" nillable="true" />
  </sequence>
  </complexType>
<complexType name="receiveNoWait">
  <sequence />
  </complexType>
<complexType name="receiveNoWaitResponse">
<sequence>
  <element name="result" type="tns:QueueMessageUser" nillable="true" />
  </sequence>
  </complexType>
<complexType name="send">
<sequence>
  <element name="QueueMessageUser_1" type="tns:QueueMessageUser" nillable="true"
/>
  </sequence>
  </complexType>
<complexType name="sendResponse">
  <sequence />
  </complexType>
```

Sample Web Service for a Topic Generated by WebServicesAssembler

For the topic described in "Sample AQ Queue and Topic Declaration" on page 10-26, Example 10–15 lists the Web service operations exposed by WebServicesAssembler.

In this example, the `publish` operation enters a payload to the topic. The argument is a payload type, for instance, `tns:TopicMessageUser`, as shown in Example 10–15. The message will be received by all topic subscribers.

The `publish2` operation sends the payload to all the subscribers in the recipients list. This operation takes the following arguments.

- `payload` of type `tns:TopicMessageUser`
- `recipients` of `String` array type

The `publish3` operation broadcasts the payload to the topic. This operation takes the following arguments.

- `payload`, the message to be sent
- `deliveryMode`, of type `xsd:int`—can be either `javax.jms.DeliveryMode.PERSISTENT` or `javax.jms.DeliveryMode.NON_PERSISTENT`. However, only `DeliveryMode.PERSISTENT` is supported in this release. The interface `javax.jms.DeliveryMode` is from the JMS APIs
- `priority`, of type `xsd:int`—specifies the priority of the message. Values can be from 0 to 9, with 0 as lowest priority and 9 as highest.
- `timeToLive`, of type `xsd:long`—indicates the life span of the message in milliseconds. Zero means no limit.

The `receive` operation returns a message sent to the receiver. This operation takes one argument: `receiver`.

The `receiveNoWait` operation returns a message sent to the specified recipient without waiting.

The `receive2` operation returns a filtered message sent to the specified recipient. This operation takes the following arguments.

- `receiver`, of type `xsd:string`—recipient of the filtered message.
- `selector`, of type `xsd:string`—a filter condition specified in the AQ convention.

The `receive3` operation returns filtered payload for the specified recipient. This operation takes the following arguments.

- `receiver`, of type `xsd:string`—recipient of the filtered message.
- `selector`, of type `xsd:string`—a filter condition specified in the AQ convention.
- `timeout`, of type `xsd:long`—specifies the timeout for the operation in milliseconds. Zero means no timeout.

The `subscribe` operation subscribes a user to the topic. The underlying connection supporting the Web service must have appropriate privileges to subscribe a consumer. Otherwise, this operation has no effect.

The `unsubscribe` operation unsubscribes a user from the topic. Again, the underlying connection supporting the Web service must have appropriate privileges to unsubscribe a consumer. Otherwise, this operation has no effect.

See Also:

Oracle Streams Advanced Queuing Java API Reference for information on the privileges needed for subscribing and unsubscribing consumers.

Example 10–15 Web Service Operations Exposed for a Topic

```
<operation name="publish">
  <input message="tns:SampleTopic_publish" />
  <output message="tns:SampleTopic_publishResponse" />
  </operation>
<operation name="publish2">
  <input message="tns:SampleTopic_publish2" />
  <output message="tns:SampleTopic_publish2Response" />
  </operation>
<operation name="publish3">
  <input message="tns:SampleTopic_publish3" />
  <output message="tns:SampleTopic_publish3Response" />
  </operation>
<operation name="receive">
  <input message="tns:SampleTopic_receive" />
  <output message="tns:SampleTopic_receiveResponse" />
  </operation>
<operation name="receive2">
  <input message="tns:SampleTopic_receive2" />
  <output message="tns:SampleTopic_receive2Response" />
  </operation>
<operation name="receive3">
  <input message="tns:SampleTopic_receive3" />
  <output message="tns:SampleTopic_receive3Response" />
  </operation>
<operation name="receiveNoWait">
  <input message="tns:SampleTopic_receiveNoWait" />
  <output message="tns:SampleTopic_receiveNoWaitResponse" />
  </operation>
<complexType name="publish">
```

```
<sequence>
  <element name="TopicMessageUser_1" type="tns:TopicMessageUser" nillable="true"
/>
  </sequence>
  </complexType>
<complexType name="TopicMessageUser">
<sequence>
  <element name="text" type="string" nillable="true" />
  <element name="subject" type="string" nillable="true" />
  </sequence>
  </complexType>
<complexType name="publishResponse">
  <sequence />
  </complexType>
<complexType name="publish2">
<sequence>
  <element name="TopicMessageUser_1" type="tns:TopicMessageUser" nillable="true"
/>
  <element name="arrayOfString_2" type="string" nillable="true" minOccurs="0"
maxOccurs="unbounded" />
  </sequence>
  </complexType>
<complexType name="publish2Response">
  <sequence />
  </complexType>
<complexType name="publish3">
<sequence>
  <element name="TopicMessageUser_1" type="tns:TopicMessageUser" nillable="true"
/>
  <element name="int_2" type="int" />
  <element name="int_3" type="int" />
  <element name="long_4" type="long" />
  </sequence>
  </complexType>
<complexType name="publish3Response">
  <sequence />
  </complexType>
<complexType name="receive">
<sequence>
  <element name="String_1" type="string" nillable="true" />
  </sequence>
  </complexType>
<complexType name="receiveResponse">
<sequence>
  <element name="result" type="tns:TopicMessageUser" nillable="true" />
  </sequence>
  </complexType>
<complexType name="receive2">
<sequence>
  <element name="String_1" type="string" nillable="true" />
  <element name="String_2" type="string" nillable="true" />
  </sequence>
  </complexType>
<complexType name="receive2Response">
<sequence>
  <element name="result" type="tns:TopicMessageUser" nillable="true" />
  </sequence>
  </complexType>
<complexType name="receive3">
<sequence>
```

```
  <element name="String_1" type="string" nillable="true" />
  <element name="String_2" type="string" nillable="true" />
  <element name="long_3" type="long" />
  </sequence>
  </complexType>
<complexType name="receive3Response">
<sequence>
  <element name="result" type="tns:TopicMessageUser" nillable="true" />
  </sequence>
  </complexType>
<complexType name="receiveNoWait">
<sequence>
  <element name="String_1" type="string" nillable="true" />
  </sequence>
  </complexType>
<complexType name="receiveNoWaitResponse">
<sequence>
  <element name="result" type="tns:TopicMessageUser" nillable="true" />
  </sequence>
  </complexType>
```

How to Access an AQ Queue Exposed as a Web Service from Client Code

Example 10–16 illustrates a sample JAX-RPC Web service client that accesses the published Web service. "Sample Web Service for a Topic Generated by WebServicesAssembler" on page 10-28 illustrates the operations exposed by the queues and topics.

Example 10–16 Client Code to Access an AQ Queue Exposed as a Web Service

```
SampleQueuePortClient queue = new SampleQueuePortClient();
     QueueMessageUser m;
     queue.send(new QueueMessageUser( "star chopper", "sample 1"));
     queue.send(new QueueMessageUser("easy blocks", "sample 2"));
     queue.send(new QueueMessageUser("back to future", "sample 3"));
     m = queue.receive();
     while (m != null) {
         System.out.println("Message received from SampleQueue: " + m.getSubject()
+ ": " + m.getText());
         m = queue.receiveNoWait();
          }
```

This client returns the following responses.

```
Message received from SampleQueue: sample 1: star chopper
Message received from SampleQueue: sample 2: easy blocks
Message received from SampleQueue: sample 3: back to future
```

How to Access an Oracle AQ Queue with JMS

By default, the Web service interface code uses the data source to get a JDBC connection. It then uses the connection to access the queue inside the database.

As an alternative to JDBC, you can use JMS to access the queue. The WebServicesAssembler tool provides these specialized arguments to the `aqAssemble` command that let you access the exposed Oracle AQ with a JMS queue instance.

- aqConnectionLocation—the JDNI location of the Oracle Streams AQ JMS queue connection connecting to the exposed AQ.

- aqConnectionFactoryLocation—the JNDI location of the Oracle Streams AQ JMS queue connection factory for the exposed AQ.

Instead of specifying the `dataSource` argument in the `aqAssemble` command, you can specify either of the preceding parameters. The Web service will use a JMS queue at runtime instead of a JDBC-based queue.

How to Assemble a Server-Side Java Class as a Web Service

Use the `dbJavaAssemble` command to generate Web services that invoke a Java class inside the Java VM in an Oracle database. You can expose either static or instance methods as Web service operations. An instance method can be invoked through either a default or singleton instance in the session.

The Java class that you want to expose can contain any of the following parameters and return types.

- primitive types (except `char`)
- serializable types (that is, classes that implement `java.io.Serializable`)
- Java Beans whose attributes are supported types
- JDBC types; that is, `oracle.sql.*` types
- arrays of supported types

This section contains the following subsections:

- Prerequisites
- Steps for Assembling a Web Service from a Server-Side Java Class
- Sample Server-Side Java Class
- Sample Web Service Operations Generated from a Server-Side Java Class

Prerequisites

Before you begin, supply the following information.

- The fully-qualified class name of the server-side Java class
- A database connection URL; used at code generation time
- The name and password of the schema which contains the Java class; used at code generation time
- A name for the Web service application
- The data source JNDI name; used at runtime

Steps for Assembling a Web Service from a Server-Side Java Class

The following steps describe how to use WebServicesAssembler to assemble a Web service from a server-side Java class.

1. Supply the information described in the Prerequisites section as input to WebServicesAssembler `dbJavaAssemble` command.

   For example, in the following `dbAssemble` command, the server-side class, `oracle.sqlj.checker.JdbcVersion`, is part of the SQLJ server-side translator. This command assembles a Web service application for the class, `javacallin.ear`. It also generates a PL/SQL wrapper and a Java stored procedure wrapper. The purpose of the Java stored procedure wrapper is to convert signature types in the server-side Java class into types can be exposed to

PL/SQL stored procedures. Since the `sysUser` argument is declared, the WebServicesAssembler automatically loads the generated wrappers into the database.

Command line:

```
java -jar wsa.jar
  -dbJavaAssemble
  -appName javacallin
  -dbJavaClassName oracle.sqlj.checker.JdbcVersion
  -dbConnection jdbc:oracle:thin:@stacd15:1521:lsqlj
  -dataSource jdbc/OracleManagedDS
  -dbUser scott/tiger
  -sysUser sys/knl_test7
```

Ant task:

```
<oracle:dbJavaAssemble
   appName="javacallin"
   dbJavaClassName="oracle.sqlj.checker.JdbcVersion"
   dbConnection="jdbc:oracle:thin:@stacd15:1521:lsqlj"
   dataSource="jdbc/OracleManagedDS"
   dbUser="scott/tiger"
   sysUser="sys/knl_test7"
/>
```

In this command and Ant task:

- `dbJavaAssemble`—Generates Web services from a Java class inside the Java VM in an Oracle database. To use this command, you must connect to a database. See "dbJavaAssemble" on page 18-12.
- `appName`—Specifies the name of an application. Usually, this name is used as a base value for other arguments like `context` and `uri`. See "appName" on page 18-42.
- `dbJavaClassName`—Specifies the name of the server-side Java class to be published as a Web service. See "dbJavaClassName" on page 18-55.
- `dbConnection`—Specifies the JDBC URL for the database. See "dbConnection" on page 18-54.
- `dataSource`—Specifies the JNDI location of the data source used by the Web services at runtime. See "dataSource" on page 18-54.
- `dbUser`—Specifies the database schema and password in the form of *user*/*password*. See "dbUser" on page 18-55.
- `sysUser`—Specifies the name and password of a user with `SYS` privileges in the form of `dbSysUser/syspassword`. Using this argument allows PL/SQL and Java wrapper code to be installed automatically into the database at code-generation time.See "sysUser" on page 18-57.

At run time, the Web service code uses JDBC to invoke the PL/SQL wrapper, which in turns calls the Java stored procedure wrapper, which eventually calls the server-side class. Example 10–18 on page 10-35 illustrates some Web service operations generated by this command.

2. Deploy the service into a running instance of OC4J and bind the application.

   This step assumes that the data source specified in Step 1 has been installed in this instance of OC4J.

   The following command lines provide sample deployment and bind commands.

```
% java -jar <J2EE_HOME>/admin_client.jar deployer:oc4j:localhost:port admin
welcome  -deploy -file dist/javacallin.ear -deploymentName javacallin

% java -jar <J2EE_HOME>/admin_client.jar deployer:oc4j:localhost:port admin
welcome  -bindWebApp javacallin javacallin-web default-web-site /javacallin
```

In this example, <*J2EE_HOME*> is the directory where J2EE is installed.

For more information on deploying EAR files, see the *Oracle Containers for J2EE Deployment Guide*.

3. (Optional) Check that deployment succeeded.

   OracleAS Web Services provides a Web Service Test Page for each deployed Web service. See "How to Use the Web Services Test Page" on page 13-6 for information on accessing and using the Web Service Test Page.

4. Generate the client-side code.

   - For the J2SE environment, generate stubs (client proxies) for a J2SE Web service client by running the WebServicesAssembler tool with the `genProxy` command. See Chapter 15, "Assembling a J2SE Web Service Client" for more information on generating and assembling client-side code for the J2SE environment.
   - For the J2EE environment, 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. See Chapter 14, "Assembling a J2EE Web Service Client" for more information on generating and assembling client-side code.

   For example, the following command uses `genProxy` to generate code for a J2SE client.

   ```
   % java -jar wsa.jar -genProxy
                       -wsdl http://locahost:8888/javacallin/javacallin?WSDL
                       -output build/src/client
                       -mappingFileName ./mapping.xml
                       -packageName oracle.demo.db.queue.stub
                       -unwrapParameters true
   ```

   In this command:

   - `genProxy`—Creates a static proxy stub that can be used by a J2SE Web service client. See "genProxy" on page 18-34.
   - `wsdl`—Specifies the absolute file path, relative file path, or URL to a WSDL document. See "wsdl" on page 18-65.
   - `output`—Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See "output" on page 18-48.
   - `mappingFileName`—Specifies a file location that points to a JAX-RPC mapping file. See "mappingFileName" on page 18-47.
   - `packageName`—Specifies the package name that will be used for generated classes if no package name is declared in the JAX-RPC mapping file. See "packageName" on page 18-48.
   - `unwrapParameters`—This argument can be set only for document-literal operations and will be ignored for other message formats. When `unwrapParameters` is set to `true`, which is the default, the return type and

response type will be unwrapped. This is usually easier to use, especially if the types are simple. See "unwrapParameters" on page 18-69.

5. Compile and run the client.

   List the appropriate JARs on the classpath before compiling the client. Table A–2, " Classpath Components for a Client Using a Client-Side Proxy" 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 Table A–2. See "Setting the Classpath for a Web Service Proxy" on page A-2 for more information on `wsclient_extended.jar` and the client classpath.

   The following command lines provide sample compile and run commands.

```
% javac -classpath path
<ORACLE_HOME>/webservices/lib/wsclient_extended.jar:
:<ORACLE_HOME>/webservices/lib/jaxrpc-api.jar JavacallinClient.java

% java -classpath path
<ORACLE_HOME>/webservices/lib/wsclient_extended.jar:
<ORACLE_HOME>/webservices/lib/jaxrpc-api.jar:
<J2EE_HOME>/lib/jax-qname-namespace.jar:
<J2EE_HOME>/lib/activation.jar:<J2EE_HOME>/lib/mail.jar:
<J2EE_HOME>/lib/http_client.jar:
<ORACLE_HOME>/webservices/lib/commons-logging.jar:
<ORACLE_HOME>/lib/xmlparserv2.jar JavacallinClient
```

   In this example, <*J2EE_HOME*> is the directory where J2EE is installed; <*ORACLE_HOME*> is the directory where the OC4J is installed.

Sample Server-Side Java Class

Example 10–17 illustrates two APIs in `oracle.sqlj.checker.JdbcVersion`, a server-side Java class distributed with Oracle9*i* and 10*g* databases. The `dbJavaAssemble` command exposes these APIs as a Web service.

Example 10–17 Sample Server-Side Java Class

```
public class oracle.sqlj.checker.JdbcVersion extends java.lang.Object {
    public static int getDriverMajorVersion();
    public static int getDriverMinorVersion();
    ...
}
```

Sample Web Service Operations Generated from a Server-Side Java Class

The WSDL fragment in Example 10–18 illustrates the Web service operations generated for the `JdbcVersion` APIs `getDriverMajorVersion` and `getDriverMinorVersion` in Example 10–17.

Example 10–18 WSDL Fragment, Illustrating Operations Generated for a Server-Side Java Class

```
<complexType name="getDriverMajorVersion">
  <sequence />
  </complexType>
<complexType name="getDriverMajorVersionResponse">
<sequence>
  <element name="result" type="decimal" nillable="true" />
```

```
  </sequence>
  </complexType>
<complexType name="getDriverMinorVersion">
  <sequence />
  </complexType>
<complexType name="getDriverMinorVersionResponse">
<sequence>
  <element name="result" type="decimal" nillable="true" />
  </sequence>
  </complexType>

<portType name="JdbcVersion">
<operation name="getDriverMajorVersion">
  <input message="tns:JdbcVersion_getDriverMajorVersion" />
  <output message="tns:JdbcVersion_getDriverMajorVersionResponse" />
  </operation>
<operation name="getDriverMinorVersion">
  <input message="tns:JdbcVersion_getDriverMinorVersion" />
  <output message="tns:JdbcVersion_getDriverMinorVersionResponse" />
  </operation>
  </portType>
```

Note: A server-side Java class can also be invoked through JDBC (rather than through Web services). If this is the case, refer to the *Oracle Database JPublisher User's Guide* to find out how to generate a proxy class for invoking database server-side Java.

Developing a Web Service Client in the Database

This section contains the following subsections:

- Understanding Web Service Call-Out
- How to Call Web Services from the Database

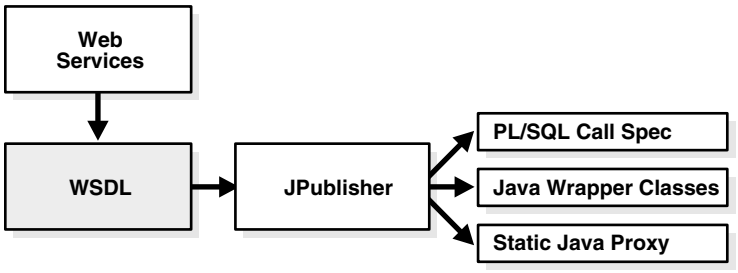
Understanding Web Service Call-Out

Oracle JPublisher supports PL/SQL and Java Web service call-outs by creating the appropriate stub code. This enables you to use PL/SQL or Java to invoke a Web service client from inside the database. For a Web service call-out, you supply a WSDL file or location to Oracle JPublisher. Oracle JPublisher generates a PL/SQL wrapper and the necessary database server-side Java classes that implement the PL/SQL wrapper. The generated PL/SQL wrapper contains a PL/SQL procedure or function for each Web service operation.

As an alternative, Oracle JPublisher has the ability to generate Java client proxies only. These Java client proxies can be used for Web service call-outs by Java code in the database.

Figure 10–3 illustrates the stub code that Oracle JPublisher can generate.

Figure 10–3 Creating Web Service Call Out Stubs



Note: If you must dynamically construct invocations of external Web services based on a WSDL which is available only at runtime, use the JAX-RPC Dynamic Invocation Interface API for Java or the PL/SQL `UTL_DBWS` package.

The client proxy which Oracle JPublisher generates is based on the simplified client code generated for Java proxies in OracleAS Web Services 10.1.3. Therefore, the Java and PL/SQL client which Oracle JPublisher generates is fully supported by OracleAS Web Services 10.1.3.1. In addition, Oracle JPublisher can also generate OracleAS Web Services 9.0.4-style Web service clients.

Web service call-out requires these utilities and tools.

- Database Release 9.2 or later
- Database Web Service call-out Utilities

  Load the JAR and SQL files into the database as instructed. These utilities are available from the Oracle Database Web Services Web site.

  `http://www.oracle.com/technology/tech/webservices/database.html`

- Oracle JPublisher 10*g*

  If you do not have Oracle JPublisher installed, you can obtain it from the JDBC, SQLJ, and Oracle JPublisher download Web site.

  `http://www.oracle.com/technology/tech/java/java_db/index.html`

These are the required Oracle JPublisher options for Web service call-outs.

- `proxywsdl`—the URL of the WSDL file for the Web service to be invoked
- `user`—the database schema (and password) for which the PL/SQL wrapper is generated

These are the optional Oracle JPublisher parameters.

- `httpproxy`—the HTTP proxy host and port for accessing the WSDL file
- `sysuser`—the database user (and password) with SYSDBA privileges
- `proxyopts`—a list of options specific to `proxywsdl`
- `dir`—the directory storing all the generated files

The `sysUser` argument allows Oracle JPublisher to load the generated file into the database. If this argument is not declared, you must manually load the generated file into the database to invoke the Web service from PL/SQL.

See Also:

Oracle Database JPublisher User's Guide for examples and options related to Web service call-out, such as `proxywsdl`, `proxyopts`, and `httpproxy`.

How to Call Web Services from the Database

The Web services call-out utility allows a client to access Web services from the database. Web services call-out can be employed by PL/SQL clients, SQL statements and Java in the database. This utility is based on Oracle JPublisher 10*g* Release 2 (10.2) and OracleAS Web Services 10g Release 3 (10.1.3.1).

Two versions of the utility are available, based on whether your target is the Oracle Database 10*g* (Release 10.1 or 10.2), or a pre-10g Oracle Database.

The call-out utility is available at the following Web site under the heading "*Database as Web Services consumer: Calling-out external Web services*".

`http://www.oracle.com/technology/sample_code/tech/java/jsp/dbwebservices.html`

The following sections describe how a client in the database can perform Web services call-out:

- How to Perform Web Services Call-Out using Static Proxy and JPublisher
- How to Perform Web Services Call-Out using DII and the SYS.UTL_DBWS Utility

How to Perform Web Services Call-Out using Static Proxy and JPublisher

The Oracle JPublisher command line option `-proxywsdl` can be used to generate database-side Java and PL/SQL wrappers from the WSDL file of a Web service. To allow JPublisher to generate and load wrappers for Web service clients into the database, the `dbwsa.jar` and `dbwsclient.jar` files must be present in the classpath and inside the database respectively.

The following procedure sets up the environment and the database for Oracle JPublisher-supported Web service call-out. This procedure needs to be performed only once.

1. Download and install the Oracle JPublisher 10*g* Release 2 (10.2) if it is not already on your system.

   You can obtain the Oracle JPublisher release from the following Web site:

   `http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html`

2. Add the `dbwsa.jar` to the directory `ORACLE_HOME\sqlj\lib` (Windows) or `ORACLE_HOME/sqlj/lib` (Solaris).

3. Set up the appropriate JDK as the Java VM and Java compiler.

   The version of the JDK must be the same as the Java VM in the target database:

   - use the JDK 1.4 for the Oracle Database 10*g* (Release 10.1 or 10.2)
   - use the JDK 1.3 for the Oracle Database 9.2

4. Add `dbwsa.jar` file to the classpath environment variable.

5. Load the `dbwsclient.jar` file either into the `SYS` schema or into the schema where the Web service client will be invoked.

   For example, the following `loadjava` command will load the `dbwsclient.jar` file into the `SYS` schema.

   ```
   %loadjava -u sys/change_on_install -r -v -f -s -grant public -noverify
   -genmissing dbwsclient.jar
   ```

   The following `loadjava` command illustrates how to load the `dbwsclient.jar` file into a specific schema.

   ```
   % loadjava -u scott/tiger -r -v -f -noverify -genmissing dbwsclient.jar
   ```

Example

The following example illustrates how to generate Java and PL/SQL wrappers for a Web service client and then invoke it by using SQL statements. The example follows these general steps:

1. Identify the Web service you want to invoke.
2. Call Oracle JPublisher with the appropriate options to generate the client proxy, the PL/SQL and Java wrappers, and load them into the database.

   An Oracle JPublisher command to do this would include the required `-proxywsdl` and `-user` options. The command could also include the optional `-endpoint`, `-httpproxy`, `-sysuser`, `-dir`, and `-proxyopts` options. For example:

   ```
   % jpub -user=username/password -sysuser=superuser_name/superuser_password
   -proxywsdl=WSDL_URL -endpoint=Web_services_endpoint
   ```

   It is assumed that the Web service has been previously deployed at `http://localhost:8888/javacallout/javacallout`

   The following command creates the Web service client and its Java and PL/SQL wrappers in the subdirectory `tmp`, then loads the wrappers into the database.

   ```
   % jpub -user scott/tiger -sysuser sys/change_on_install
   -proxywsdl=sample/javacallout.wsdl
   -endpoint=http://localhost:8888/javacallout/javacallout -dir=tmp
   ```

   This command produces the following output:

   ```
   tmp/HelloServiceEJBJPub.java
   tmp/plsql_wrapper.sql
   tmp/plsql_dropper.sql
   tmp/plsql_grant.sql
   tmp/plsql_revoke.sql
   Executing tmp/plsql_dropper.sql
   Executing tmp/plsql_wrapper.sql
   Executing tmp/plsql_grant.sql
   Loading tmp/plsql_proxy.jar
   ```

3. Invoke the Web service from inside the database.

   You can invoke the PL/SQL functions provided in `tmp/plsql_wrapper.sql`. Each PL/SQL function corresponds to an operation in the Web service. For example, if your Web service is available at the following endpoint:

   ```
   http://localhost:8888/javacallout/javacallout
   ```

   Then you can issue the following SQL command.

```
SQL> select jpub_plsql_wrapper.sayhello('hello') from dual;
```

The command will return the following output.

```
JPUB_PLSQL_WRAPPER.SAYHELLO('HELLO')
-----------------------------------
HELLO!! You just said :hello
```

For more information about Oracle JPublisher call-out support, see "Web service Call-out using Dynamic Invocation Interface and the SYS.UTL_DBWS Utility" in *Oracle Database JPublisher User's Guide*. This is available from the following Web site.

http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html

How to Perform Web Services Call-Out using DII and the SYS.UTL_DBWS Utility

A PL/SQL user can invoke Web services using the PL/SQL package `SYS.UTL_DBWS`. This package provides wrapper functions for the JAX-RPC Dynamic Invocation Interface (DII) APIs.

The following procedure applies to Oracle Database 10*g* Release 1 or later. The procedure sets up the database for Web service call-out using `SYS.UTL_DBWS`, and refreshes the utility with more recent features provided in by the download.

1. Load the `dbwsclient.jar` file into the database, as described in "How to Perform Web Services Call-Out using Static Proxy and JPublisher".
2. Execute `utl_dbws_decl.sql` and `utl_dbws_body.sql` SQL scripts as `SYS`.

   You can now use `SYS.UTL_DBWS` to call Web services.

Example

The call-out utilities download includes the SQL scripts `samples/test-plsql-dii.sql` and `samples/test-plsql-dii2.sql`. Both scripts invoke the `sayHello` operation defined in `javacallout.wsdl`. Examine the two scripts for their correspondences with the WSDL file. Both scripts produce the following output; the phrase `PL/SQL DII client return` is produced by the client code.

```
PL/SQL DII client return HELLO!! You just said :hello
```

For more information on the `SYS.UTL_DBWS` utility, see *Oracle Database PL/SQL Packages and Types Reference* available from the following Web site.

http://www.oracle.com/technology/documentation/database10g.html

Tool Support for Web Services that Expose Database Resources

With Oracle JDeveloper, you can create a Web service based on program units in a PL/SQL package that is stored in an Oracle database. You can use the wizards in Oracle JDeveloper to perform the following tasks.

- Create the PL/SQL Package in the Database
- Create the PL/SQL Web service
- Deploy the PL/SQL Web service
- Create a stub to use the Web service

For more information on using Oracle JDeveloper to create PL/SQL package units and expose them as a Web service, see the Oracle JDeveloper on-line help.

Limitations

See "Developing Web Services From Database Resources" on page F-7.

Additional Information

For more information on:

- using the Test Page to test Web service deployment, see Chapter 13, "Testing Web Services Deployment".
- building J2SE clients, see Chapter 15, "Assembling a J2SE Web Service Client".
- building J2EE clients, see Chapter 14, "Assembling a J2EE Web Service Client".
- JAX-RPC handlers, see Chapter 16, "Using JAX-RPC Handlers".
- using the WebServicesAssembler tool to assemble Web services, see Chapter 18, "Using WebServicesAssembler".
- packaging and deploying Web services, see Chapter 19, "Packaging and Deploying Web Services"
- JAR files that are needed to assemble a client, see Appendix A, "Web Service Client APIs and JARs".
- Web services interoperability, see "Ensuring Interoperable Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- using quality of service features in Web service clients, see "Managing Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- adding security to a Web service, see the *Oracle Application Server Web Services Security Guide*.
- adding reliability to a Web service, see "Ensuring Web Service Reliability" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- adding an auditing and logging configuration to a Web service, see "Auditing and Logging Messages" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- processing nonstandard data types, see "Custom Serialization of Java Value Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- the JAX-RPC mapping file, see "JAX-RPC Mapping File Descriptor" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- data types supported by OracleAS Web Services, see "Mapping Java Types to XML and WSDL Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- Oracle JDeveloper tool support for Web service development, see the Oracle JDeveloper on-line help.

11

Assembling Web Services with Annotations

This chapter describes how to use annotations to quickly develop Web services from Java classes. The Web service can use either HTTP or JMS as the transport mechanism. You can use the annotations feature only when you are performing bottom up development of a Web service from Java classes.

This chapter contains the following sections:

- OracleAS Web Services and J2SE 5.0 Web Service Annotations
- Supported Annotations

OracleAS Web Services and J2SE 5.0 Web Service Annotations

The standard Java mechanisms for the design, development, and deployment of Web services require you to supply a substantial amount of information. For example, to deploy a service based on a Java class you must provide the class itself, an associated service endpoint interface, and potentially, additional metadata files.

> **See Also:**
>
> Chapter 7, "Assembling a Web Service with Java Classes" for more information on generating a Web service with Java classes.

Oracle Application Server Web Services provides support for J2SE 5.0 Web Service annotations (also known as the Web Services Metadata for the Java Platform (JSR-181) specification). The specification, available from the following Web site, defines an annotated Java syntax for programming Web services.

`http://www.jcp.org/en/jsr/detail?id=181`

The following lists describe the advantages and disadvantages of using the J2SE 5.0 Web Services annotations.

J2SE 5.0 Web Services Annotations Advantages

- Based on a community standard.
- Supported by the J2SE 5.0 JDK. The metadata is stored directly in the class file, making them portable.
- Language-level IDE support. IDEs that support the J2SE 5.0 JDK will recognize J2SE 5.0 Web Service annotations and be able to provide things like auto-completion and syntax checking.

J2SE 5.0 Web Services Annotations Disadvantages

- Requires you to use the J2SE 5.0 JDK.

- See the limitations described in "Assembling Web Services with Annotations" on page F-8.

Using the annotated Java syntax requires the J2SE 5.0 JDK. The J2SE 5.0 JDK also provides documentation on the language-level support for annotations, such as how to define them, their syntax, and how to create new annotations. The J2SE 5.0 JDK is available from the following Web site.

`http://java.sun.com/j2se/1.5.0/download.jsp`

Java Metadata Annotations is a J2SE 5.0 JDK feature that enables you to add special tags to your Java classes. These tags are then compiled into the class byte code and made accessible to third party annotation processors. The Web Services Metadata for the Java Platform specification further extends this feature with metadata tags for Web services. OracleAS Web Services supports this specification and allows developers to customize the Web services generation process through the use of annotations.

You must have the JVM from the J2SE 5.0 JDK to use Web service annotations in OracleAS Web Services. The WebServicesAssembler tool can generate Web services from Java classes that have J2SE 5.0 annotations. When the tool is used in a J2SE 5.0 JDK JVM, the `assemble` and `genWsdl` commands will process annotations in a given Java class.

The EJB 3.0 specification has extended the Web Services Metadata for the Java Platform specification by adding metadata tags for EJBs. The specification supports the Web Services Metadata for the Java Platform tags, allowing EJBs to be exposed as Web services. EJB 3.0 annotations are processed when an EAR file containing EJBs is deployed to a running Oracle Application Server. Any interfaces with J2SE 5.0 JDK annotations that are implemented by session beans will be processed. Note that you cannot use the WebServicesAssembler tool to process EJB 3.0 annotations.

Supported Annotations

The following sections describe the annotations that are supported by OracleAS Web Services.

- Supported J2SE 5.0 JDK Annotations
- Oracle Additions to J2SE 5.0 JDK Annotations

Supported J2SE 5.0 JDK Annotations

OracleAS Web Services enables you to use the entire set of annotations described in the Web Services Metadata for the Java Platform specification. Table 11–1 provides a brief description of these annotations. The Web Services Metadata for the Java Platform specification provides more information on these annotations.

Also included in Table 11–1 is a brief description of the `@WebServiceRef` annotation. The OracleAS Web Services enables you to use the `@WebServiceRef` annotation to reference a Web service endpoint from within a version 3.0 EJB, a servlet, or a Java class in an application client container. Support for the `@WebServiceRef` annotation also includes support for the injection of the `service-ref` target in the `ejb-jar.xml` deployment descriptor. For more information on this annotation, see the Java API for XML-based Web Services 2.0 (JSR-224) specification:

`http://www.jcp.org/en/jsr/detail?id=224`

Table 11–1 J2SE 5.0 JDK Annotations Supported by OracleAS Web Services

J2SE 5.0 JDK Annotation	Description
javax.jws.HandlerChain	Associates the Web service with an externally defined handler chain.
javax.jws.Oneway	Indicates that the given Web method has only an input message, but no output.
javax.jws.WebMethod	Indicates that the method will be exposed as a Web service operation.
javax.jws.WebParam	Customizes the mapping of an individual parameter to a Web service message part and XML element.
javax.jws.WebResult	Customizes the mapping of the return value to a WSDL part and XML part.
javax.jws.WebService	Marks a Java class as implementing a Web service, or a Java interface as defining a Web Service Interface.
javax.jws.soap.SOAPBinding	Specifies the mapping of the Web service onto the SOAP message protocol.
javax.jws.soap.SOAPMessageHandlers	Specifies a list of SOAP protocol handlers that run before and after business methods on the Web service.
javax.xml.ws.WebServiceRef	Declares a reference to a Web service endpoint from within a version 3.0 EJB, a Java class, or a servlet in the application client container. The properties for the `WebServiceRef` annotation can play a role in how WebServicesAssembler generates the name of the JAX-RPC mapping file. See "How the WebServiceRef Annotation Determines the Mapping File Name" for more information.

Using WebServiceRef within OracleAS Web Services

The Java API for XML-based Web Services 2.0 (JSR-224) specification defines the `@WebServiceRef` annotation for referencing a Web service endpoint from within a version 3.0 EJB, a servlet, or a Java class in an application client container. Table 11–2 describes the properties that can be used with the `@WebServiceRef` annotation.

Table 11–2 WebServiceRef Annotation Properties

Property	Description
mappedName	A proprietary name for JNDI lookups.
name	The JNDI lookup name of the Web service. This value can be found in the `java:comp/env` namespace.
type	The resource type as a Java class object. By default, this value is `javax.xml.rpc.Service`.
value	The service type as a Java class object. By default, this is the class of the injection target.
wsdlLocation	The URL for the WSDL description of this Web service. If this annotation is present, it overrides the WSDL URL specified by the `@WebService` annotation of the referenced generated service interface.

The following sections provide more information about using the `@WebServiceRef` annotation in OracleAS Web Services.

- How the WebServiceRef Annotation Determines the Mapping File Name
- Specifying WSDLs with Multiple Ports in the WebServiceRef Annotation
- WebServiceRef Annotation Examples

How the WebServiceRef Annotation Determines the Mapping File Name For each `@WebServiceRef` that occurs in a Java class or version 3.0 EJB, a JAX-RPC mapping file must be included in the deployed archive. WebServicesAssembler will generate the mapping file, but you must ensure that it appears in the appropriate location for your Web service. If the mapping file does not appear in these locations, then you must copy it manually.

- If the Web service is based on a version 3.0 EJB, then the generated JAX-RPC mapping file must appear directly under the `META-INF` directory.
- If the Web service is based on Java classes, then the generated JAX-RPC mapping file must appear directly under the `WEB-INF` directory.

The following steps describe how WebServicesAssembler determines the name of the mapping file. These steps are followed in order until a name can be successfully assigned. If the last step fails, then the process fails and a name is not assigned.

1. If the injection target class is a service endpoint interface (SEI) and the `@WebService` annotation and the `serviceName` property is specified, then the deployment system will try to load the mapping file with the following name:

   `[@WebService.serviceName]Service-java-wsdl-mapping.xml`

   In this example, `[@WebService.serviceName]` represents the value of the `serviceName` property.

2. If the injection target class is a service endpoint interface with the `@WebServiceRef` annotation and the `serviceName` property is specified, the deployment system will try to load the mapping file with the following name:

   `[@WebServiceRef.name]Service-java-wsdl-mapping.xml`

   In this example, `[@WebServiceRef.name]` represents the value of the `name` property.

3. If the injection target class is a service endpoint interface with the `@WebServiceRef` annotation and the `wsdlLocation` property is specified, then the mapping file will have the same base name as the WSDL file referenced by the `wsdlLocation` property.

   `[base name of WSDL file]Service-java-wsdl-mapping.xml`

   For example, if the WSDL file is named `MyService.wsdl`, then the mapping file for that service must be `MyServiceService-java-wsdl-mapping.xml`. This is consistent with what WebServicesAssembler generates when you issue a `genWSDL` or `genInterface` command. So, copying a mapping file that WebServicesAssembler generates should work naturally.

   Note: The `wsdlLocation` property cannot point to a remote WSDL. The WSDL file must be located in the same archive as the mapping file.

Specifying WSDLs with Multiple Ports in the WebServiceRef Annotation According to the Java API for XML-based Web Services 2.0 specification, if the WSDL specified for the `@WebServiceRef` has multiple ports, then the injection target for the annotation must

be of type `javax.xml.rpc.Service`. When the service is injected to this target, a call to `getPort(QName portName, Class serviceEndpointInterface)` can be made to get the exact port for invocation. In this signature, `portname` is the name of the specific port you want to work with and `serviceEndpointInterface` is the name of the service endpoint interface class file.

WebServiceRef Annotation Examples This section provides examples of using the `@WebServiceRef` annotation.

In Example 11–1, the reference to the service endpoint interface is being injected into a field and into a setter method. In this case the mapping file would be named `SimpleService-java-wsdl-mapping.xml`.

Example 11–1 Injecting the Reference into a Field and a Setter Method

```
@Stateless(name="WSRefBean")
public class StatelessEJBClient implements StatelessEJBClientInterface{


@WebServiceRef(name="Simple",wsdlLocation="META-INF/SimpleEJBServiceService.wsdl")
    private SimpleEJBService svc;

    private SimpleEJBService svc2;

    public String echoString(String str) throws RemoteException {
        if(svc == null){
            return "fail1";
        }
        if(svc2 == null){
            return "fail2";
        }
        String test = svc.echoString("Success");
        test = svc2.echoString(test);
        return str + test;
    }


@WebServiceRef(name="Simple",wsdlLocation="META-INF/SimpleEJBServiceService.wsdl")
    public void setSvc2(SimpleEJBService service){
        svc2 = service;
    }
}
```

Example 11–2 is similar to the previous example, the only difference is that the injection is into a `javax.xml.Service` object instead of an actual service endpoint interface. Again, based on the `wsdlLocation`, the mapping file is expected to be in `META-INF` (because this is an EJB) and its name should be `SimpleService-java-wsdl-mapping.xml`.

Example 11–2 Injecting the Reference into javax.xml.Service Object

```
@Stateless(name="WSRefBean3")
public class StatelessEJBClient3 implements StatelessEJBClientInterface3{


@WebServiceRef(name="Simple",wsdlLocation="META-INF/SimpleEJBServiceService.wsdl")
    private Service svc;

    public String echoString(String test) throws RemoteException {
        if(svc == null){
```

```
            return "fail";
        }
        try{
            SimpleEJBService port =
(SimpleEJBService)svc.getPort(SimpleEJBService.class);
            return port.echoString(test);
        }catch(Exception ex){
            ex.printStackTrace();
            throw new RemoteException(ex.getMessage(),ex);
        }
    }
}
```

Example 11–3 uses the `ejb-jar.xml` deployment descriptor for injection. When using this technique, you must specify the `version="3.0"` on the `ejb-jar.xml`. This deployment descriptor can work in conjunction with `@WebServiceRef`.

When you use the `ejb-jar.xml` deployment descriptor for injection, you can also directly specify the mapping file in the `<jaxrpc-mapping-file>` element. In this case, the mapping file is `CustomSessionBeanService-java-wsdl-mapping.xml`.

Example 11–3 Injecting a Reference by Using the ejb-jar.xml Deployment Descriptor

```
<ejb-jar version="3.0">
    <enterprise-beans>
        <session>
            <ejb-name>WSRefBean2</ejb-name>
            <ejb-class>oracle.j2ee.tests.ejb.impl.StatelessEJBClient2</ejb-class>
            <session-type>Stateless</session-type>
            <service-ref>
                <service-ref-name>CustomSessionEJB</service-ref-name>
                <service-interface>javax.xml.rpc.Service</service-interface>

<service-ref-type>oracle.j2ee.tests.ejb.impl.CustomSession</service-ref-type>
                <wsdl-file>META-INF/CustomSessionBeanService.wsdl</wsdl-file>

<jaxrpc-mapping-file>META-INF/CustomSessionBeanService-java-wsdl-mapping.
xml</jaxrpc-mapping-file>

                <injection-target>

<injection-target-class>oracle.j2ee.tests.ejb.impl.
StatelessEJBClient2</injection-target-class>
                    <injection-target-name>svc</injection-target-name>
                </injection-target>
            </service-ref>
        </session>
    </enterprise-beans>
</ejb-jar>
```

Oracle Additions to J2SE 5.0 JDK Annotations

This section describes the Oracle proprietary annotations that can be read and processed by the Java Metadata Annotations feature in the J2SE 5.0 JDK.

- Deployment Annotation
- Schema Annotation
- WSIFEJBBinding Annotation

Deployment Annotation

The `@Deployment` annotation (`oracle.webservices.annotations.Deployment`) can be used to supply the deployment attributes to an endpoint implementation or to a service endpoint interface.

Table 11–3 describes the `@Deployment` annotation's properties. All of the properties are optional.

> **Note:** The properties in the following table could potentially be overridden by the deployment descriptor or by argument values passed by WebServicesAssembler. For more information, see "How to Override Annotation Values with WebServicesAssembler" on page 11-13.

Table 11–3 Deployment Annotation Properties

Properties	Data Type	Description
contextPath	String	This value specifies the default context for the Web service. If you want to package multiple Web services in a WAR file and each endpoint class has an `@Deployment` annotation, then the value of the `contextPath` property must be the same for all of the services. This is because `contextPath` is global to a Web service deployment. If the Web services have differing `contextPath` values, then their behavior will be unpredictable. If only one `contextPath` property is set, then that will be the root context for all Web services in the same EAR or WAR file. You can override the value of this property by providing a valid value for the `<context-root>` element in the `oracle-webservices.xml` deployment descriptor that is deployed with the EAR file. Setting `<context-root>` in the deployment descriptor overrides all `contextPath` properties in all Web services that have the `Deployment` annotation. **Default:** the name of the EAR file.

Table 11–3 (Cont.) Deployment Annotation Properties

Properties	Data Type	Description
portName	String	This value specifies the `portName` element in the generated WSDL. Each Web service can have a different `portName` value. You can override the value of the `portName` property by entering a value for the `<port-component[name]>` attribute in the `oracle-webservices.xml` deployment descriptor deployed with the EAR file. **Default:** For version 3.0 EJBs, this value is taken from the name of the EJB. This name can be specified by the `@Stateless.name` property on the EJB. If the `@Stateless.name` property is not set, then the default value for `portName` is will be the simple name of the EJB. For all other Web services, the default value for `portName` will be generated with the following pattern: `[`*`protocol`*`][`*`binding_and_version`*`]` For example, `HttpSoap11` or `HttpSoap12`.
restSupport	Boolean	This Boolean value identifies whether the service is a REST Web service. If `true`, the port will supports REST-style `GET` and `POST` requests and responses. **Default:** `false`.
uriPath	String	This value is appended to the value of the `contextPath` property. Each Web service can have a different `uriPath` value. You can override the value of this property by providing a URI in the `<endpoint-address-uri>` element for the corresponding `<port-component[name]>`. These elements must appear in the same service element in the `oracle-webservices.xml` deployment descriptor deployed with the EAR file. Note: if there is a conflict in between the values of `<url-pattern>` in the `web.xml` deployment descriptor and the value of `uriPath`, then the value of `<url-pattern>` will take precedence. **Default:** For version 3.0 EJBs, the default for `uriPath` is the `@Stateless.name` property. For other Web services, the default is the simple (unqualified) class name.

The following example illustrates an interface that uses the `@Deployment` annotation. In this example, the Web service will be accessed by the URL `http://$HOST:$PORT/ejb30/ejb30-simple` after it is deployed.

```
...
@WebService(name="CustomSession",
            targetNamespace="http://ejb.oracle.com/targetNamespace",
            serviceName="CustomSessionBeanService")
@Deployment(contextPath="ejb30",
            uriPath="ejb30-simple",
            portName="Custom")
public interface CustomSessionIntf extends Remote{
...
```

Schema Annotation

The `@Schema` annotation (`oracle.webservices.annotations.Schema`) can be used to configure how schema elements are generated in the WSDL. This is an optional annotation that has only optional properties: `elementFormDefault` and `targetNamespace`.

The `elementFormDefault` property enables you to change the value of the `elementFormDefault` attribute in the WSDL. The value of this attribute indicates whether locally declared elements must be qualified by the target namespace in an instance document. If the value of this attribute is `unqualified`, then locally declared elements should not be qualified by the target namespace. If the value of this attribute is `qualified`, then locally declared elements must be qualified by the target namespace.

The `@Schema.elementFormDefault` property is an enumeration of two possible values: `@Schema.ElementFormDefault.QUALIFIED` and `@Schema.ElementFormDefault.UNQUALIFIED`. By default, OracleAS Web Services tools set the `elementFormDefault` attribute in all schemas in the WSDL to `qualified`. By using the `@Schema.elementFormDefault` property, you can change this behavior.

The `targetNamespace` property can be used to set the `targetNamespace` attribute of the generated schema. This namespace will become the default for all types.

Table 11–4 provides a summary of the `@Schema` annotation's properties.

> **Note:** The properties in the following table could potentially be overridden by the deployment descriptor or by argument values passed by WebServicesAssembler. For more information, see "How to Override Annotation Values with WebServicesAssembler" on page 11-13.

Table 11–4 Schema Annotation Properties

Property	Data Type	Description
elementFormDefault	`enum`, with values `QUALIFIED` and `UNQUALIFIED`	Enables you to change the value of the `elementFormDefault` attribute in the WSDL. This property can have the following values: ▪ `QUALIFIED`—Changes the value of the `elementFormDefault` attribute in the WSDL to `qualified`. Locally declared elements will be qualified by the target namespace. ▪ `UNQUALIFIED`—Changes the value of the `elementFormDefault` attribute in the WSDL to `unqualified`. All locally declared elements will not be qualified by the target namespace. **Default:** `QUALIFIED`
targetNamespace	String	Sets the `targetNamespace` attribute of the generated schema. This namespace will become the default for types. **Default:** An empty string, however, this will result in the same `targetNamespace` of the WSDL becoming the default.

WSIFEJBBinding Annotation

The `@WSIFEJBBinding` annotation (`oracle.webservices.annotations.@WSIFEJBBinding`) generates a Web Services Invocation Framework (WSIF) binding in the WSDL for a version 3.0 EJB. The WSIF bindings are applied only to a single port. The annotation and its associated properties add extensions to the WSDL that allow a WSIF client to access a 3.0 EJB exposed as a Web service.

Note: In OracleAS Web Services you cannot use a WSDL document which defines only WSIF bindings; it must also define a SOAP binding.

When you assemble a Web Service with the `WSIFEJBBinding` annotation, always include the `SoapBinding` annotation, set to the appropriate style and use, to ensure that you get the message encoding you want.

See Also:

The following sections in the *Oracle Application Server Advanced Web Services Developer's Guide* provide more information on WSIF, EJBs and WSIF clients.

- "WSIF and EJBs" provides information on using WSIF functionality in Web services based on EJBs and WSIF extensions to the WSDL.
- "Writing a WSIF Client" provides information on how to write a client for a service that has WSIF bindings.

Table 11–5 describes the properties that can be specified for the `@WSIFEJBBinding` annotation. The `className` and `jndiName` properties are required.

Note: The properties in the following table could potentially be overridden by the deployment descriptor or by argument values passed by WebServicesAssembler. For more information, see "How to Override Annotation Values with WebServicesAssembler" on page 11-13.

Table 11–5 WSIFEJBBinding Annotation Properties

Property	Data Type	Description
className	String	(required) Specifies the class name of the EJB's home interface. **Default:** WebServicesAssembler will use the class name of the parent Ant task, if it was specified.
initialContextFactory	String	Specifies the name of the factory that will provide the initial context. **Default:** The value defaults to `com.evermind.server.rmi.RMIInitialContextFactory`.
jndiName	String	(required) Specifies the JNDI name for the EJB. **Default:** The name of the EJB that is found by querying `@Stateless.name` or the simple name of the EJB class.
jndiProviderURL	String	Specifies the URL for the JNDI Provider. **Default:** The value defaults to the empty string.
name	String	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. **Default:** n/a

The following example illustrates the `@WSIFEJBBinding` annotation being used in an interface file for a version 3.0 EJB.

```
...
@WebService(name="Hello",
            targetNamespace="http://ejb.oracle.com/targetNamespace",
            serviceName="CustomHelloBeanService")
@WSIFEJBBinding(
   className="oracle.demo.hello.HelloBean",
   jndiName="Hello",
   initialContextFactory="com.evermind.server.rmi.RMIInitialContextFactory",
   jndiProviderUrl="ormi://localhost:23791/ejb30wsif")
public interface CustomHelloBeanIntf extends Remote {
...
```

When the file is compiled, it generates an EJB WSIF binding in the WSDL, and an EJB port with the following values.

```
...
<port name="WsifEJBPort" binding="tns:WsifEjbBinding">
    <ejb:address className="oracle.demo.hello.HelloBean"
        jndiName="Hello"
        initialContextFactory="com.evermind.server.rmi.RMIInitialContextFactory"
        jndiProviderURL="ormi://localhost:23791/ejb30wsif"/>
</port>
...
```

How to Use Annotations

This section contains the following subsections:

- Steps to Use Annotations to Assemble a Web Service from Java Classes
- Steps to Use Annotations to Assemble a Web Service from a Version 3.0 EJB
- How to Override Annotations
- Sample Java File with Annotations

Steps to Use Annotations to Assemble a Web Service from Java Classes

To assemble a Web service from a Java class with J2SE 5.0 annotations, follow these steps.

1. Add Web service metadata annotations to the implementation class.

   Example 11–4 illustrates a sample annotated Java file.

   a. At a minimum, the `@WebService` annotation must be present in the class.

   b. Add the `endpointInterface` property to the `@WebService` annotation in the implementation class if you want to reference a service endpoint interface. Note that if the service endpoint interface also has an `@WebService` annotation, the annotations will be processed at that starting point instead.

   c. Add the `@WebMethod` annotation to each method you want to expose in the Web service. Note that if your annotations are in the service endpoint interface, then all of the methods will be exposed, regardless of whether they have the `@WebMethod` annotation

> **Note:** If you enter an `@WebService` annotation on a service endpoint interface and no `@WebMethod` annotations, then all of the methods will be exposed regardless of the number of `@WebMethod` annotations you enter.
>
> If you enter an `@WebService` annotation on an implementation bean and no `@WebMethod` annotations, then you should get a verification exception, because you must have at least one method exposed. See Sections 4.1 and 4.2 of the Web Services Metadata for the Java Platform specification.

2. Compile the annotated classes.

   The classes must be compiled with a JDK 5.0-compliant compiler.

3. Generate the Web service artifacts (such as the WSDL, the deployment descriptors, and so on) by using the WebServicesAssembler `assemble` command.

   a. You must specify the implementation class as the value of the `assemble` command's `className` argument. This is true even if all of your annotations are in the service endpoint interface. The implementation class must have an `@WebService` annotation.

   b. If you only want to generate a WSDL from an annotated Java class, use the `genWsdl` command and supply the implementation class as the value of the `className` argument.

Steps to Use Annotations to Assemble a Web Service from a Version 3.0 EJB

Web services can be assembled from EJB 3.0-compliant beans at deployment time. To generate a Web service endpoint from an EJB 3.0 bean, follow these steps.

1. Add an `@Stateless` annotation to the bean.

   Only stateless EJB session beans are supported.

   The `name` property of the `@Stateless` annotation provides the name of the Web service endpoint. If this property is not specified, then the default endpoint name will be the unqualified class name of the EJB bean class.

2. Add the `@WebService` annotation to the interface that the bean implements.

   a. Enter the annotation in the interface. Do not enter it in the actual bean class.

   b. At a minimum, the interface must contain the `@WebService` annotation.

   c. Add an `@WebMethod` annotation to each method in the bean implementation that you want to expose in the Web service.

> **Note:** If you enter an `@WebService` annotation on a service endpoint interface and no `@WebMethod` annotations, then all of the methods will be exposed regardless of the number of `@WebMethod` annotations you enter.
>
> If you enter an `@WebService` annotation on an implementation bean and no `@WebMethod` annotations, then you should get a verification exception, because you must have at least one method exposed. See Sections 4.1 and 4.2 of the Web Services Metadata for the Java Platform specification.

d. The interface must be implemented by the EJB bean class.

3. (Optional) Add the `@Deployment` annotation to the interface with values for the `uriPath` and `portName` properties.

   By default, the port name and default URI for an EJB 3.0 bean is the EJB name. This name is found in the `@Stateless` annotation of the bean. If the name of the bean is not specified in the `@Stateless` annotation, the short class name of the bean will be used instead. You can override this by providing values for the `uriPath` and `portName` properties of the `@Deployment` annotation.

4. Compile the EJB classes and package them into an EAR file. Deploy the EAR file to a running instance of OC4J.

How to Override Annotations

This section describes how you can override annotation values in a Java file by using deployment descriptors or WebServicesAssembler.

- How to Override Annotation Values with WebServicesAssembler
- How to Override Deployment Annotation Values with Deployment Descriptors

How to Override Annotation Values with WebServicesAssembler

Command line arguments passed to the WebServicesAssembler `assemble` and `genWsdl` commands will override any annotations in the Java class file that perform the same function. For example, if you pass the `portName` argument to the `assemble` or `genWsdl` command, then its value will override the value of the `@Deployment.portName` annotation.

Table 11–6 WebServicesAssembler Arguments that can Override Annotation Values

This WebServicesAssembler argument	Will override this annotation
`context`	`@Deployment.contextPath`
`portName`	`@Deployment.portName`
`portTypeName`	`@WebService.name` (the `@WebService.name` annotation is used to derive the `portType` name in the WSDL. See section 5.11 of the Web Services Metadata for the Java Platform specification.
`qualifiedElementForm`	`@Schema.elementFormDefault`
`restSupport`	`@Deployment.restSupport`
`serviceName`	`@WebService.serviceName`
`targetNamespace`	`@WebService.targetNamespace`
`typeNamespace`	`@Schema.targetNamespace`
`uri`	`@Deployment.uriPath`

How to Override Deployment Annotation Values with Deployment Descriptors

If you want to override any of the properties specified by the Oracle proprietary `@Deployment` annotation, you can package a deployment descriptor that provides an override in the EAR file in `META-INF/oracle-webservices.xml` (or `WEB-INF/oracle-webservices.xml` for Web modules).

When you use an `oracle-webservices.xml` deployment descriptor to override `@Deployment` annotation properties, each `<webservice-description>` element

in the descriptor must match a Web service being deployed based on the `serviceName` of the Web service. The `serviceName` is specified by `@WebService.serviceName` annotation.

Note that if you are assembling multiple Web services that use the `@Deployment` annotation and you specify a deployment descriptor that overrides the properties for only one of the services, then the other services will not be affected. In the deployment descriptor, you should specify only those properties you want to override.

For example, the values in the following XML fragment from an `oracle-webservices.xml` deployment descriptor override the `@Deployment` annotation properties. Since this fragment provides values for the `<port-component name>` attribute and the `<endpoint-address-uri>` element, the annotation's `portName` and `uriPath` properties will be overridden. The `<context-root>` element was not specified in this example, so the annotation's `contextPath` property will not be overridden.

```
<webservice-description name="CustomSessionBeanService">
    <port-component name="CustomSession">
          <endpoint-address-uri>/custom-session</endpoint-address-uri>
    </port-component>
</webservice>
```

Of course, instead of annotating the class you could just provide an `oracle-webservices.xml` deployment descriptor to specify the deployment properties.

Sample Java File with Annotations

To generate Web services from Java classes, you can enter J2SE 5.0 annotations in either of these files.

- the endpoint class only
- the endpoint class and the service endpoint interface

If you choose to enter the annotations in both the endpoint class and the service endpoint interface, then you need to enter only minimal annotations in the endpoint class.

Example 11–4 illustrates an example of an implementation class with Web service metadata annotations. Note how the `@WebService`, `@WebMethod`, and `@WebParam` annotations are used in the example.

- The `@WebService` annotation enables you to provide values for the `serviceName` and `targetNamespace` properties. These properties are used to populate the `wsdl:service name` and `wsdl:targetNamespace` elements in the generated WSDL file. The `targetNamespace` property provides the default namespace for the WSDL and the schema types.

Note: The target namespace for schema elements and types will be the same as that specified in `@WebService.targetNamespace` or the generated namespace of the WSDL. For a document-literal Web service, you can assign operation parameters to different namespaces by using the `@WebParam.targetNamespace` annotation property. For example:

```
@WebParam(name="param", targetNamespace="http://mytns")
```

Note that in this case, you must set the `@WebParam.name` property. For document-literal wrapped services, the wrapper element will always be created in the same target namespace as the WSDL.

Optionally, you can specify the `endpointInterface` property with the fully-qualified class name of an service endpoint interface. In this case, provide the following annotations.

- Enter only the `@WebService` annotations in the endpoint class, and enter the correct value for `endpointInterface`. In the case where `endpointInterface` is used, all Web service-related annotations are ignored, except for the `@WebService` annotation on the endpoint class.
- Enter the `@WebMethod` and `@WebParam` annotations in the service endpoint interface.

- The `@WebMethod` annotations identify the methods that should be present in the Web service. These annotated methods will be assembled into `wsdl:operation` elements in the generated WSDL. If a value for the `operationName` property is specified in the `@WebMethod` annotation, then it will be used as the operation name for the `wsdl:operation`. If a value is not provided, then the name of the method is used by default.
- The `@WebParam` annotations identify message parts or parameters to `wsdl:operations`. An optional `Mode` can be specified for each parameter. The `INOUT` or `OUT` modes can be specified only for parameters that implement the `javax.xml.rpc.holders.Holder` interface. By default, a parameter that implements the `Holder` interface becomes an `INOUT` parameter, unless `Mode.OUT` is explicitly specified by an annotation. All other parameters must be `IN` parameters.

Example 11–4 Sample Java File with J2SE 5.0 Web Service Metadata Annotations

```
package oracle.webservices.examples.annotations;

import java.rmi.RemoteException;
import javax.jws.*;
import javax.jws.WebParam.Mode;

@WebService (
    serviceName = "annotatedBank",
    targetNamespace = "http://service.annotatedBank"
)
public class BankServiceImpl {
    @WebMethod (operationName="create-account")
    public String createAccount( @WebParam(name="accountName") String acctName,
                                 float initBalance )
        throws RemoteException, AccountException {
        return m_bank.addNewAccount(acctName,initBalance);
```

```
    }

    @WebMethod
    public void deposit( @WebParam(name="accountID", mode=Mode.IN) String acctID,
                         float amount )
        throws RemoteException, AccountException {
        Account theAccount = m_bank.getAccount(acctID);
        if ( theAccount == null ) {
            throw new AccountException("No account found for " + acctID);
        }
        theAccount.deposit(amount);
    }
    //class truncated…
}
```

Limitations

See "Assembling Web Services with Annotations" on page F-8.

Additional Information

For more information on:

- using the Test Page to test Web service deployment, see Chapter 13, "Testing Web Services Deployment".
- building J2SE clients, see Chapter 15, "Assembling a J2SE Web Service Client".
- building J2EE clients, see Chapter 14, "Assembling a J2EE Web Service Client".
- JAX-RPC handlers, see Chapter 16, "Using JAX-RPC Handlers".
- JAR files that are needed to assemble a client, see Appendix A, "Web Service Client APIs and JARs".
- Web services interoperability, see "Ensuring Interoperable Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- using quality of service features in Web service clients, see "Managing Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- adding security to a Web service, see the *Oracle Application Server Web Services Security Guide*.
- adding reliability to a Web service, see "Ensuring Web Service Reliability" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- adding an auditing and logging configuration to a Web service, see "Auditing and Logging Messages" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- processing nonstandard data types, see "Custom Serialization of Java Value Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- the JAX-RPC mapping file, see "JAX-RPC Mapping File Descriptor" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- data types supported by OracleAS Web Services, see "Mapping Java Types to XML and WSDL Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- Oracle JDeveloper tool support for Web service development, see the Oracle JDeveloper on-line help.

12

Assembling REST Web Services

This chapter describes how Oracle Application Server Web Services supports the generation and use of REST Web Services. This chapter contains the following sections:

- Understanding REST Web Services
- WebServicesAssembler Support for REST Web Services
- How to Assemble REST Web Services
- How to Test REST Web Services
- REST Additions to Deployment Descriptors
- How REST Builds Requests and Responses
- Tool Support for REST Web Services

Understanding REST Web Services

REST (Representational State Transfer) Web services architecture conforms to the Web architecture defined by the World Wide Web Consortium (W3C), and uses its architectural principles. It employs the semantics of HTTP whenever possible. REST Web services use XML documents, not SOAP envelopes, for sending messages. Unlike SOAP Web Services, REST is a "style" and has no standards or tools support from vendors.

When you use Oracle Application Server Web Services to expose an endpoint by using REST, that endpoint is also exposed as a SOAP port. OracleAS Web Services limits the support of HTTP commands to GET and POST. REST Web services deploy like any other OracleAS Web Services.

> **Note:** OracleAS Web Services can assemble REST Web services only where the use, or encoding mechanism, is literal (`use=literal`). It does not support REST Web services where the message format is encoded.

Web service management features, such as security and reliability, are not available with REST Web service invocations (either REST or SOAP). This is because SOAP headers, which are typically used to carry this information, cannot be used with REST Web service invocations.

WebServicesAssembler Support for REST Web Services

This section contains the following subsections:

- Commands that Can Assemble REST Web Services
- Annotations that Support REST Web Services

Commands that Can Assemble REST Web Services

You can use WebServicesAssembler to add REST Web service capabilities to any Web application that can use HTTP as a protocol. This includes Web service applications built on Java classes, EJBs, and database resources. WebServicesAssembler provides a boolean `restSupport` argument that will allow any of the following commands to assemble a REST Web service.

- aqAssemble
- assemble
- corbaAssemble
- dbJavaAssemble
- ejbAssemble
- plsqlAssemble
- sqlAssemble
- topDownAssemble

Annotations that Support REST Web Services

An optional boolean `restSupport` attribute can be applied to the Oracle proprietary `@Deployment` annotation. If the value of the attribute is `true`, then the port to which the annotation is applied will support REST-style `GET` and `POST` requests and responses.

> **See Also:**
>
> "Oracle Additions to J2SE 5.0 JDK Annotations" on page 11-6 for more information on the `restSupport` attribute and the `Deploy` tag.

How to Assemble REST Web Services

This section contains the following subsections:

- Steps to Assemble a REST Web Service Top Down
- Steps to Assemble a REST Web Service Bottom Up

Steps to Assemble a REST Web Service Top Down

The following steps illustrate assembling a REST Web service from a WSDL. This example provides only an outline of the steps required for top down Web service assembly.

> **See Also:**
>
> Chapter 6, "Assembling a Web Service from WSDL" for a detailed description of each of the steps for assembling a Web service from a WSDL (top down).

1. Provide a WSDL from which the Web service will be generated as input to the WebServicesAssembler `genInterface` command.

The WSDL used in this example is illustrated in Example 12–1. The following is a sample `genInterface` command.

```
java -jar wsa.jar -genInterface
                  -output build/src/service
                  -wsdl wsdl/MovieFacility.wsdl
                  -unwrapParameters false
```

In this command:

- `genInterface`—Creates a service endpoint interface for each port type and a Java value type class (beans) for any complex type defined in a WSDL. It also creates a JAX-RPC mapping file that describes the mapping between the XML schema types and the Java value type classes. See "genInterface" on page 18-31.
- `output`—Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See"output" on page 18-48.
- `wsdl`—Specifies the absolute file path, relative file path, or URL to a WSDL document. See "wsdl" on page 18-65.
- `unwrapParameters`—This argument can be set only for document-literal operations and will be ignored for other message formats. When set to `false` the generated service endpoint interface will be generated with wrappers around the input parameter and the return type. See "unwrapParameters" on page 18-69.

2. Compile the generated interface and type classes.
3. Implement the Java service endpoint for the Web service you want to provide.
4. Compile the Java service endpoint.
5. Generate the service by running the WebServicesAssembler tool with the `topDownAssemble` command.

   Set the `restSupport` argument to `true`. For example:

```
java -jar wsa.jar -topDownAssemble
            -wsdl ./wsdl/MovieFacility.wsdl
            -unwrapParameters false
            -className oracle.webservices.examples.rest.RpcLitMovieImpl
            -input build/classes/service
            -output build
            -ear dist/rpclit_topdown.ear
            -restSupport true
```

   In this command:

   - `topDownAssemble`—Creates the required classes and deployment descriptors for a Web service based on a WSDL description. The files can be stored in either an EAR file, a WAR file, or a directory. See "topDownAssemble" on page 18-22.
   - `wsdl`—Specifies the absolute file path, relative file path, or URL to a WSDL document. See "wsdl" on page 18-65.
   - `unwrapParameters`—This argument can be set only for document-literal operations and will be ignored for other message formats. When set to `false` the generated service endpoint interface will be generated with wrappers around the input parameter and the return type. See "unwrapParameters" on page 18-69.

- `className`—Specifies the name of the class (including the package name) that is the implementation class for the Web service. See "className" on page 18-43.
- `input`—Specifies the directory or JAR containing the classes that should be copied to `WEB-INF/classes`. This argument will be added to the classpath used by the WebServicesAssembler. See "input" on page 18-46.
- `output`—Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See "output" on page 18-48.
- `ear`—Specifies the name and location of the generated EAR. See "ear" on page 18-44.
- `restSupport`—Specifies whether REST (Representational State Transfer) support will be enabled for this Web service. See "restSupport" on page 18-49.

6. 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*.

7. (Optional) Check that deployment succeeded.

   OracleAS Web Services provides a Web Service Test Page for each deployed Web service. The Test Page enables you to generate and invoke any REST POST or GET requests. See "How to Use the Web Services Test Page" on page 13-6 for information on accessing and using the Web Service Test Page.

> **Note:** REST Web services do not use J2EE or J2SE clients. However, since every REST endpoint is also a SOAP endpoint, you can assemble J2SE or J2EE clients for those endpoints. For examples of how to create GET or POST REST messages, use the Web Services Test Page.

How to Access REST Web Service Operations

Example 12–1 illustrates a fragment of the WSDL used to assemble the RPC-literal service in "Steps to Assemble a REST Web Service Top Down".

Example 12–1 WSDL Fragment for an RPC-Literal Web Service

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:tns0="http://www.oracle.com/rest/doc/types"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
xmlns:tns="http://www.oracle.com/rest" name="rest-service"
targetNamespace="http://www.oracle.com/rest"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/">
    <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/"
xmlns:tns="http://www.oracle.com/rest/doc/types" elementFormDefault="qualified"
targetNamespace="http://www.oracle.com/rest/doc/types">
            <complexType name="Movie">
                <sequence>
                    <element name="Title" type="xsd:string"/>
                    <element name="Director" type="xsd:string"/>
```

```
                    <element name="Year" type="xsd:int" />
                </sequence>
            </complexType>
            <complexType name="ArrayOfMovie">
                <sequence>
                    <element name="Movie" type="tns:Movie"
                                    maxOccurs="unbounded" />
                </sequence>
            </complexType>
        </schema>
    </types>
    <message name="FindMoviesRequest">
        <part name="TitleWords" type="xsd:string" />
        <part name="Year" type="xsd:int" />
    </message>
    <message name="FindMoviesResponse">
        <part name="Movies" type="tns0:ArrayOfMovie" />
    </message>
    <message name="AddMovieRequest">
        <part name="Movie" type="tns0:Movie" />
    </message>
    <message name="AddMovieResponse">
        <part name="Added" type="xsd:boolean" />
    </message>
    <portType name="MovieDB">
         <operation name="findMovies">
             <input message="tns:FindMoviesRequest" />
             <output message="tns:FindMoviesResponse" />
         </operation>
         <operation name="addMovie">
             <input message="tns:AddMovieRequest" />
             <output message="tns:AddMovieResponse" />
         </operation>
     </portType>

    <binding name="HttpSoap11Binding" type="tns:MovieDB">
        <soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="findMovies">
            <soap:operation soapAction="http://www.oracle.com/rest/findMovies"/>
            <input>
                <soap:body use="literal" parts="TitleWords Year"
namespace="http://www.oracle.com/rest"/>
            </input>
            <output>
                <soap:body use="literal" parts="Movies"
namespace="http://www.oracle.com/rest"/>
            </output>
        </operation>
        <operation name="addMovie">
            <soap:operation soapAction="http://www.oracle.com/rest/addMovie"/>
            <input>
                <soap:body use="literal" parts="Movie"
namespace="http://www.oracle.com/rest"/>
            </input>
            <output>
                <soap:body use="literal" parts="Added"
namespace="http://www.oracle.com/rest"/>
            </output>
        </operation>
```

```
    </binding>
    <service name="rest-service">
        <port name="HttpSoap11" binding="tns:HttpSoap11Binding">
            <soap:address
location="http://localhost:8888/webservice/webservice/"/>
        </port>
    </service>
</definitions>
```

Using the preceding WSDL, the procedure described in "Steps to Assemble a REST Web Service Top Down" produces the following generated interface:

```
interface MovieDb {
    public Movie[] findMovies (String titleWords, int year);
    public boolean addMovie (Movie movie);
}
```

The first method in the generated interface has only simple parameters. This method can be invoked with an HTTP GET. For example:

```
http://{yourhost}/{context-path}/{service-url}/findMovies?TitleWords=Star+Wars&Yea
r=1977
```

This query string returns the following XML response:

```
<ns0:findMoviesResponse xmlns:ns0="http://www.oracle.com/rest">
   <Movies>
      <ns1:Movie xmlns:ns1="http://www.oracle.com/rest/doc/types">
         <ns1:Title>tim</ns1:Title>
         <ns1:Director>tim</ns1:Director>
         <ns1:Year>1978</ns1:Year>
      </ns1:Movie>
   </Movies>

</ns0:findMoviesResponse>
```

The `addMovie` method in the generated interface takes a complex parameter; it can only be invoked with an HTTP POST. For example, you can POST the following XML message to the URL of your Web service, `http://{yourhost}/{context-path}/{service-url}`.

```
<ns0:addMovie xmlns:ns0="http://www.oracle.com/rest">
   <Movies>
      <ns1:Movie xmlns:ns1="http://www.oracle.com/rest/doc/types">
         <ns1:Title>tim</ns1:Title>
         <ns1:Director>tim</ns1:Director>
         <ns1:Year>1978</ns1:Year>
      </ns1:Movie>
   </Movies>
</ns0:addMovie>
```

Steps to Assemble a REST Web Service Bottom Up

The following steps illustrate assembling a REST Web service from Java classes. This example provides only an outline of the steps required for bottom up Web service assembly.

See Also:

Chapter 7, "Assembling a Web Service with Java Classes" for a detailed description of each of the steps for assembling a Web service with Java classes.

1. Provide the compiled Java class that you want to expose as a Web service and its compiled interface.

   Example 12–2 illustrates the `StringTools` interface that is used in this example.

2. Generate the service artifacts by specifying the `assemble` command with the `restSupport` argument to `true`.

```
java -jar wsa.jar -assemble
           -appName tools
           -serviceName StringTools
           -interfaceName oracle.webservices.examples.rest.StringTools
           -className oracle.webservices.examples.StringToolsImpl
           -input ./build/classes/service
           -output build
           -use literal
           -ear dist/tools.ear
           -uri StringToolsService
           -restSupport true
```

   In this command:

   - `assemble`—Generates a Web service from Java files bottom up. The command creates all of the files required to create a deployable archive. See "assemble" on page 18-7.
   - `appName`—Specifies the name of an application. Usually, this name is used as a base value for other arguments like `context` and `uri`. See "appName" on page 18-42.
   - `serviceName`—Specifies the service name. See "serviceName" on page 18-50.
   - `interfaceName`—Specifies the name of a Java class (including the package name) that contains the service endpoint interface (SEI). See "interfaceName" on page 18-47.
   - `className`—Specifies the name of the class (including the package name) that is the implementation class for the Web service. See "className" on page 18-43.
   - `input`—Specifies the directory or JAR containing the classes that should be copied to `WEB-INF/classes`. This argument will be added to the classpath used by the WebServicesAssembler. See "input" on page 18-46.
   - `output`—Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See "output" on page 18-48.
   - `use`—For bottom up Web service assembly, this argument specifies the `use` attribute of the message format in the generated WSDL. See "use" on page 18-68.
   - `ear`—Specifies the name and location of the generated EAR. See "ear" on page 18-44.
   - `uri`—Specifies the URI to use for the Web service. See "uri" on page 18-63.

- `restSupport`—Specifies whether REST (Representational State Transfer) support will be enabled for this Web service. See "restSupport" on page 18-49.

3. 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*.

4. (Optional) Check that deployment succeeded. OracleAS Web Services provides a Web Service Test Page for each deployed Web service.

   The Test Page enables you to generate and invoke any REST POST or GET requests. See "How to Use the Web Services Test Page" on page 13-6 for information on accessing and using the Web Service Test Page.

> **Note:** REST Web services do not use J2EE or J2SE clients. However, since every REST endpoint is also a SOAP endpoint, you can assemble J2SE or J2EE clients for those endpoints. For examples of how to create GET or POST REST messages, use the Web Services Test Page.

How to Access REST Web Service Operations

Example 12–2 illustrates the `StringTools` interface that is used to assemble the REST Web service in Step 2 described earlier.

Example 12–2 Interface Used to Assemble REST Web Services

```
interface StringTools {
package oracle.webservices.examples.rest;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface StringTools extends Remote{
    public String appendStrings (String a, String b) throws RemoteException;
    public String toUpperCase (String c) throws RemoteException;
    public String concatArrayOfStrings (String s[]) throws RemoteException;
```

The first two methods in the interface, `appendStrings` and `toUpperCase`, use atomic parameters. As REST Web service operations, these operations can be accessed with HTTP GET. The following example illustrates a call to the `appendStrings` operation, if `document` style is specified at assembly time:

```
http://{yourserver}/{context-path}/{service-URL}/appendStrings?String_
1=Hello+&String_2=World
```

This query string would return the following XML response:

```
<appendStringsResponseElement xmlns="http://oracle.webservices.examples.rest/">
     <result>Hello World</result>
</appendStringsResponseElement>
```

The third method in the interface, `concatArrayOfStrings` takes a non-atomic parameter. As a REST Web service operation, it cannot be called using HTTP GET. It can only be called with HTTP POST. For example:

```
<ns1:concatArrayOfStringsElement
xmlns:ns1="http://oracle.webservices.examples.rest/">
            <ns1:arrayOfString_1>a,</ns1:arrayOfString_1>
            <ns1:arrayOfString_1>b.</ns1:arrayOfString_1>
```

```
</ns1:concatArrayOfStringsElement>
<concatArrayOfStringsRequest xmlns="http://oracle.webservices.examples.rest/">
```

This request string would return the following XML code:

```
<ns0:concatArrayOfStringsResponseElement
xmlns:ns0="http://oracle.webservices.examples.rest/">
   <ns0:result>a,b.</ns0:result>
</ns0:concatArrayOfStringsResponseElement>
```

How to Test REST Web Services

You can use the Web Services Test Page to test whether REST Web services deployed successfully.

See Also:

"Invoking the Test for a REST Web Service" on page 13-14 for more information on using the Web Services Test Page.

REST Additions to Deployment Descriptors

If REST support is enabled for a Web service, then an optional boolean `<rest-support>` subelement is added to the `<port-component>` element of the `oracle-webservices.xml` deployment descriptor. If `<rest-support>` is set to `true`, then the port to which the subelement is applied will support REST-style `GET` and `POST` requests and responses. The default value is `false`.

See Also:

"<rest-support>" on page C-10 for additional information on the `<rest-support>` subelement.

How REST Builds Requests and Responses

The following sections describe how REST Web service requests are formed on the client side and how they are processed on the server side.

HTTP GET Requests

Suppose a SOAP endpoint is deployed at the following URL:

```
http://example.com/my-app/my-service
```

If this endpoint is REST enabled, then HTTP GET requests will be accepted at the following URL:

```
http://example.com/my-app/my-service/{operationName}?{param1}={value1}&{param2}={value2}
```

In this example, `{operationName}` is one of the operation names in the WSDL for the service. For RPC-literal operations, `{param1}`, `{param2}`, and so on, are the part names defined in the operation's input `wsdl:message`. Note that these must be `simpleTypes` (`xsd:int`, and so on).

> **Note:** Some browsers limit the size of the HTTP GET URL (typically less than 2000 characters). Try to keep the size of the URL small by using a limited number of parameters and short parameter names and values.

For document-literal operations, messages have only a single parameter. To simulate multiple parameters, the WSDL specifies a single parameter that is defined in the schema as a `sequence`. Each member of the sequence is considered a parameter. In this case, {*param1*}, {*param2*}, and so on, will be the members of the sequence type, instead of `message` parts. As with RPC, these must be `simpleTypes`.

Example 12–3 illustrates the request message defined for an operation named `addNumbers`.

Example 12–3 GET Request on an Operation

```
<wsdl:message name="AddNumbersRequest">
      <wsdl:part name="a" type="xsd:int" />
      <wsdl:part name="b" type="xsd:int" />
</wsdl:Message>
```

This request can be invoked by using a GET with the following URL:

```
http://{yourhost}/{context-path}/{service-url}/addNumbers?a=23&b=24
```

Example 12–4 illustrates the SOAP envelope that the OracleAS Web Services platform will create on the server side from the GET request. This message will be processed like any other incoming SOAP request.

Example 12–4 SOAP Envelope Created from a GET Request

```
<soap:Envelope>
    <soap:Body>
        <ns:addNumbers>
            <ns:a>23</ns:a>
            <ns:b>24</ns:b>
        </ns:addNumbers>
    <soap:Body>
<soap:Envelope>
```

Example 12–5 illustrates the request message sent for the `addNumbers` service when it is defined as a document-literal operation.

Example 12–5 Sample GET Request on an Document-Literal Wrapped Operation

```
<wsdl:message name="AddNumbersRequest">
     <wsdl:part name="params" type="tns:AddNumbersRequstObject" />
</wsdl:Message>
```

Example 12–6 illustrates the definition of the `AddNumbersRequestObject` as it would be defined in the schema.

Example 12–6 XML Definition of a Document-Literal Wrapped Operation

```
<xsd:complexType name="AddNumbersRequestObject">
    <xsd:complexContent>
        <xsd:sequence>
            <xsd:element name="a" type="xsd:int"/>
```

```
            <xsd:element name="b" type="xsd:int"/>
        </xsd:sequence>
    </xsd:complexContent>
</xsd:complexType>
```

This operation can be invoked by a GET request with the following URL. Note that this is the same URL that is used for the RPC-literal request in Example 12–3.

```
http://{yourhost}/{context-path}/{service-url}/addNumbers?a=23&b=24
```

HTTP POST Requests

REST Web services support HTTP POST requests that are simple XML messages—not SOAP envelopes. REST requests do not support messages with attachments. Since the service also supports SOAP requests, the implementation must determine if a given request is meant to be SOAP or REST.

When a SOAP service receives a POST request, it looks for a `SOAPAction` header. If it exists, the implementation will assume that it is a SOAP request. If it does not, it will find the `QName` of the root element of the request. If it is the SOAP Envelope `QName`, it will process the message as a SOAP request. Otherwise it will process it as a REST request.

REST requests will be processed by wrapping the request document in a SOAP envelope. The HTTP headers will be passed through as received, except for the Content-Type header in a SOAP 1.2 request. This Content-Type header will be changed to the proper content type for SOAP 1.2, `application/soap+xml`.

For example, the following REST request illustrated in Example 12–7 will be wrapped in the SOAP envelope illustrated in Example 12–8.

Example 12–7 REST Request

```
<ns:addNumbers>
    <ns:a>23</ns:a>
    <ns:b>24</ns:b>
</ns:addNumbers>
```

The request illustrated in Example 12–8 will be processed as a normal SOAP request.

Example 12–8 SOAP Envelope Wrapping the REST Request

```
<soap:Envelope>
       <soap:Body>
              <ns:addNumbers>
                     <ns:a>23</ns:a>
                     <ns:b>24</ns:b>
              </ns:addNumbers>
          </soap:Body>
</soap:Envelope>
```

REST Responses

For any request (either GET or POST) that was processed as a REST request, the response must also be in REST style. The OracleAS Web Services platform will transform the SOAP response on the server into a REST response before sending it to the client. The REST response will be an XML document whose root Element is the first child Element of the SOAP Body. For example, assume that the SOAP envelope illustrated in Example 12–9 exists on the server.

Example 12–9 SOAP Response

```
<soap:Envelope>
    <soap:Body>
        <ns0:result xmlns:nso="…">
            <ns:title>How to Win at Poker</ns:title>
            <ns:author>John Doe</ns:author>
        </ns0:result>
    </soap:Body>
</soap:Envelope>
```

Example 12–10 illustrates the response sent back to the client. Note that the Content-Type will always be `text/xml`. Any SOAP Headers or attachments will not be sent back to the client.

Example 12–10 REST Response

```
<ns0:result xmlns:ns0="…">
       <ns:title>How to Win at Poker</ns:title>
       <ns:author>John Doe</ns:author>
</ns0:result>
```

Tool Support for REST Web Services

The **Create Java Web Service** wizard in Oracle JDeveloper provides an option for enabling REST functionality for a Web service. For more information on using Oracle JDeveloper to enable REST functionality in a Web Service, see the Oracle JDeveloper on-line help.

Limitations

See "Assembling REST Web Services" on page F-8.

Additional Information

For more information on:

- assembling a Web service from a WSDL, see Chapter 6, "Assembling a Web Service from WSDL".
- assembling a Web service from Java classes, see Chapter 7, "Assembling a Web Service with Java Classes".
- assembling a Web service from EJBs, see Chapter 8, "Assembling a Web Service with EJBs".
- assembling a Web service from database resources, see Chapter 10, "Assembling Database Web Services".
- testing REST Web services, see Chapter 13, "Testing Web Services Deployment".

13

Testing Web Services Deployment

This chapter provides information on how to test your JAX-RPC or REST Web Service deployment. A Web service has deployed successfully if you can invoke its operations. To do this, Oracle Application Server Web Services provides a test page. You can use the Web Services Test Page to perform any of the following tasks:

- View the Web service's deployed service description (WSDL)
- Exercise Web service operations with different values
- Exercise operations for different values for Web service security and reliability
- Provide values for HTTP authentication for the request
- Perform stress testing

The following sections provide more information on accessing and using the Web Services Test Page:

- Steps to Access and Test Web Service Operations
- How to Access the Web Services Test Page
- How to Use the Web Services Test Page
- How to Disable the Web Services Test Page
- How to Obtain a Web Service WSDL Directly

See Also:

Chapter 12, "Assembling REST Web Services" for more information about REST Web Services.

Steps to Access and Test Web Service Operations

The following steps describe how to access and use the functionality in the Test Page for a JAX-RPC or REST Web Service. The following sections provide more detail on each step.

1. Access the Test Page for a Web Service.

   You can access the Test Page through Application Server Control or through a Web browser. See "How to Access the Web Services Test Page" on page 13-2 for more information on this step.

2. Select the operation you want to test from the **Operation** drop down list.
3. Enter the parameter values for the Web Service operation you want to test.

See "Editing Values in the Test Page" on page 13-9 for more information on this step.

4. (Optional) Enter parameter values to exercise security and reliable messaging features (for JAX-RPC Web services only).

   See "Testing WS-Security and Reliable Messaging Features" on page 13-10 for more information on this step.

5. (Optional) Enter HTTP authentication values to access the Web service.

   See "Enabling HTTP Authentication for the Web Service Test" on page 13-11 for more information on this step.

6. (Optional) Enter parameters to enable stress testing.

   See "Stress Testing the Web Service Operation" on page 13-12 for more information on this step.

7. Click the **Invoke** button to send the test message to the service endpoint.

   You can invoke the request to a JAX-RPC or REST Web service as a SOAP request. For a REST Web service, you can invoke it as a SOAP request, or an HTTP POST or HTTP GET operation. See "Invoking the Web Service Test" on page 13-13 for more information on this step.

How to Access the Web Services Test Page

You can access the Test Page either by entering the URI for a JAX-RPC or REST Web service directly in a Web browser or by using Application Server Control.

- Accessing the Web Services Test Page through a Web Browser
- Accessing the Web Services Test Page through Application Server Control
- Accessing a Web Services Test Page Protected with SSL

Accessing the Web Services Test Page through a Web Browser

To access a Test Page for a JAX-RPC or REST Web service directly, you can enter the address of its service endpoint in a Web browser. The address has the format:

`http://host:port/context-root/service`

Table 13–1 describes the components of the address.

Table 13–1 URL Components for Accessing the Test Page

URL Component	Description
context-root	The value specified in the `<context-root>` element for the Web module associated with the Web service. See the `META-INF/application.xml` in the Web service's EAR file to determine this value.
host	The host name of the Web service's server running OracleAS Web Services.
port	The port name of the Web service's server running OracleAS Web Services.
service	The value specified in the `<url-pattern>` element for the servlet associated with the Web service. This is the service name. See the `WEB-INF/web.xml` file in the Web service's WAR file to determine this value.

Accessing the Web Services Test Page through Application Server Control

You can use Application Server Control to access the Test Page for a particular JAX-RPC or REST Web service. The following steps provide a brief overview of how to use the tool to navigate to the Test Page. For more detailed information, see the topic "Web Services Page" in the Application Server Control on-line help.

1. Open Application Server Control
2. Click the **Web Services** tab on the **OC4J: home** page.
3. In the table of Web services, click the **Select** radio button associated with the Web service application and port that you want to test
4. Click the **Test Service** button to open **Test Web Service:** ***port_name*** page.

   Enterprise Manager displays the Test Web service page, which lists the Web sites defined in your current environment.
5. Click the **Select** radio button to select a Web site listener for the Web service.

   Enterprise Manager generates a URL based on information about the context root defined for the application and the Web site you selected. The URL appears in the URL field.
6. Click the **Test Web Service** button to open the Web Service Test Page.

Accessing a Web Services Test Page Protected with SSL

If the Web Services Test Page is protected by SSL (Secure Sockets Layer), you must perform some additional configuration steps in order to use it. The steps are different depending on whether OracleAS Web Services is running on Oracle HTTP Server.

- If you are running OracleAS Web Services as a standalone application, you only need to add keystore and truststore configuration properties to the `opmn.xml` file. This is the main configuration file for the Oracle Process Manager and Notification Server (OPMN).
- If you are running OracleAS Web Services on Oracle HTTP Server, in addition to editing the `opmn.xml` file you must export the user certificate in Oracle Wallet and import it to the OC4J keystore as a trusted certificate.

  See Also:

  - *Oracle Process Manager and Notification Server Administrator's Guide* for more information on OPMN.
  - *Oracle Containers for J2EE Security Guide* for more information on keystore and truststore properties.

The following sections describe how to make the necessary changes so that you can access the SSL-protected Test page.

- Accessing the SSL-Protected Test Page when OracleAS Web Services Runs as Standalone
- Accessing the SSL-Protected Test Page when OracleAS Web Services runs on Oracle HTTP Server

Accessing the SSL-Protected Test Page when OracleAS Web Services Runs as Standalone

If you are running OracleAS Web Services as a standalone application, follow these steps to access an SSL-protected Web Services Test Page.

1. Locate the `opmn.xml` file in your installation.

   The `opmn.xml` file can be found here:

   ORACLE_HOME`/opmn/conf/opmn.xml`

   Here, *ORACLE_HOME* is the directory where you installed OracleAS Web Services.

   It is assumed that the file has already been configured for SSL.

2. Add the following properties to the OC4J `start-parameters` section of `opmn.xml`, then save the file. These properties are described in Table 13–2. See Example 13–1 for a sample `opmn.xml` file that uses these properties.

   - `-Djavax.net.ssl.trustStore=<`*full path to OC4J keystore file*`>`
   - `-Djavax.net.ssl.trustStorePassword=<`*OC4J keystore password*`>`
   - `-Djavax.net.ssl.keyStore=<`*full path to OC4J keystore file*`>`
   - `-Djavax.net.ssl.keyStorePassword=<`*OC4J keystore password*`>`

3. Restart OPMN.

   For example, the following command will stop, then restart all OPMN processes.

   prompt`> opmnctl startall`

4. Enter the URL for the Web Service's Test page in a browser.

5. Enter values to test the service and press **Invoke** button

You should receive the expected SOAP message response.

Table 13–2 describes the properties supported by Oracle HTTPS for keystores and truststores.

Table 13–2 Keystore and Truststore Properties

Property Name	Description
javax.net.ssl.keyStore	Specifies the location and name of the keystore file or wallet file to use as the keystore. This property can be set to point to the text wallet file exported from Oracle Wallet Manager that contains the credentials that are to be used for a specific connection. If no other credentials have been set for the HTTPS connection, then the file indicated by this property is opened when a handshake first occurs. If any errors occur as this file is read, then the connection fails and an `IOException` is thrown. If this property has no setting, the application is responsible for verifying that the certificate chain contains a certificate that can be trusted.
javax.net.ssl.keyStorePassword	This property can be set to indicate the password that is necessary to open the keystore (keystore file or wallet file).

Table 13–2 (Cont.) Keystore and Truststore Properties

Property Name	Description
javax.net.ssl.trustStore	This property is used similarly to javax.net.ssl.keyStore, but specifies the location and name of the keystore file or wallet file to use as the truststore (a file that includes the trusted certificate authorities that a client will implicitly accept).
javax.net.ssl.	This property is used similarly to `javax.net.ssl.keyStorePassword`, but specifies the password that is necessary to open the truststore (keystore file or wallet file).

Example 13–1 illustrates a fragment of an opmn.xml file that contains the keystore and truststore properties. In this example, `<ias-component id="default-group">` indicates the start of the OC4J start parameters section.

Example 13–1 opmn.xml Fragment with Keystore and Truststore Properties

```
<opmn>
 ...
  <ias-component id="default-group">
   <process-type id="home" module-id="OC4J" status="enabled">
    <module-data>
      <category-id="start-parameters">
        <data-id="java-options" value=".. -Djavax.net.ssl.trustStore=<full path to
OC4J keystore file>, -Djavax.net.ssl.trustStorePassword=<OC4J keystore password>,
-Djavax.net.ssl.keyStore=<full path to OC4J keystore file>,
-Djavax.net.ssl.keyStorePassword=<OC4J keystore password>, ...>
      </category>
       ...
    </module-data>
   </process-type>
  </ias-component>
 ...
</opmn>
```

Accessing the SSL-Protected Test Page when OracleAS Web Services runs on Oracle HTTP Server

If you are running OracleAS Web Services on a Oracle HTTP Server, follow these steps to access an SSL-protected Web Services Test Page.

> **Note:** In this procedure, you use the Oracle Wallet Manager and Java keytool utilities to export and import certificates. A discussion of how to use these utilities is beyond the scope of this manual. For detailed information, see the following resources:
>
> - For Oracle Wallet Manager, see "Using Oracle Wallet Manager" in the *Oracle Database Advanced Security Administrator's Guide*. This document is available from the Oracle Server Technologies group.
> - For the Java keytool utility, see "keytool - Key and Certificate Management Tool" at the following Web site.
>
>   `http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/keytool.html`

1. Locate the `opmn.xml` file in your installation.

The `opmn.xml` file can be found here:

ORACLE_HOME`/opmn/conf/opmn.xml`

Here, *ORACLE_HOME* is the directory where you installed OracleAS Web Services.

It is assumed that the file has already been configured for SSL.

2. Add the following properties to the `OC4J start-parameters` section of `opmn.xml`, then save the file. These properties are described in Table 13–2 in the previous section. See Example 13–1 for a sample `opmn.xml` file that uses these properties.
   - `-Djavax.net.ssl.trustStore=<`*full path to OC4J keystore file*`>`
   - `-Djavax.net.ssl.trustStorePassword=<`*oc4j keystore password*`>`
   - `-Djavax.net.ssl.keyStore=<`*full path to OC4J keystore file*`>`
   - `-Djavax.net.ssl.keyStorePassword=<`*oc4j keystore password*`>`
3. Use Oracle Wallet Manager to export the user certificate from the Oracle HTTP Server wallet file.

   You can find the Oracle Wallet manager here:

   ORACLE_HOME`/bin/owm`

   The following is a sample path to a wallet file:

   ORACLE_HOME`/Apache/Apache/conf/ssl.wlt/default/ewallet.p12`

   You can refer to Oracle Wallet Manager documentation for how to export the certificate.
4. Use the Java keytool utility to import the certificate into the OC4J keystore as a trusted certificate.
5. Restart OPMN.

   For example, the following command will stop, then restart all OPMN processes.

   ```
   prompt> opmnctl startall
   ```
6. Enter the URL for the Web Service's Test page in a browser.
7. Enter values to test the service and press **Invoke** button

You should receive the expected SOAP message response.

How to Use the Web Services Test Page

Oracle Application Server Web Services provides a Test Page for each deployed Web service. The Test Page can be used to exercise the operations exposed by either JAX-RPC Web services or REST services.

- Viewing the Web Service WSDL
- Editing Values in the Test Page
- Testing WS-Security and Reliable Messaging Features
- Enabling HTTP Authentication for the Web Service Test
- Stress Testing the Web Service Operation

- Invoking the Web Service Test

Figure 13–1 displays the Web Service Test Page for the `sayHello` operation of the `HelloServiceEJB` JAX-RPC Web service.

Figure 13–1 Web Service Test Page

Viewing the Web Service WSDL

To view a JAX-RPC or REST Web service's WSDL, click the **Service Description** link. You can use your Web browser's **File>Save As** operation to save the WSDL locally.

Click the **Return** button to return to the Test Page.

Figure 13–2 illustrates a fragment of a WSDL for a JAX-RPC Web service.

Figure 13–2 WSDL Fragment for a JAX-RPC Web Service

```
<?xml version="1.0" encoding="UTF-8" ?>
- <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:ns1="http://oracle.j2ee.ws/ejb/Hello/types"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    xmlns:tns="http://oracle.j2ee.ws/ejb/Hello" name="HelloServiceEJB"
    targetNamespace="http://oracle.j2ee.ws/ejb/Hello">
  - <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/"
        xmlns:tns="http://oracle.j2ee.ws/ejb/Hello/types"
        elementFormDefault="qualified"
        targetNamespace="http://oracle.j2ee.ws/ejb/Hello/types">
      - <complexType name="sayHello">
        - <sequence>
            <element name="String_1" nillable="true" type="string" />
          </sequence>
        </complexType>
      - <complexType name="sayHelloResponse">
        - <sequence>
            <element name="result" nillable="true" type="string" />
          </sequence>
        </complexType>
        <element name="sayHelloElement" type="tns:sayHello" />
        <element name="sayHelloResponseElement" type="tns:sayHelloResponse" />
      </schema>
    </types>
  - <message name="HelloServiceInf_sayHello">
      <part name="parameters" element="ns1:sayHelloElement" />
```

Figure 13–3 illustrates a fragment of a WSDL for a REST Web service.

Figure 13–3 WSDL Fragment for a REST Web Service

```
<?xml version="1.0" encoding="UTF-8" ?>
- <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns0="http://oracle.demo/types/"
    xmlns:java="http://schemas.xmlsoap.org/wsdl/java/"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:format="http://schemas.xmlsoap.org/wsdl/formatbinding/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    xmlns:tns="http://oracle.demo/" name="HelloWebService"
    targetNamespace="http://oracle.demo/">
  - <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/"
        xmlns:tns="http://oracle.demo/types/"
        targetNamespace="http://oracle.demo/types/" elementFormDefault="qualified">
      - <complexType name="sayHelloTo">
        - <sequence>
            <element name="name" type="string" nillable="true" />
          </sequence>
        </complexType>
      - <complexType name="sayHelloToResponse">
        - <sequence>
```

Editing Values in the Test Page

The Test Page enables you to test any of the operations exposed by a JAX-RPC or REST Web service. By default, the editable parameters and attributes of the operation are displayed in an HTML form. Optional parameters and attributes on the form are indicated by a checkbox. Select the checkbox to provide a value for the parameter.

> **Note:** To distinguish attributes from parameters in the HTML Form, attributes are preceded with an "**@**" symbol.

Use the **Operation** drop-down list to select the Web service operation that you want to test.

A "**+**" link in the form indicates that you can add additional copies of a structure.

An "**x**" link in the form indicates that you can remove copies of a structure that you do not want.

How to Edit the Test Page as XML Source

Instead of entering values for a JAX-RPC or REST Web service operation in an HTML form, you can enter them directly in XML source code. To do this, select the **XML Source** radio button on the Test Page.

Note:

- If you enter values in the XML source, you do not have to precede attributes with **@** symbols.
- If you enter values for an operation in XML source, then toggle to **HTML Form** mode, the values you entered will not be preserved. The form will display, cleared of any values.

Testing WS-Security and Reliable Messaging Features

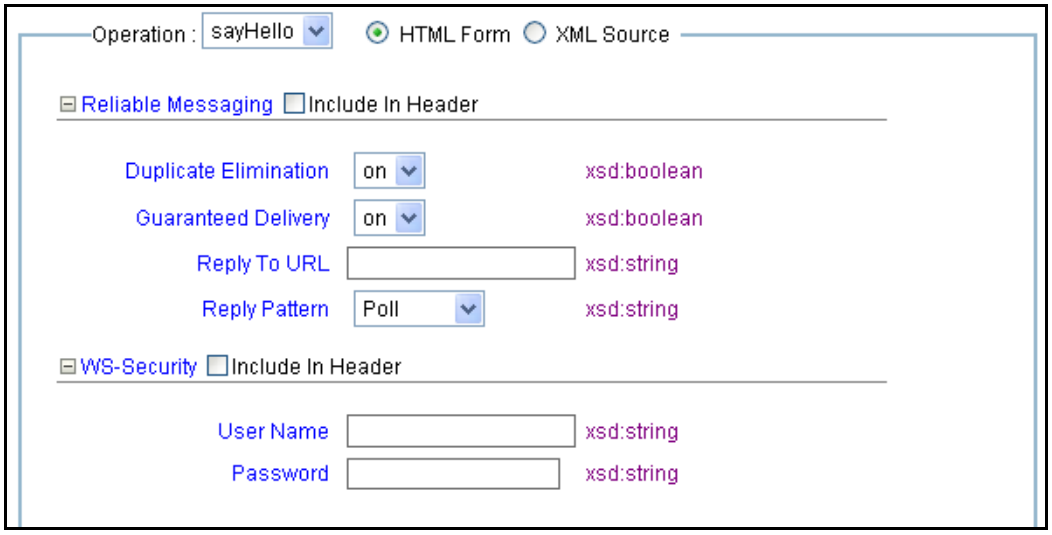
You can invoke a JAX-RPC Web service operation with different values for security and reliability. To expose these parameters, expand the nodes next to **Reliable Messaging** and **WS-Security**.

Note: REST Web Services, as implemented by OracleAS Web Services, do not support security or reliability features. If you specify any of the security or reliability options for a REST Web service request, they will be ignored.

Figure 13–4 illustrates the parameters and default values for security and reliability in the HTML form.

Tip: After you provide values for the parameters, you can click the **XML Source** radio button to see the reliability and security headers inserted into the SOAP request.

Figure 13–4 WS-Security and Reliable Message Components in the Test Page



Reliable Messaging Parameters

Expand the **Reliable Messaging** node to indicate that reliability features will participate in the test. A reliability SOAP header will be inserted into the SOAP envelope of the request.

You can choose different settings for these parameters:

- **Duplicate Elimination**—turning this feature "on" will insert the **Duplicate Elimination** reliability header into the message. This tells the reliable endpoint to eliminate duplicates of the message that will be sent. The default value is **on**.
- **Guaranteed Delivery**—turning this feature "on" will insert the **Guaranteed Delivery** reliability header into the message. This tells the reliable endpoint that it must acknowledge receiving the message. The default value is **on**.
- **Reply To URL**—indicates the URL to which acknowledgments and faults will be sent for messages that want asynchronous acknowledgements. The URL is typically the host name of the client, with the port that the listener is on.
- **Reply Pattern**—indicates how the client can interact with the endpoint. The values can be **Callback** (asynchronous acknowledgment/fault), **Response** (synchronous acknowledgement/fault), or **Polling** (the acknowledgment or fault must be polled for). The default value is **Polling**.

See Also:

"Ensuring Web Service Reliability" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on reliability features.

WS-Security Parameters

Select the **WS-Security** checkbox to indicate that security features will participate in the test. A security SOAP header will be inserted into the SOAP envelope.

You can choose different settings for these parameters:

- User Name
- Password

See Also:

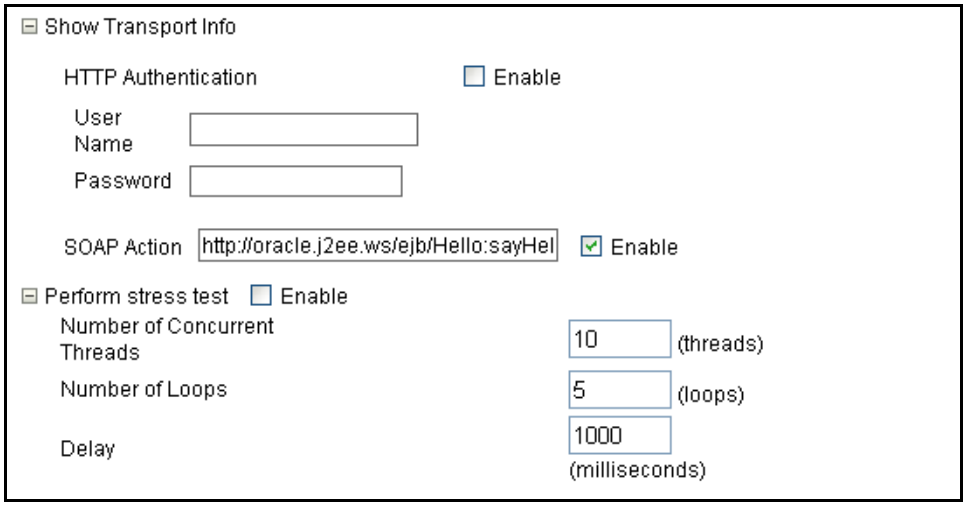
Oracle Application Server Web Services Security Guide for more information on the security features that are available for OracleAS Web Services.

Enabling HTTP Authentication for the Web Service Test

Expand the **Show Transport Info** node to display the HTTP Authentication options for the JAX-RPC or REST Web service. If the HTTP service you are testing is password protected, the parameters under **Show Transport Info** enable you to provide a user name and password. You can also specify a value for **SOAP Action** in case the service needs to provide any specialized filtering on the SOAP request.

Figure 13–5 illustrates the **Show Transport Info** parameters in the Test Page.

Figure 13–5 HTTP Authentication and Stress Test Parameters in the Test Page



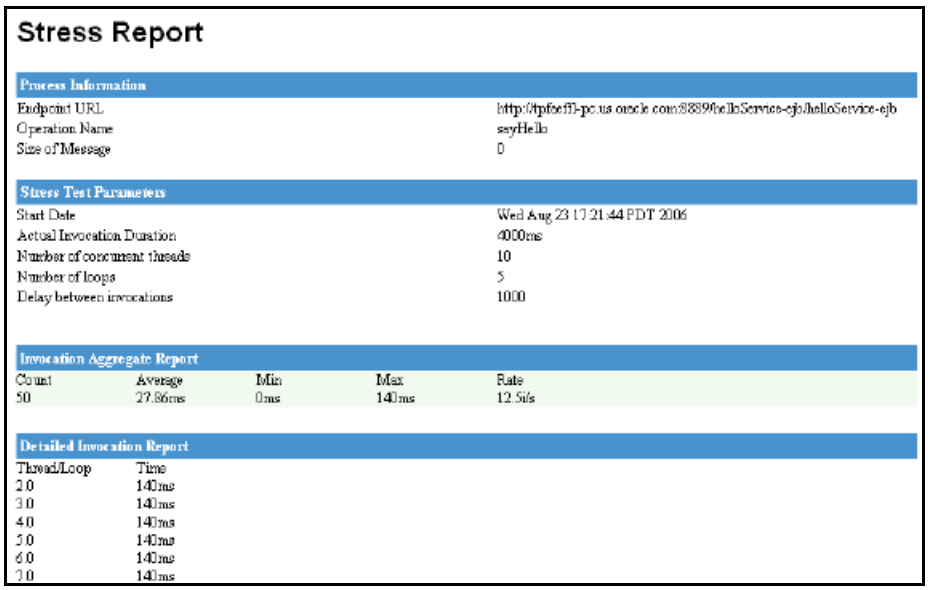
Stress Testing the Web Service Operation

Expand the **Perform Stress Test** node to display the options to create and configure a continuous series of invocations of the JAX-RPC or REST Web service operation. The **Perform Stress Test** options are displayed in Figure 13–5.

- **Number of Concurrent Threads**—The number of concurrent threads on which the invocations should be sent. The default is 10 threads.
- **Number of Loops**—The number of times to invoke the operation. The default is 5 times for each thread.
- **Delay**—Number of milliseconds to wait between operation invocations. The default is 1000 milliseconds (1 second).

When you invoke the test, a stress report page is returned. The report page identifies the service endpoint and operation being tested, the size of the message sent, the number of concurrent threads on which it is run, the number of times it is run on each thread, and the delay between each operation invocation. Figure 13–6 illustrates s fragment of a sample stress test report.

Figure 13–6 Sample Stress Test Report



Stress Report

Process Information

Endpoint URL	http://tpfseff1-pc.us.oracle.com:8889/helloService-ejb/helloService-ejb
Operation Name	sayHello
Size of Message	0

Stress Test Parameters

Start Date	Wed Aug 23 17:21:44 PDT 2006
Actual Invocation Duration	4000ms
Number of concurrent threads	10
Number of loops	5
Delay between invocations	1000

Invocation Aggregate Report

Count	Average	Min	Max	Rate
50	27.86ms	0ms	140ms	12.5/s

Detailed Invocation Report

Thread/Loop	Time
2 0	140ms
3 0	140ms
4 0	140ms
5 0	140ms
6 0	140ms
7 0	140ms

Invoking the Web Service Test

After you have specified the parameter values and functionality you want to test for the operation, you can send the message to the Web service endpoint. For a JAX-RPC Web service, the message is sent to the service as a SOAP request. For a REST Web service, the message can be sent as either a SOAP request, or an XML REST GET, or POST operation.

- Invoking the Test for a JAX-RPC Web Service
- Invoking the Test for a REST Web Service

Invoking the Test for a JAX-RPC Web Service

Click the **Invoke** button to send the message as a SOAP request to the JAX-RPC Web service endpoint. The Test Page displays the response from the service. The response can be displayed in formatted XML (default) or as raw XML (wire format).

Figure 13–7 Response from a JAX-RPC Web Service

Invoking the Test for a REST Web Service

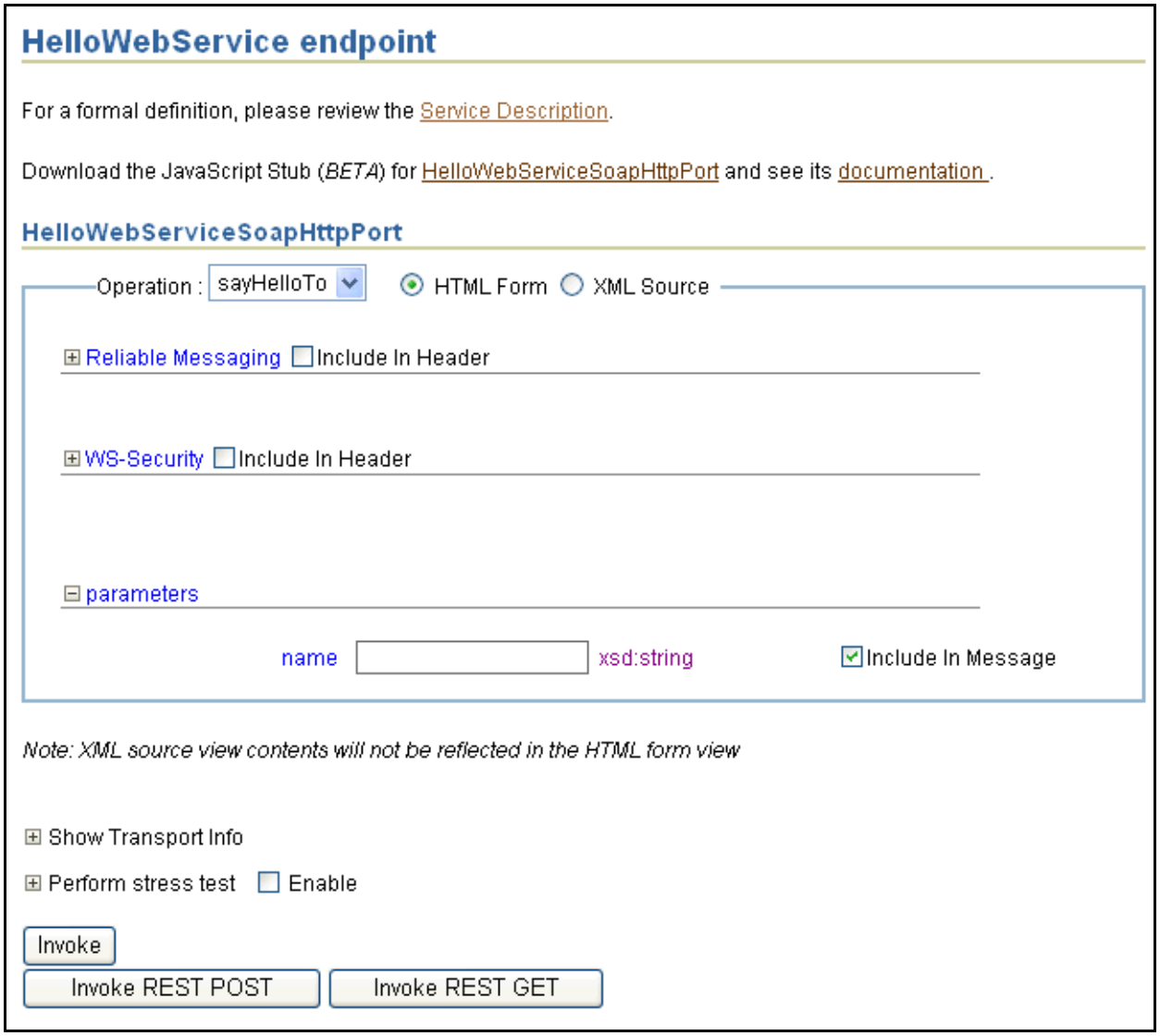
The Test Page for a REST Web service enables you to send the test message to the REST service as either an XML REST POST or GET operation. In addition, OracleAS Web Services allows you the option of sending the message as a SOAP request.

The Test Page provides the following buttons that enable you to invoke the Web service operation on the test message.

- Invoke—invokes the request as a SOAP request.
- Invoke REST POST—generates and invokes a REST POST request.
- Invoke REST GET—redirects the page to the GET URL.

Figure 13–8 illustrates a Test Page for a REST-enabled Web service operation.

Figure 13–8 Test Page for a REST Web Service Operation



Invoke Click the **Invoke** button to send the XML REST request to the service as a SOAP message over HTTP. The service returns a SOAP response message to the Test Page application. The response can be displayed in either formatted XML (default) or raw XML (wire format).

Figure 13–9 illustrates a SOAP response from a REST Web service in formatted XML.

Figure 13–9 Response from a REST Service as a SOAP Response in Formatted XML

Invoke REST POST Click **Invoke REST POST** on the Test Page to send the message to the REST Web service. The response is returned to the Test Page application. The response can be displayed in either formatted XML (default) or raw XML (wire format).

Note:

- REST Web Services, as implemented by OracleAS Web Services, do not support security or reliability features. If you specify these options for a REST Web service request, they will be ignored.
- REST Web service responses do not include the SOAP `Envelope` or `Body` elements.

Figure 13–10 illustrates a REST Web service response from a REST POST operation.

Figure 13–10 Response from a REST POST Operation in Formatted XML

Invoke REST GET Click **Invoke REST GET** on the Test Page to send the request to the service as an HTML GET command in the Web browser. The response is displayed in the browser without the Test Page application.

Note:

- REST Web Services, as implemented by OracleAS Web Services, do not support security or reliability features. If you specify these options for a REST Web service request, they will be ignored.
- REST Web service responses do not include the SOAP `Envelope` or `Body` elements.

Figure 13–11 illustrates the REST Web service response from a REST GET operation.

Figure 13–11 Response from a REST GET Operation

How to Disable the Web Services Test Page

Disabling the Test Page allows you to increase security by reducing the externally visible details of an application that exposes Web services.

To disable the Test Page, follow these general steps:

1. Extract the `oracle-webservices.xml` deployment descriptor from the JAR file before deployment.
2. Insert the boolean `<expose-testpage>` element set to `false` in the `<webservice-description>` section of the deployment descriptor.

   Example 13–2 illustrates this step.
3. Replace the edited `oracle-webservices.xml` deployment descriptor in the JAR file.
4. Deploy the JAR file as usual.

Note: WebServicesAssembler will overwrite this entry if you reassemble the Web service with a deployment descriptor passed in with the `ddFileName` command.

Example 13–2 illustrates a fragment of an `oracle-webservices.xml` deployment descriptor that disables the Test Page.

Example 13–2 Disabling the Web Services Test Page in the oracle-webservices.xml File

```
<oracle-webservices>
  ...
  <webservice-description name="with-test-page-blocked">
    <expose-testpage>false</expose-testpage>
     ...
   </webservice-description>
```

```
</oracle-webservices>
```

How to Obtain a Web Service WSDL Directly

If you do not use the Web Services Test Page to get the WSDL file for a Web service, you can obtain it directly.

To obtain the WSDL, use the Web service URL and append a query string. The format for the URL to obtain the WSDL service description is:

`http://`*`host:port/context-root/service`*`?WSDL`

`WSDL` can be either uppercase or lowercase. Table 13–1 on page 13-2 contains a description of the URL components.

This URL returns a WSDL description in the form *service*`.wsdl`. The *service*`.wsdl` description contains the WSDL for the Web service named *service*, located at the specified URL. Using the WSDL, you can build a client application to access the Web service.

Limitations

See "Testing Web Service Deployment" on page F-9.

Additional Information

For more information on Web services that can use the Web Services Test Page, see the following chapters:

- Chapter 6, "Assembling a Web Service from WSDL"
- Chapter 7, "Assembling a Web Service with Java Classes"
- Chapter 8, "Assembling a Web Service with EJBs"
- Chapter 10, "Assembling Database Web Services"
- Chapter 12, "Assembling REST Web Services"
- "Using Web Service Providers" in the *Oracle Application Server Advanced Web Services Developer's Guide*.

For more information on Web service security and reliability, see these resources:

- for adding security to a Web service, see the *Oracle Application Server Web Services Security Guide*.
- for adding reliability to a Web service, see "Ensuring Web Service Reliability" in the *Oracle Application Server Advanced Web Services Developer's Guide*.

14

Assembling a J2EE Web Service Client

This chapter describes how to develop a Web service client from within a J2EE container. Any component of a J2EE 1.4-compliant container, such as a version 2.4 servlet, 2.1 EJB, or 2.0 JSP application can act as a J2EE Web service client.

This chapter contains the following sections:

- Understanding J2EE Web Service Clients
- How to Assemble a J2EE Web Service Client
- Writing J2EE Web Service Client Code
- Understanding the Packaging Structure for J2EE Clients

See Also:

Chapter 15, "Assembling a J2SE Web Service Client" for information on assembling a Web service client that runs in the J2SE environment.

Understanding J2EE Web Service Clients

The J2EE platform provides an environment that allows client applications to access Web services. In a J2EE environment, the deployment descriptors define the client-side Web service access information. This access information can be changed at deployment time. In addition, the J2EE platform handles the underlying work of creating and initializing access to Web services.

J2EE Web service clients inherit the advantages of the J2EE platform, such as declarative security, transactions, and instance management. In addition to these platform characteristics, the OracleAS Web Services management framework makes it possible to configure SOAP logging and auditing, WS-Reliability, and WS-Security.

Unlike the J2SE Web service client, the J2EE client resides in and is managed by the OC4J container. It requires no proxy code generation or packaging. You get a portable client application with Web service access that is easy to embed in JSPs, servlets, and EJBs. EJB variants such as Container Managed Persistence (CMP), Bean Managed Persistence (BMP), and Message-Driven Beans (MDB) can call out to Web service endpoints.

How to Assemble a J2EE Web Service Client

Required information—These subsections contain the steps that are required for assembling a J2EE Web service client:

- Prerequisites

- Steps to Assemble a J2EE Web Service Client
- How to Add J2EE Web Service Client Information to Deployment Descriptors

Optional information—These subsections may be optional, depending on the type of client you are writing, or the functionality they employ:

- Steps to Deploy and Run an Application Client Module
- How to Access a Web Service from a Client in the Same Module
- How to Add OC4J-Specific Platform Information for Deployment and Runtime
- How to Add JAX-RPC Handlers for Message Processing to Deployment Descriptors

Prerequisites

Before you begin, supply the following files and information.

- The WSDL file or location from which the service endpoint interface and JAX-RPC mapping file will be generated.
- The location where the generated service endpoint interface and JAX-RPC mapping file will be stored.

Steps to Assemble a J2EE Web Service Client

Use the WebServicesAssembler tool to assemble a service endpoint interface and the J2EE Web service client. Then, edit the deployment descriptor to add Web service access information. The following steps describe these tasks in more detail.

1. Provide the WSDL and the information described in the Prerequisites section as input to the WebServicesAssembler `genInterface` command.

   In the following example, the `HelloService.wsdl` is used to generate `HelloInterface` in the `oracle.demo.hello` package.

   Command line:

   ```
   java -jar wsa.jar
        -genInterface
        -wsdl HelloService.wsdl
        -output build
        -packageName oracle.demo.hello
   ```

   Ant task:

   ```
   <oracle:genInterface wsdl="${etc.web1.dir}/HelloService.wsdl"
           output="build"
           packageName="oracle.demo.hello"
   />
   ```

   In this command and Ant task:

   - `genInterface`—Creates a service endpoint interface for each port type and a Java value type class (beans) for any complex type defined in a WSDL. It also creates a JAX-RPC mapping file that describes the mapping between the XML schema types and the Java value type classes. See "genInterface" on page 18-31.
   - `wsdl`—Specifies the absolute file path, relative file path, or URL to a WSDL document. See "wsdl" on page 18-65.

- `output`—Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See "output" on page 18-48.
- `packageName`—Specifies the package name that will be used for generated classes if no package name is declared in the JAX-RPC mapping file. See "packageName" on page 18-48.

2. Edit the deployment descriptor of the J2EE component to add a `<service-ref>` element. This element captures all of the Web service access information.

   See "How to Add J2EE Web Service Client Information to Deployment Descriptors" on page 14-4 for a sample of the `<service-ref>` element and its subelements.

   If the client also provides message processing in the form of JAX-RPC handlers, then these also must be added to the deployment descriptor. See "How to Add JAX-RPC Handlers for Message Processing to Deployment Descriptors" on page 14-12 for more information on adding handler information to the deployment descriptor.

3. Assemble the client deployment module into an EAR file:

   a. Compile all the client files.

   b. Copy deployment descriptor files to their appropriate locations.

      For example, for an EJB, copy the WSDL to `META-INF/wsdl/`, the JAX-RPC mapping file and the deployment files such as `ejb-jar.xml` and `orion-ejb-jar.xml` to `META-INF`, and so on. For a description of where files should reside for servlet, EJB, or JSP Web service clients, see "Understanding the Packaging Structure for J2EE Clients" on page 14-17.

   c. Package the client deployment module.

   Note: The current tool set cannot package J2EE Web service clients. You must package the client manually. "Understanding the Packaging Structure for J2EE Clients" on page 14-17 provides more information on how to package a J2EE Web service client.

4. Deploy the client deployment module.

   The following steps will deploy an EJB, JSP, or other J2EE client. If you are deploying an application client, skip these steps and continue with "Steps to Deploy and Run an Application Client Module".

   a. Start OC4J. The following is a sample command to start OC4J.

   ```
   java -jar oc4j.jar
   ```

   b. Deploy the client module into OC4J. The following is a sample deployment command.

   ```
   java -jar admin_client.jar deployer:oc4j:<oc4jHost>:<oc4jPort> <adminID>
   <adminPassword>
            -deploy
            -file .\client\myClient.ear
            -deploymentName myClient
            -bindWebApp default-web-site
   ```

   The *`oc4jHost`* and *`oc4jPort`* variables are the host name and port number of the OC4J server. The *`adminID`* and *`adminPassword`* are the OC4J server

user name and password. The following are sub-switches of the `-deploy` switch.

- `file`—path and filename of the EAR file to deploy.
- `deploymentName`—user-defined application deployment name, used to identify the application within OC4J.
- `bindWebApp`—specifies the Web site to which the web application should be bound. This is the Web site that will be used to access the application.

5. Run the EJB or JSP client.

   If you are running an application client, see "Steps to Deploy and Run an Application Client Module".

How to Add J2EE Web Service Client Information to Deployment Descriptors

You must edit the J2EE component's deployment descriptor to add information that allows the component to access the Web service endpoint.

- For an EJB 2.1 Web service client, edit the `META-INF/ejb-jar.xml` deployment descriptor.
- For a JSP 2.0 or servlet Web service client, edit the `WEB-INF/web.xml` deployment descriptor.
- For an application client, edit the `META-INF/application-client.xml` deployment descriptor.

Edit the deployment descriptor to add a `<service-ref>` element. By adding this element, you can employ an EJB, JSP, or servlet as a Web service client that can invoke a remote Web service. The `<service-ref>` element and its subelements capture all the Web service access information, such as the location of the WSDL and mapping file, the service interface, the service ports, their related service endpoint interfaces, and so on. For a complete listing of all the information that can be included in the `<service-ref>` element, see the `service-ref` (J2EE client) schema.

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

Example 14–1 illustrates a sample `<service-ref>` element that has been added to a `web.xml` deployment descriptor for the `MyHelloService` Web service. The `<service-ref>` subelements in this example are described in Table 14–1. Note that this sample `<service-ref>` uses only a subset of all of the Web service access information available in the schema.

Example 14–1 Contents of a Sample service-ref Element

```
<service-ref>
    <service-ref-name>service/MyHelloServiceRef</service-ref-name>
    <service-interface>javax.xml.rpc.Service</service-interface>
    <wsdl-file>WEB-INF/wsdl/HelloService.wsdl</wsdl-file>
    <jaxrpc-mapping-file>WEB-INF/HelloService-java-wsdl-mapping.xml
    </jaxrpc-mapping-file>
    <service-qname xmlns:service-qname_ns__="http://hello.demo.oracle/">
    service-qname_ns__:HelloService</service-qname>
    <port-component-ref>
         <service-endpoint-interface>oracle.demo.hello.HelloInterface
         </service-endpoint-interface>
         <port-component-link></port-component-link>
    </port-component-ref>
</service-ref>
```

Table 14–1 describes the `<service-ref>` subelements used in this sample.

Table 14–1 Subelements of the <service-ref> Element

service-ref Subelement	Description
<jaxrpc-mapping-file>	Specifies the fully-qualified path to the JAX-RPC mapping file.
<port-component-link> (optional)	If the Web service is implemented in the same module as the client, you can add this element to access the service. For more information on this element, see "How to Access a Web Service from a Client in the Same Module".
<port-component-ref>	Declares a client dependency on the container for resolving a service endpoint interface to a WSDL port. It optionally associates the service endpoint interface with a particular port-component. The container uses this only for a `Service.getPort(Class)` method call.
<service-endpoint-interface>	Specifies the fully-qualified Java class that represents the service endpoint interface of a WSDL port.
<service-interface>	Specifies the fully-qualified class name of the JAX-RPC Service interface the client depends on. In most cases the value will be `javax.xml.rpc.Service`. A JAX-RPC generated Service Interface class may also be specified.
<service-qname>	Specifies the service `QName` for the service: ▪ `xmlns:ns`—maps to the `targetNamespace` value in the WSDL. ▪ `ns`—maps to the `service name` attribute in the WSDL.
<service-ref-name>	Specifies the JNDI path and service name assigned by the client.
<wsdl-file>	Specifies the fully-qualified path to the WSDL file.

Steps to Deploy and Run an Application Client Module

The following steps describe how to deploy and run an application client module. Unlike EJB, JSP, or other J2EE clients, you must specify the directory where the generated `deployment-cache.jar` will be stored. You must also specify the location of the `deployment-cache.jar` in the run command.

1. Start OC4J. The following is a sample command to start OC4J.

   ```
   java -jar oc4j.jar
   ```

2. Deploy the application client module into OC4J.

   The following is a sample deployment command.

   ```
   java -jar admin_client.jar deployer:oc4j:<oc4jHost>:<oc4jPort> <adminID>
   <adminPassword>
            -deploy
            -file .\client\myAppClient.ear
            -deploymentName myAppClient
            -deploymentDirectory C:\home\myDir
   ```

This command creates a `deployment-cache.jar` file and stores it in `C:\home\myDir`.

The *`oc4jHost`*, *`oc4jPort`*, *`adminID`*, and *`adminPassword`* variables and the `file` and `deploymentName` sub-switches of `-deploy` are described in **Step 4b** in "Steps to Assemble a J2EE Web Service Client" on page 14-2.

The `deploymentDirectory` sub-switch indicates the location where OC4J deploys `deployment-cache.jar`. In this example, OC4J deploys it into `C:\home\myDir`. If you do not specify this sub-switch, OC4J deploys the application into the *`OC4J_HOME`*`/application-deployments/` directory. If you supply the empty string (" "), OC4J will always read the deployment configurations from the EAR file each time the application is deployed.

3. Run the client deployment module. For an application client, the location of the `deployment-cache.jar` must be present in the classpath. The following is a sample run command:

```
java -classpath .:C:\home\myDir\deployment-cache.jar:'oc4jclient.jar'
  :appclient.jar oracle.myappclient.classname
```

In this sample, it is assumed that `appclient.jar` contains the class `oracle.myappclient.`*`classname`*.

How to Access a Web Service from a Client in the Same Module

To enable the Oracle Application Server to access a Web service that resides in the same module as the client, add the `<port-component-link>` element to the `<service-ref>` clause of the client deployment descriptor and add the `PortComponentLinkResolver` property to the `system-application.xml` configuration. The following steps summarize these tasks.

1. Add the `<port-component-link>` element to the `<service-ref>` clause in the J2EE client deployment descriptor.

   "How to Add a Port Component Link to a J2EE Client Deployment Descriptor" provides more information on this step.

2. Shut down the Oracle Application Server.

3. Add the `PortComponentLinkResolver` property to the `system-application.xml` server configuration file. This file resides in the directory *`ORACLE_HOME`*`/j2ee/home/config`.

   Add the following lines to this file.

```
<ejb-module id="PortComponentLinkResolver"
path="../../../webservices/lib/wsserver.jar"/>
```

4. Re-start the Oracle Application Server.

How to Add a Port Component Link to a J2EE Client Deployment Descriptor

If the Web service resides in the same container as the client, then you can access the service by adding the `<port-component-link>` element to the `<service-ref>` clause of the J2EE client deployment descriptor (`web.xml`, `ejb-jar.xml`, or `application-client.xml`).

The `<port-component-link>` element links a `<port-component-ref>` to a specific port component in the server-side deployment descriptor. The `<port-component-name>` element resides in the server-side deployment descriptor, `webservices.xml`.

The following examples illustrate this relationship. The `webservices.xml` fragment in Example 14–2 illustrates the deployment configuration for the Web service that exposes the EJB `InterModuleEjb`. In this fragment, the port component is named `InterPC`. Example 14–3 illustrates a fragment of a client-side deployment descriptor where this name is referenced from the `<port-component-link>` element in the `<service-ref>` clause. The presence of this element allows the J2EE client to look up the Web service.

These examples assume that the Web service is running in the same container as the J2EE Web service client.

Example 14–2 webservices.xml Fragment, Identifying a Port Component Name

```
<webservices>
  <webservice-description>
    <webservice-description-name>InterModuleEjb</webservice-description-name>
    <wsdl-file>META-INF/wsdl/InterModuleService.wsdl</wsdl-file>
    <jaxrpc-mapping-file>META-INF/InterModuleService.xml</jaxrpc-mapping-file>
    <port-component>
      <port-component-name>InterPC</port-component-name>
      <wsdl-port
        xmlns:wsdl1="http://PortCompLink.org/ejb/inter">wsdl1:InterModuleSeiPort
      </wsdl-port>
      <service-endpoint-interface>oracle.demo.InterModuleSei
      </service-endpoint-interface>
      <service-impl-bean>
        <ejb-link>InterModuleEjb</ejb-link>
      </service-impl-bean>
    </port-component>
</webservices>
```

Example 14–3 illustrates a fragment of a client-side deployment descriptor where the value of the `<port-component-name>` element in the server-side deployment descriptor is referenced from the `<port-component-link>` element in the `<service-ref>` clause. The presence of this element allows the client to look up the Web service.

Note that the value of the port component name in the `<port-component-link>` is prefixed by `PortCompLinkEjb-ejb.jar#`. This value qualifies the name of the EJB with the JAR file where it resides.

Example 14–3 <port-component-link> Element in the Client-Side Deployment Descriptor

```
  <service-ref>
    <service-ref-name>service/portcomplink/inter</service-ref-name>
    <service-interface>javax.xml.rpc.Service</service-interface>
    <wsdl-file>META-INF/wsdl/InterModuleService.wsdl</wsdl-file>
    <jaxrpc-mapping-file>META-INF/InterModuleService.xml</jaxrpc-mapping-file>
    <port-component-ref>
       <service-endpoint-interface>oracle.demo.InterModuleSei
       </service-endpoint-interface>
       <port-component-link>PortCompLinkEjb-ejb.jar#InterPC</port-component-link>
    </port-component-ref>
  </service-ref>
```

How to Add OC4J-Specific Platform Information for Deployment and Runtime

The `<service-ref-mapping>` element can appear as a subelement of the `<orion-web-app>` element in the `orion-web.xml`, `orion-ejb-jar.xml`, or

orion-application-client.xml proprietary deployment descriptor files. It defines OC4J-specific runtime and deployment-time generated settings for a Web service reference. You can use this element to configure the following information:

- configure the client to consume stateful Web Services
- configure the client to make JMS transport calls
- configure security, logging, and auditing quality of service (QOS) features for the corresponding Web service.

The <service-ref-mapping> element is used in conjunction with the <service-ref> element that appears in the standard deployment descriptors. The <service-ref> element contains the information that lets you employ an EJB, JSP, or servlet as a Web service client that can invoke a remote Web service.

Note that whenever a <service-ref> element can appear in a web.xml, ejb-jar.xml, or application-client.xml file, a corresponding <service-ref-mapping> element can appear in an orion-web.xml, orion-ejb-jar.xml, or orion-application-client.xml file.

The <service-ref-mapping> element's supported features are described in the service-ref-mapping-10_0.xsd that is imported into the orion-web, orion-ejb-jar, and orion-application-client XSDs. Currently, there is no tool support, such as Oracle JDeveloper wizards, for providing values to service-ref-mapping-10_0.xsd. You must refer to the schema and edit the XML file by hand.

In its simplest case, the <service-ref-mapping> element contains only deployment information. Do not add run-time or quality of service elements if you do not want your client to be managed. A managed client is more expensive in terms of performance.

Example 14–4 contains a sample <service-ref-mapping> fragment that can be appear in a proprietary deployment descriptor. So that you can see this element's hierarchy, all the subelements are displayed. The tables following the XML sample describe the subelements.

Example 14–4 Sample <service-ref-mapping> Segment

```
...
<service-ref-mapping name="service/MyJAXRPCTime">
  <service-impl-class>oracle.demo.MyTime_Impl</service-impl-class>
  <wsdl-file final-location="file:/myhome/mytime/client-wsdl/MyJAXRPCTime.wsdl">
  <wsdl-location wsdl-override-last-modified=19NOV>
  <service-qname namespaceURI="urn:oracle-ws" localpart="MyService" />
  <stub-property>
   <name>...</name>
   <value>...</value>
  </stub-property>
  <call-property>
     <name>javax.xml.rpc.service.endpoint.address</name>
     <value>http://myhost:8888/time-ejb/timeport</value>
  </call-property>
  <port-info>
    <wsdl-port>
    <service-endpoint-interface>time.TimeService</service-endpoint-interface>
    <stub-property>
      <name>another.endpoint.address</name>
      <value>http://anotherhost:8888/time-ejb/timeport</value>
    </stub-property>
```

```
    <call-property>
      <name>...</name>
      <value>...</value>
    </call-property>
    <runtime>...</runtime>
    <operations>
      <operation name="echo">
         <runtime>
           <auditing request="true" response="false" fault="false"/>
           <reliability><reject-non-reliable-messages
value="false"/></reliability>
           ...
         </runtime>
     </operation>
   </operations>
  </port-info>
</service-ref-mapping>
...
```

Table 14–2 describes the subelements of `<service-ref-mapping>`.

Table 14–2 Subelements of the <service-ref-mapping> Element

Element Name	Description
<call-property>	Defines the call property values applicable to all ports. This is a convenient way to specify a property without specifying the port name. The `name` and `value` subelements of `<call-property>` are described in Table 14–6 on page 14-12. Note that the `<port-info>` element also contains a `<call-property>` element. If call property values are specified for a particular port inside the `<port-info>` tag, then they will override the values set here.
<port-info>	Defines a port within a service-reference. See Table 14–3 on page 14-10 for a description of the subelements of `<port-info>`.
<service-impl-class>	Defines a deployment-time generated name of a `Service` implementation.
<service-qname>	Derived at deployment-time, this element contains the `QName` of the Web service.
<stub-property>	Defines the stub property values applicable to all ports. This is a convenient way to specify a property without specifying the port name. The `name` and `value` subelements of `<stub-property>` are described in Table 14–6 on page 14-12. Note that the `<port-info>` element also contains a `<stub-property>` element. If stub property values are specified for a particular port inside the `<port-info>` tag, then they will override the values set here.
<wsdl-file>	Defines a deployment-time generated name for the WSDL file. This element has this attribute: ▪ `final-location`—points to the copy of the WSDL document associated with the `service-ref` in the standard deployment descriptor.

Table 14–2 (Cont.) Subelements of the <service-ref-mapping> Element

Element Name	Description
<wsdl-location>	(Optional) Contains a valid URL pointing to a WSDL document. If a URL is specified, then the WSDL document at this URL will be used during deployment instead of the WSDL document associated with the `service-ref` in the standard deployment descriptor. Sample values for `<wsdl-location>` include: `http://hostname:port/myservice/myport?WSDL` and `file:/home/user1/myfinalwsdl.wsdl`. This element has this attribute: ▪ `wsdl-override-last-modified`—this optional string value is generated at deployment time and lists the time when the WSDL file was last modified.

Table 14–3 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 `<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 14–3 Subelements of the <port-info> Element

Element Name	Description
<call-property>	Defines the call property values applicable to the port defined by the `<port-info>` element. The `name` and `value` subelements of `<call-property>` are described in Table 14–6 on page 14-12. Note that the `<service-ref-mapping>` element also contains a `<call-property>` subelement (described in Table 14–2 on page 14-9). If call property values are specified for a particular port inside the `<port-info>` tag, then they override the `<call-property>` element values set under `<service-ref-mapping>`.
<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. For a description of the `<operations>` subelement, see Table 14–4 on page 14-11.
<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.
<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.

Table 14–3 (Cont.) Subelements of the <port-info> Element

Element Name	Description
<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 14–6 on page 14-12. Note that the `<service-ref-mapping>` element also contains a `<stub-property>` subelement (described in Table 14–2 on page 14-9). If stub property values are specified for a particular port inside the `<port-info>` tag, then they override the `<stub-property>` element values set under `<service-ref-mapping>`.
<wsdl-port>	Specifies the name of a port in the WSDL that the container will use for container-managed port selection. In container-managed port selection, the container manages calls to the instance directly, and the client requests a generic port that might be used to access multiple different instances. This element has the following attributes ■ `localpart`—The local part of a WSDL name. For example, `authenticateHeader`. ■ `namespaceURI`—The namespace URI of a WSDL. For example, `http://oracle.j2ee.ws/Header`

Table 14–4 describes the `<operation>` subelement of the `<operations>` element.

Table 14–4 Subelement of the <operations> Element

Element Name	Description
<operation>	Specifies client-side quality of service configuration for a particular operation provided by the referenced Web service. The configuration appears within this element's `<runtime>` subelement. The `<runtime>` subelement is described in Table 14–5. This `<operation>` element has these attributes: ■ `inputName`—contains the input name of the operation from the WSDL. It is required only if the `name` attribute cannot be used to uniquely identify the operation. ■ `name`—associates the contained quality of service configuration to a specific operation. The value of the attribute must match the operation name from the WSDL. ■ `outputName`—contains the output name of the operation from the WSDL. It is required only if the `name` and `input` attributes cannot be used to uniquely identify the operation.

Table 14–5 describes the `<runtime>` subelement of the `<operation>` element.

Table 14–5 Subelement of the <operation> Element

Element Name	Description
<runtime>	Contains client-side quality of service configuration for individual operations within the port. Each child element contains configuration for one of the quality of services features (security, reliability, and/or auditing).

Table 14–6 describes the `name` and `value` subelements of the `<stub-property>` and `<call-property>` elements.

Table 14–6 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.

How to Add JAX-RPC Handlers for Message Processing to Deployment Descriptors

J2EE Web service clients can use JAX-RPC handlers to provide additional message processing facilities for Web service endpoints. For example, you can use a handler to process a SOAP message.

You must enter the handler information as a subelement of `<service-ref>` in a J2EE Web service client's deployment descriptor. The `<handler>` element encapsulates this information.

See Also:

"Client-Side JAX-RPC Handlers" on page 16-4 for more information on client-side handlers and how to register them with the deployment descriptor.

Writing J2EE Web Service Client Code

This section contains the following subsections:

- Steps for Writing a J2EE Web Service Client
- How to Configure a J2EE Web Service Client for a Stateful Web Service
- How to Configure a J2EE Web Service Client to Make JMS Transport Calls
- How to Enable Chunked Data Transfer for HTTP 1.1
- How to Set a Character Encoding for a SOAP Message

Steps for Writing a J2EE Web Service Client

This section describes some of the common code that allows a J2EE component to access a Web service. At runtime, all J2EE Web service clients use a standard JNDI lookup to find Web services. The following steps describe the general pattern for coding a JNDI lookup that could be used within a servlet, EJB, or a JSP.

1. Create the initial JNDI context.

   ```
   Context ic = new InitialContext();
   ```

   The OC4J container sets up the initial context properties.

2. Locate the service using the `lookup` method from the initial context. The `comp/env/service/MyHelloServiceRef` in Example 14–5 provides the service reference. The JNDI call returns a reference to a `service` object.

   ```
   Service service = (Service)
   ic.lookup("java:comp/env/service/MyHelloServiceRef");
   ```

The client always accesses the service implementation by using a JNDI lookup. This lookup returns a container-managed service reference. This allows the container to intervene and provide the client with additional service functionality, such as logging, security, and management.

3. Get a handle to the service port using the `getPort` method on the container-managed service object. Cast the return value to the interface type.

```
HelloInterface helloPort = (HelloInterface) service.getPort(portQName,
oracle.demo.hello.HelloInterface.class);
```

Note that this step assumes that a `QName` has already been defined. For example:

```
QName portQName = new QName("http://hello.demo.oracle/", "HelloInterfacePort");
```

Instead of `getPort`, the client can make a DII Web service call by using the `service` object to get a handle on the `Call` object.

```
Call call = service.createCall(new QName("http://hello.demo.oracle/",
"HelloInterfacePort");
```

4. Call a method on the remote object.

```
resultFromService = helloPort.sayHello(name);
```

Example 14–5 illustrates code that a servlet or JSP Web service client can use to look up a Web service.

Example 14–5 Servlet or JSP Code to Look Up a Web Service

```
public String consumeService (String name)
{
   ......
   Context ic = new InitialContext();
   Service service =
(Service)ic.lookup("java:comp/env/service/MyHelloServiceRef");
   // declare the qualified name of the port, as specified in the wsdl
   QName portQName= new QName("http://hello.demo.oracle/","HelloInterfacePort");
   //get a handle on that port : Service.getPort(portQName,SEI class)
   HelloInterface helloPort =
(HelloInterface)
service.getPort(portQName,oracle.demo.hello.HelloInterface.class);
   //invoke the operation : sayHello()
   resultFromService = helloPort.sayHello(name);
   .....
}
```

How to Configure a J2EE Web Service Client for a Stateful Web Service

J2EE Web service clients can be configured, either by using configuration files or programmatically, to consume stateful Web services.

- How to Configure a J2EE Client with Deployment Descriptor Files
- How to Configure a J2EE Client Programmatically

See Also:

"Exposing Java Classes as a Stateful Web Service" on page 7-9 for more information on stateful Web services.

How to Configure a J2EE Client with Deployment Descriptor Files

A J2EE client can be configured to consume stateful Web services by editing the `<service-ref-mapping>` clause of the appropriate Oracle proprietary deployment descriptor (either `orion-web.xml`, `orion-ejb-jar.xml`, or `orion-application-client.xml`).

Within the `<service-ref-mapping>` clause, add a `<stub-property>` element with its `<name>` subelement set to the `javax.xml.rpc.session.maintain` property and its `<value>` subelement set to `true`.

The value of the J2EE standard property `javax.xml.rpc.session.maintain` indicates to the client whether it wants to participate in a session with a service endpoint. If this property is set to `true`, the client indicates that it wants the session to be maintained.

Example 14–6 illustrates a Web service client configuration for a stateful Web service. The definition of the `<stub-property>` allows the client to participate in a session with the port identified by the `CycleCounterInterface` service endpoint. The `<stub-property>` element, with its setting for `javax.xml.rpc.session.maintain`, is highlighted in bold.

Example 14–6 Configuration for a Client Participating with a Stateful Web Service

```
     <service-ref-mapping name="service/CycleCounter">
         <port-info>

<service-endpoint-interface>test.oracle.stateful.CycleCounterInterface</service-en
dpoint-interface>
          <!-- set the javax.xml.rpc.session.maintain property to true for a
stateful client -->
                <stub-property>
                     <name>javax.xml.rpc.session.maintain</name>
                     <value>true</value>
                </stub-property>
                <stub-property>
                     <name>javax.xml.rpc.service.endpoint.address</name>
                     <value>http://%J2EE_HOST%:%HTTP_
PORT%/testsfWS-session/testsfWS-session</value>
                 </stub-property>
          </port-info>
     </service-ref-mapping>
```

How to Configure a J2EE Client Programmatically

A J2EE client can be configured programmatically to consume stateful Web services. To do this, ensure that the client participates in the session by setting the `SESSION_MAINTAIN_PROPERTY` runtime property (`javax.xml.rpc.session.maintain`) to `true` either on the stub, the DII call, or the endpoint client instance.

For example, you can set the value of this property inside the generated implementation `_port` of `javax.xml.rpc.Stub`:

```
((Stub)__port)._setProperty(Stub.SESSION_MAINTAIN_PROPERTY, Boolean.valueOf
(maintainSession));
```

Instead of setting this property directly, OracleAS Web Services provides a helpful wrapper class with a `setMaintainSession(boolean)` method. When this method is set to `true` the session is maintained. The wrapper takes care of setting the property inside of the client. For example, in the client code, you can enter the following:

```
HttpSoap11Client c = new HttpSoap11Client(); // client wrapper class
```

```
c.setMaintainSession(true);
```

How to Configure a J2EE Web Service Client to Make JMS Transport Calls

You can statically configure the J2EE client to make JMS transport calls. To do this, add a `<service-ref-mapping>` clause to the appropriate Oracle proprietary J2EE client deployment descriptor file for your Web service (`orion-web.xml`, `orion-ejb-jar.xml`, or `orion-application-client.xml`). Within the clause, configure a `<stub-property>` element with `name` and `value` attributes for each of these items.

- ReplyTo queue—Enter a `<stub-property>` element with the `name` subelement set to the `ReplyToQueueName` API (`oracle.webservices.transport.ReplyToQueueName`) and the `value` subelement set to the JNDI name of the ReplyTo queue.
- ReplyTo factory name—Enter a `<stub-property>` element with the `name` attribute set to the `ReplyToFactoryName` API (`oracle.webservices.transport.ReplyToFactoryName`) and the `value` subelement set to the JNDI name of the ReplyTo factory.
- service endpoint address—Enter a `<stub-property>` element with `name` subelement set to the service endpoint address API (`javax.xml.rpc.service.endpoint.address`) and the `value` subelement set to the service endpoint interface file.

Example 14–7 illustrates a sample configuration.

Example 14–7 J2EE Client Configuration for JMS Transport Calls

```
<service-ref-mapping name="service/MyJMSService">
     <stub-property>
        <name>oracle.webservices.transport.ReplyToQueueName</name>
        <value>jms/receiverQueue</value>
     </stub-property>
     <stub-property>
        <name>oracle.webservices.transport.ReplyToFactoryName</name>
        <value>jms/receiverQueueConnectionFactory</value>
     </stub-property>
     <stub-property>
        <name>javax.xml.rpc.service.endpoint.address</name>
        <value>/bank/soap12bank</value>
     </stub-property>
</service-ref-mapping>
```

How to Enable Chunked Data Transfer for HTTP 1.1

OracleAS Web Services permits the chunked transfer encoding of messages when the protocol is HTTP 1.1. Chunked data transfer can be invoked on J2SE stub, J2EE stub and DII Web service clients.

Chunking can increase performance by breaking the payload into smaller pieces. These pieces can be sent over the wire faster than one large payload. Chunked transfer encoding includes all of the information that the recipient needs to verify that it has received the entire message. Chunked transfer encoding happens at the transport level; it is not detected or handled by the invoker of a Web services call or the server.

The following properties in the `oracle.webservices.ClientConstants` class can be set on the `Stub` or `Call` object to enable chunking and set the chunk size.

- DO_NOT_CHUNK—if this property is not set, or set to `true`, then chunking is turned off by default. If this property is set to `false`, then chunking is enabled.
- CHUNK_SIZE—sets the chunk size in bytes. If this property is not set, then the default chunk size is 4096 bytes.

Example 14–8 illustrates setting the chunking and chunk size property in client proxy stub code.

Example 14–8 Stub Code to Set Data Chunk Size

```
import oracle.webservices.ClientConstants;
...
((OracleStub)port)._setProperty(ClientConstants.DO_NOT_CHUNK, true);
((OracleStub)port)._setProperty(ClientConstants.CHUNK_SIZE, 1024);
...
```

Example 14–9 illustrates using the DO_NOT_CHUNK and CHUNK_SIZE properties in DII client code to set the chunk size to 1024 bytes.

Example 14–9 DII Client Code to Set Data Chunk Size

```
import oracle.webservices.ClientConstants;
...
ServiceFactory factory = ServiceFactory.newInstance();
Service service = factory.createService(new
QName("http://whitemesa.net/wsdl/rpc-lit-test", "tns" ) );
QName stringType = new QName( "http://www.w3.org/2001/XMLSchema", "string");
Call call = service.createCall();
...
call.setProperty(ClientConstants.DO_NOT_CHUNK, false);
call.setProperty(ClientConstants.CHUNK_SIZE, 1024);
...
```

How to Set a Character Encoding for a SOAP Message

By default, a J2EE client (either static stub or DII) assembled under OracleAS Web Services sends a request message with a SOAP envelope encoded with UTF-8 characters. To override this behavior, you can set the following Oracle proprietary property:

```
oracle.webservices.ClientConstants.CHARACTER_SET_ENCODING
```

You can apply this property to the `javax.xml.rpc.Stub` or `javax.xml.rpc.Call` object with the `setProperty` method.

The value of the CHARACTER_SET_ENCODING property can be of type `java.lang.String` or `java.nio.charset.Charset`. The set of supported character encodings depends on the underlying Java Virtual Machine (JVM). Use the `Charset.availableCharsets` method to return the list of character encodings that are supported by your JVM. for more information on the `Charset.availableCharsets` method, see the output of the Javadoc tool for the `java.nio.charset.Charset` class.

This property can also be used for J2SE Web service clients.

Example 14–10 illustrates Stub client code that sets `Shift_JIS` as the character encoding that will be used by the SOAP envelope.

Example 14–10 Setting Shift_JIS Characters for a SOAP Envelope on a Stub Client

```
import oracle.webservices.ClientConstants;
...
((OracleStub)port)._setProperty(ClientConstants.CHARACTER_SET_ENCODING, "Shift_
JIS");
...
```

Example 14–11 illustrates DII client code that sets `Shift_JIS` as the character encoding that will be used by the SOAP envelope.

Example 14–11 Setting Shift_JIS Characters for a SOAP Envelope on a DII Client

```
import oracle.webservices.ClientConstants;
...
ServiceFactory factory = ServiceFactory.newInstance();
Service service = factory.createService(new URL("path to wsdl"),
new QName("service namespace", "service name" ) );
Call call = service.createCall();
call.setProperty(ClientConstants.CHARACTER_SET_ENCODING, Charset.forName("Shift_
JIS"));
...
```

Understanding the Packaging Structure for J2EE Clients

Oracle JDeveloper creates a standard package structure for Web application and EJB client files. This section describes that structure in case you need to customize the contents of the client EAR file.

- Understanding the Packaging Structure for a Servlet or Web Application Client
- Understanding the Packaging Structure for an EJB Client

Understanding the Packaging Structure for a Servlet or Web Application Client

This section describes the packaging of servlet or Web application clients. The values for a number of elements in the deployment descriptors reflect the names of the files and their storage position in the EAR file. If you change the content of the EAR file, you might need to change the content of the deployment descriptors.

- Packaging Structure for Servlet or Web Application Clients
- Relationship Between Deployment Descriptors for Servlet or Web Application Clients

Packaging Structure for Servlet or Web Application Clients

Servlet or Web application clients are packaged in an EAR file with the name `<ear_file_name>.ear`. At the top level, the EAR file contains a `META-INF` directory for the manifest file and the `application.xml` file and `<war_file_name>.war` file for the servlet or Web application files, the JAX-RPC mapping file, the WSDL file, and the deployment descriptors. Example 14–12 illustrates the standard package structure of the EAR file.

Example 14–12 Structure of a Servlet or Web Application Client EAR File

```
./META-INF
    ./MANIFEST.MF
    ./application.xml
./<war file>.war
    ./WEB-INF/
```

```
            /orion-web.xml
            /web.xml
            /wsdl/<wsdl file name>.wsdl
            /<mapping file>.xml
            /classes
                 /class files
            /lib
                 /.jar files
    ./*.jsp or html files
```

Relationship Between Deployment Descriptors for Servlet or Web Application Clients

This section identifies the relationships between the J2EE standard deployment descriptor `web.xml`, the OC4J deployment descriptor for servlets or Web applications `orion-web.xml`, and the packaging structure of the client EAR file. These relationships are important because if you edit the structure or contents of the client EAR file, you might have to edit the content of the deployment descriptors.

The client information is contained in the `<service-ref>` element in `web.xml`. This element contains information about the Web service that can be looked up and consumed from inside a servlet or JSP. For example, it contains the locations for the WSDL (`<wsdl-file>`), the JAX-RPC mapping file (`<jaxrpc-mapping-file>`), the service interface used for JNDI lookup (`<service-ref-name>`), the service interface class (`<service-interface>`), and the service endpoint interface (`<service-endpoint-interface>`). Note that the `<service-ref-name>` in `web.xml` also appears as an attribute in the `<service-ref-mapping>` element in `orion-web.xml`. If you change the names and locations of any of these items in the EAR, then you must make the corresponding changes in the deployment descriptors.

> **See Also:**
>
> "How to Add J2EE Web Service Client Information to Deployment Descriptors" on page 14-4 for information on the `<service-ref>` and `<service-ref-mapping>` elements and their subelements.

Example 14–13 lists the contents of `web.xml` for a servlet or Web application client. The `<service-ref>` element is highlighted in bold.

Example 14–13 web.xml Contents for a Servlet or Web Application Client

```
<web-app>
  <servlet>
    <servlet-name>consumer</servlet-name>
    <servlet-class>oracle.ServiceConsumerServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>consumer</servlet-name>
    <url-pattern>/consumer</url-pattern>
  </servlet-mapping>
  <service-ref>
    <service-ref-name>service/MyHelloServiceRef</service-ref-name>
    <service-interface>javax.xml.rpc.Service</service-interface>
    <wsdl-file>WEB-INF/wsdl/HelloService.wsdl</wsdl-file>

<jaxrpc-mapping-file>WEB-INF/HelloService-java-wsdl-mapping.xml</jaxrpc-mapping-fi
le>
    <service-qname
xmlns:service-qname_ns__="http://hello.demo.oracle/">service-qname_ns__
```

```
:HelloService</service-qname>
        <port-component-ref>

<service-endpoint-interface>oracle.demo.hello.HelloInterface</service-endpoint-int
erface>
        </port-component-ref>
  </service-ref>
</web-app>
```

Example 14–14 lists the contents of OC4J proprietary `orion-web.xml` deployment descriptor for Web applications and servlets. The `<service-ref-mapping>` element is highlighted in bold.

Example 14–14 orion-web.xml Contents for a Client-Side Servlet or Web Application

```
<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/MyHelloServiceRef">
                        <!-- stub property applicable across all ports -->
                        <stub-property>

<name>javax.xml.rpc.service.endpoint.address</name>

<value>http://localhost:8888/hello/HelloService</value>
                        </stub-property>
        </service-ref-mapping>
</orion-web-app>
```

Understanding the Packaging Structure for an EJB Client

This section describes the packaging of EJB clients. The values for a number of elements in the deployment descriptors reflect the names of the files and their storage position in the EAR file. If you change the content of the EAR file, you might need to change the content of the deployment descriptors.

- Package Structure for EJB Application Clients
- Relationship Between Deployment Descriptors for EJB Application Clients

Package Structure for EJB Application Clients

EJB clients are packaged in an EAR file with the name `<ear_file_name>.ear`. At the top level, the EAR file contains a `META-INF` directory for the manifest file and the `application.xml` file, the EJB class files, and `<ejb_jar_file_name>.jar` file. The JAR file contains the JAR manifest file, the JAX-RPC mapping file, the WSDL file, and the deployment descriptors. Example 14–15 illustrates the packaging structure of an EJB client EAR file.

Example 14–15 Package Structure for a Client-Side EJB Application EAR File

```
./META-INF
     ./MANIFEST.MF
     ./application.xml
./<ejb jar file name>.jar
     ./class files
     ./META-INF/
          /MANIFEST.MF
```

```
/ejb-jar.xml
/orion-ejb-jar.xml
/wsdl/<wsdl file name>.wsdl
/<mapping file>.xml
```

Relationship Between Deployment Descriptors for EJB Application Clients

This section identifies the relationships between the J2EE standard deployment descriptor `ejb-jar.xml`, the OC4J deployment descriptor for servlets or Web applications `orion-ejb-jar.xml`, and the packaging structure of the EJB client EAR file. These relationships are important because if you edit the structure or contents of the client EAR file, you might have to edit the content of the deployment descriptors.

The client information is contained in the `<service-ref>` element in `ejb-jar.xml`. This element contains information about the servlet or Web application that can be used as a Web service client. For example, it contains the locations for the WSDL (`<wsdl-file>`), the JAX-RPC mapping file (`<jaxrpc-mapping-file>`), the service interface used for JNDI lookup (`<service-ref-name>`), the service interface class (`<service-interface>`), and the service endpoint interface (`<service-endpoint-interface>`). Note that the `<service-ref-name>` in `ejb-jar.xml` also appears as an attribute in the `<service-ref-mapping>` element in `orion-ejb-jar.xml`. If you change the names and locations of any of these items in the EAR, then you must make the corresponding changes in the deployment descriptors.

See Also:

"How to Add J2EE Web Service Client Information to Deployment Descriptors" on page 14-4 for information on the `<service-ref>` and `<service-ref-mapping>` elements and their subelements.

Example 14–16 lists the contents of `ejb-jar.xml` for an EJB client. The `<service-ref>` element is highlighted in bold.

Example 14–16 ejb-jar.xml Contents for a Client-Side EJB Application

```
<ejb-jar>
  <display-name>serviceConsumerEJB</display-name>
  <enterprise-beans>
    <session>
        <ejb-name>ServiceConsumer</ejb-name>
        <home>oracle.ServiceConsumerHome</home>
        <remote>oracle.ServiceConsumerRemote</remote>
        <ejb-class>oracle.ServiceConsumerBean</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Container</transaction-type>
        <service-ref>
                <service-ref-name>service/MyHelloService</service-ref-name>
                <service-interface>javax.xml.rpc.Service</service-interface>
                <wsdl-file>META-INF/wsdl/HelloService.wsdl</wsdl-file>

<jaxrpc-mapping-file>META-INF/HelloService-java-wsdl-mapping.xml</jaxrpc-mapping-f
ile>

<service-qnamexmlns:ns="http://hello.demo.oracle/">ns:HelloService</service-qname>
                <port-component-ref>

<service-endpoint-interface>oracle.demo.hello.HelloInterface</service-endpoint-int
erface>
```

```
                </port-component-ref>
            </service-ref>
        </session>
    </enterprise-beans>
</ejb-jar>
```

Example 14–17 lists the contents of OC4J proprietary `orion-ejb-jar.xml` deployment descriptor for Web applications and servlets. The `<service-ref-mapping>` element is highlighted in bold.

Example 14–17 orion-ejb-jar.xml Contents for a Client-Side EJB Application

```
<orion-ejb-jar>
    <enterprise-beans>
        <session-deployment name="ServiceConsumer">
            <service-ref-mapping name="service/MyHelloService">
                <stub-property>
                    <name>javax.xml.rpc.service.endpoint.address</name>
                    <value>http://localhost:8888/hello/HelloService</value>
                </stub-property>
            </service-ref-mapping>
        </session-deployment>
    </enterprise-beans>
</orion-ejb-jar>
```

Limitations

See "Assembling a J2EE Web Service Client" on page F-9.

Additional Information

For more information on:

- assembling Web services from a WSDL, see Chapter 6, "Assembling a Web Service from WSDL".
- assembling stateful Web services, see Chapter 7, "Assembling a Web Service with Java Classes".
- assembling Web services from EJBs, see Chapter 8, "Assembling a Web Service with EJBs".
- assembling Web services from a JMS queue or topic, see Chapter 9, "Assembling Web Services with JMS Destinations".
- assembling Web services from database resources, see Chapter 10, "Assembling Database Web Services".
- assembling Web services with J2SE 5.0 Annotations, see Chapter 11, "Assembling Web Services with Annotations".
- building J2SE clients, see Chapter 15, "Assembling a J2SE Web Service Client".
- using the WebServicesAssembler tool to assemble Web services, see Chapter 18, "Using WebServicesAssembler".
- packaging and deploying Web services, see Chapter 19, "Packaging and Deploying Web Services"
- jar files that are needed to assemble a client, see Appendix A, "Web Service Client APIs and JARs".

- Web services interoperability, see "Ensuring Interoperable Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- using quality of service features in Web service clients, see "Managing Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- adding security to a Web service, see the *Oracle Application Server Web Services Security Guide*.
- how to write clients to access Web services secured on the transport level, see "Adding Transport-level Security for Web Services Based on EJBs" and "Accessing Web Services Secured on the Transport Level" in the *Oracle Application Server Web Services Security Guide*.
- adding reliability to a Web service, see "Ensuring Web Service Reliability" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- adding an auditing and logging configuration to a Web service, see "Auditing and Logging Messages" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- processing nonstandard data types, see "Custom Serialization of Java Value Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- the JAX-RPC mapping file, see "JAX-RPC Mapping File Descriptor" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- data types supported by OracleAS Web Services, see "Mapping Java Types to XML and WSDL Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- Oracle JDeveloper tool support for Web service development, see the Oracle JDeveloper on-line help.

15

Assembling a J2SE Web Service Client

This chapter provides information on developing a Web services client for the J2SE platform. This chapter has the following sections.

- Understanding J2SE Web Service Clients
- How to Assemble a J2SE Web Service Client
- Writing Web Service Client Applications
- How to Use Dynamic Invocation Interface to Invoke Web Services
- Tool Support for Assembling J2SE Web Service Clients

Understanding J2SE Web Service Clients

The J2SE client, unlike the J2EE client, is responsible for much of the underlying work of looking up the service, and creating and maintaining the instances of classes that access the Web service. Since developers cannot rely on the container, they must create and manage their own services, and ensure the availability of all runtime environments needed to access the Web service.

The following sections describe static stub clients and Dynamic Invocation Interface (DII) clients.

- Understanding Static Stub Clients
- Understanding JAX-RPC Dynamic Invocation Interface

Understanding Static Stub Clients

The WebServicesAssembler command `genProxy` generates static stubs from a supplied WSDL document. The generated stubs implement the `javax.xml.rpc.Stub` interface and a service endpoint interface. Additionally, the generated stubs are specifically bound to the HTTP transport and SOAP protocol. You can instantiate the generated stubs and invoke their methods directly to send requests to the associated Web service.

In addition, WebServicesAssembler generates a client utility class that demonstrates how a client leverages the static stubs to interact with a Web service. The name of the utility client class is *<WSDL_port_name>*`Client.java`. This class handles all steps necessary for creating a stub instance. You may want to instantiate the utility client and use it to invoke the remote service's operations.

Note: The client utility class file is regenerated every time WebServicesAssembler is executed. It is strongly recommended that you place your own code in a separate file; otherwise, you will lose your changes.

Understanding JAX-RPC Dynamic Invocation Interface

The JAX-RPC Dynamic Invocation Interface enables you to invoke a remote Web service operation even if the name of the service or the signature of the remote method is unknown prior to runtime.

Support for DII is provided through OC4J's implementation of the `javax.xml.rpc.Call` interface. The `javax.xml.rpc.Service` class acts as a factory for `Call` instances by using the overloaded `Service.createCall()` method. Once created, the various getters and setters that the `Call` interface provides are used to configure the port type, the operation name, the service endpoint address, and other attributes required for executing the remote method.

See Also:

"How to Use Dynamic Invocation Interface to Invoke Web Services" on page 15-9 for examples of using DII clients to invoke Web services.

How to Assemble a J2SE Web Service Client

This section contains the following subsections:

- Prerequisites
- Steps to Assemble a J2SE Web Service Client with a Static Stub

Prerequisites

Before you begin, provide the following files and information.

- Supply the URI to the WSDL you want to employ to generate the client. This chapter uses the WSDL file described in "Sample WSDL File" on page 15-3.
- Decide on the destination location for generated artifacts.
- Decide on a package name for the client files.

Steps to Assemble a J2SE Web Service Client with a Static Stub

You can use WebServicesAssembler to create a J2SE Web service client using static stubs. To create the static stub, follow these steps.

1. Provide the URI to the WSDL, the name of the output directory, the package name, and the other information and files described in the Prerequisites section as input to the WebServicesAssembler `genProxy` command. The following command generates the client proxies and stores them in `build/src/client`. The client application uses the stub to invoke operations on a remote service:

   Command line:

   ```
   java -jar wsa.jar -genProxy
                     -output build/src/client/
                     -wsdl http://localhost:8888/hello/HelloService?WSDL
                     -packageName oracle.demo.hello
   ```

Ant task:

```
<oracle:genProxy
            wsdl="http://localhost:8888/hello/HelloService?WSDL"
            output="build/src/client"
            packageName="oracle.demo.hello"/>
```

In this command and Ant task:

- `genProxy`—Creates a static proxy stub that can be used by a J2SE Web service client. See "genProxy" on page 18-34.
- `wsdl`—Specifies the absolute file path, relative file path, or URL to a WSDL document. See "wsdl" on page 18-65.
- `output`—Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See "output" on page 18-48.
- `packageName`—Specifies the package name that will be used for generated classes if no package name is declared in the JAX-RPC mapping file. See "packageName" on page 18-48.

For more information on the required and optional arguments to `genProxy`, see "genProxy" on page 18-34.

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`.

   You can also use the client utility class file to test your endpoint. Example 15–2, "HelloInterfacePortClient.java Listing" on page 15-6, illustrates the client utility class file created in this example. For more information about this file, see "Writing Web Service Client Applications" on page 15-5.

3. Compile the client files and put them in the classpath.

   List the appropriate JARs on the classpath before compiling the client. Table A–2, " Classpath Components for a Client Using a Client-Side Proxy" 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 Table A–2. See "Setting the Classpath for a Web Service Proxy" on page A-2 for more information on `wsclient_extended.jar` and the client classpath.

4. Run the J2SE client from the command line.

Sample WSDL File

Example 15–1 contains a partial listing of the `HelloService.wsdl` used to generate the client. One of the files generated from this WSDL is the client utility class file listed in Example 15–2.

This partial listing illustrates a number of entries in the WSDL file that are employed in the generation of the client utility class file. For example:

- The name of the client utility class file, `HelloInterfacePortClient.java`, is derived from the value of the `<port name>` attribute.
- The operation name, `sayHello`, which appears under the `<portType>` element, becomes a method in the client utility class file.
- The parameter and data type belonging to `sayHello` is defined by the `complexType` defining the `sayHello` request:

```
    <complexType name="sayHello">
        <sequence>
              <element name="name" nillable="true" type="string"/>
        </sequence>
    </complexType>
```

The preceding elements appear in bold in the partial listing of `HelloService.wsdl`.

Example 15–1 WSDL Fragment, With Elements Used in the Client Utility Class File

```
<definitions
    ...
  >
    <types>
        <schema targetNamespace="http://hello.demo.oracle/" ...
xmlns="http://www.w3.org/2001/XMLSchema"
           ...
            <complexType name="sayHello">
                <sequence>
                    <element name="name" nillable="true" type="string"/>
                </sequence>
            </complexType>
            <complexType name="sayHelloResponse">
                <sequence>
                    <element name="result" nillable="true" type="string"/>
                </sequence>
            </complexType>
            <element name="sayHelloElement" type="tns:sayHello"/>
            <element name="sayHelloResponseElement" type="tns:sayHelloResponse"/>
        </schema>
    </types>
    <message name="HelloInterface_sayHelloResponse">
        <part name="parameters" element="tns:sayHelloResponseElement"/>
    </message>
    <message name="HelloInterface_sayHello">
        <part name="parameters" element="tns:sayHelloElement"/>
    </message>
    <portType name="HelloInterface">
        <operation name="sayHello">
            <input message="tns:HelloInterface_sayHello"/>
            <output message="tns:HelloInterface_sayHelloResponse"/>
        </operation>
    </portType>
    <binding name="HelloInterfacePortBinding" type="tns:HelloInterface">
        <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="sayHello">
            <soap:operation soapAction="http://hello.demo.oracle/:sayHello"/>
            <input>
                <soap:body use="literal" parts="parameters"/>
            </input>
            <output>
                <soap:body use="literal" parts="parameters"/>
            </output>
        </operation>
    </binding>
    <service name="HelloService">
        <port name="HelloInterfacePort" binding="tns:HelloInterfacePortBinding">
            <soap:address location="HelloService"/>
        </port>
```

```
    </service>
</definitions>
```

Writing Web Service Client Applications

This section contains the following subsections:

- Understanding the Client Utility Class File
- Steps to Create a Service Factory
- How to Enable Chunked Data Transfer for HTTP 1.1
- How to Set a Character Encoding for a SOAP Message on a J2SE Client
- How to Set Cookies in a Web Service Client Stub

Understanding the Client Utility Class File

The `genProxy` command generates a client utility class file that enables you to invoke Web service methods. You can use this file as your application client, or use it as a template to write your own application client code.

> **Note:** The client utility class file is regenerated each time you run `genProxy`. If you add code to this file for testing purposes, then your changes will be lost if you regenerate the proxy. For production code, your client code should exist outside of this utility class.

The command derives the name of the file by appending the suffix `Client` to the port name. For example, for the `HelloInterfacePort` port name, `genProxy` generates the `HelloInterfacePortClient.java` file.

> **See Also:**
>
> Example 15–2, "HelloInterfacePortClient.java Listing" on page 15-6 for an example of a client utility class file generated by the `genProxy` command.

The client utility class serves as a proxy to the Web service implementation. The client-side proxy code constructs a SOAP request and marshals and unmarshals parameters for you. Using the proxy classes saves you the work of creating SOAP requests and data marshalling for accessing a Web service or processing Web service responses.

The most important part of the client utility class file is the factory code for creating `javax.xml.rpc.Service` objects and obtaining the operations available on the service. The `Service` object acts as an instance of the generated stub class.

Note the following lines of code in the client utility class file:

```
public HelloInterfaceClient() throws Exception {
    ServiceFactory factory = ServiceFactory.newInstance();
    _port = ((HelloService)factory.loadService
                (HelloService.class)).getHelloInterfacePort();
}
```

If you write your own application client, then you *must* supply this code to create the service factory and obtain a port implementation.

Steps to Create a Service Factory

The following steps describe the code.

1. Instantiate a new `javax.xml.rpc.ServiceFactory` instance or use an existing instance.

   ```
   ServiceFactory factory = ServiceFactory.newInstance();
   ```

2. Load the service for a particular service endpoint interface using the `loadService` method. This returns an object of type `Service` that also implements the requested service endpoint interface.

   ```
   (HelloService)factory.loadService(HelloService.class)
   ```

   In this example, the returned `Service` is cast to the service endpoint interface `HelloService`.

3. Use the `get...()` method to get the desired port. The ellipsis ("...") represents the value of the `port name` element in the WSDL.

   ```
   HelloService.getHelloInterfacePort();
   ```

   In this example, the method name is `getHelloInterfacePort()`, where `HelloInterfacePort` is the `port name` in the WSDL. The method returns a Java implementation for `HelloInterfacePort`.

Example 15–2 displays the client utility class file, `HelloInterfacePortClient.java`, which was generated from the `HelloService.wsdl` in Example 15–1 by the `genProxy` command.

The lines of code that create the `Service` objects and obtain the operations available on the service appear in bold. Note the generated code that can be used to set the `SESSION_MAINTAIN_PROPERTY` system property to alert the client that the Web service is stateful. The session will be maintained when the client is operating in conjunction with a server side, stateful Web service. By maintaining the session, subsequent requests to the service are faster.

Example 15–2 HelloInterfacePortClient.java Listing

```
import oracle.webservices.transport.ClientTransport;
import oracle.webservices.OracleStub;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.Stub;

public class HelloInterfacePortClient {
    private HelloInterface _port;

    public HelloInterfacePortClient() throws Exception {
        ServiceFactory factory = ServiceFactory.newInstance();
        _port =((HelloService)factory.loadService(
           HelloService.class)).getHelloInterfacePort();
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        try {
            HelloInterfacePortClient myPort = new HelloInterfacePortClient();
            System.out.println("calling " + myPort.getEndpoint());
            // Add your own code here
```

```
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    /**
     * delegate all operations to the underlying implementation class.
     */
    // sayHello
    public String sayHello(String name) throws java.rmi.RemoteException {
        return _port.sayHello(name);
    }

    /**
     * used to access the JAX-RPC level APIs
     * returns the interface of the port instance
     */
    public oracle.demo.hello.HelloInterface getPort() {
        return _port;
    }

    public String getEndpoint() {
        return (String) ((Stub)
_port)._getProperty(Stub.ENDPOINT_ADDRESS_PROPERTY);
    }

    public void setEndpoint(String endpoint) {
        ((Stub) _port)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY,
endpoint);
    }

    public String getPassword() {
        return (String) ((Stub) _port)._getProperty(Stub.PASSWORD_PROPERTY);
    }

    public void setPassword(String password) {
        ((Stub) _port)._setProperty(Stub.PASSWORD_PROPERTY, password);
    }

    public String getUsername() {
        return (String) ((Stub) _port)._getProperty(Stub.USERNAME_PROPERTY);
    }

    public void setUsername(String username) {
        ((Stub) _port)._setProperty(Stub.USERNAME_PROPERTY, username);
    }

    public void setMaintainSession(boolean maintainSession) {
        ((Stub) _port)._setProperty(Stub.SESSION_MAINTAIN_PROPERTY, new
Boolean(maintainSession));
    }

    public boolean getMaintainSession() {
        return ((Boolean) ((Stub)
_port)._getProperty(Stub.SESSION_MAINTAIN_PROPERTY)).booleanValue();
    }

   /**
     * returns the transport context
```

```
     */
    public ClientTransport getClientTransport() {
        return ((OracleStub) _port).getClientTransport();
    }
}
```

How to Enable Chunked Data Transfer for HTTP 1.1

OracleAS Web Services permits the chunked transfer encoding of messages when the protocol is HTTP 1.1. Chunked data transfer can be invoked on J2SE stub, J2EE stub and DII Web service clients.

See Also:

"How to Enable Chunked Data Transfer for HTTP 1.1" on page 14-15 for more information on how to enable this feature.

How to Set a Character Encoding for a SOAP Message on a J2SE Client

By default, a client assembled under Oracle Application Server Web Services sends a request message with a SOAP envelope encoded with UTF-8 characters. To override this behavior, you can set the following Oracle proprietary property:

`oracle.webservices.ClientConstants.CHARACTER_SET_ENCODING`

Use this property to set the character encoding for a J2SE client or a J2EE client.

See Also:

"How to Set a Character Encoding for a SOAP Message" on page 14-16 for more information on how to use the `CHARACTER_SET_ENCODING` property.

How to Set Cookies in a Web Service Client Stub

A client stub can be used to set the cookies used in an HTTP request. The `Cookie` class represents an HTTP cookie and can be found in the Oracle Applications Server 10*g* `HTTPClient` package. `Cookie` has the following constructor:

`Cookie (java.lang.String` *`name`*`, java.lang.String` *`value`*`, java.lang.String` *`domain`*`, java.lang.String` *`path`*`, java.util.Date` *`expires`*`, boolean` *`secure`*`)`

All parameters except *`expires`* are required to the `Cookie` constructor.

The domain that you specify for the cookie must be either:

- the full host name, or
- in the case where the cookie is shared between multiple hosts, the domain must start with a period. For example:

  `.oracle.com`

The following general steps describe how to set a cookie in an HTTP request:

1. Create a `java.util.Map` object to contain the cookies.
2. Construct one or more cookies, using the `Cookie` class from the Oracle Applications Server 10*g* `HTTPClient` package.
3. Load the cookies into the `Map` object.
4. Set the property `oracle.webservices.ClientConstants.COOKIE_MAP`.

The value of the property is a `java.util.Map` object that contains items and keys of type `HTTPClient.Cookie`.

5. Set the `javax.xml.rpc.session.SESSION_MAINTAIN_PROPERTY` runtime property set to `true`.

   This property alerts the client that the Web service is stateful. If this property is not set to `true`, then the cookies will be ignored.

Example 15–3 illustrates stub code that sets the value of two cookies. The `cookieMap` variable is declared to be of type `java.util.Map` and obtains its contents from a `HashMap`. The `Cookie` constructor is used to define two cookies, `cookie` and `cookie2`. The `cookieMap.put` lines add the cookies to the hashmap. The `Stub.SESSION_MAINTAIN_PROPERTY` is present and set to `true` and the `ClientConstants.COOKIE_MAP` is set to `cookieMap`.

Example 15–3 Setting a Cookie in a Client Stub

```
import HTTPClient.Cookie;
   Map cookieMap = new HashMap();

   Cookie cookie = new Cookie("name", "value", ".oracle.com", "/",  null, false);
   Cookie cookie2 = new Cookie("name2", "value2", ".oracle.com", "/", null,
false);
   cookieMap.put(cookie, cookie);
   cookieMap.put(cookie2, cookie2);

   ((Stub) port)._setProperty(ClientConstants.COOKIE_MAP, cookieMap);
   ((Stub) port)._setProperty(Stub.SESSION_MAINTAIN_PROPERTY, Boolean.TRUE);
...
```

How to Use Dynamic Invocation Interface to Invoke Web Services

Invoking 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 15–4, "Basic Call with parameter registration and Java bindings"
   - Example 15–7, "Basic Call with SOAPElement, but without parameter registration"

- Example 15–9, "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 15–7, "Basic Call with SOAPElement, but without parameter registration"
     - Example 15–9, "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 15–4, "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 15–7, "Basic Call with SOAPElement, but without parameter registration"
     - Example 15–9, "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 15–4, "Basic Call with parameter registration and Java bindings"
     - Example 15–5, "Configured Call with Java bindings, but without parameter registration"
     - Example 15–6, "Configured Call with registration of wrapper parameters and Java bindings"
     - Example 15–8, "Configured Call with a WSDL, complex return parameter registration, and Java bindings"
     - Example 15–10, "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 15–5, "Configured Call with Java bindings, but without parameter registration"
- Example 15–6, "Configured Call with registration of wrapper parameters and Java bindings"
- Example 15–8, "Configured Call with a WSDL, complex return parameter registration, and Java bindings"
- Example 15–10, "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 15–8, "Configured Call with a WSDL, complex return parameter registration, and Java bindings"
     - Example 15–10, "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 15–7 and Example 15–9 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 15–6, "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 15–5, "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 15–7, "Basic Call with SOAPElement, but without parameter registration"
     - Example 15–9, "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 15–4, "Basic Call with parameter registration and Java bindings"
     - Example 15–5, "Configured Call with Java bindings, but without parameter registration"
     - Example 15–6, "Configured Call with registration of wrapper parameters and Java bindings"

- Example 15–8, "Configured Call with a WSDL, complex return parameter registration, and Java bindings"
- Example 15–10, "Configured Call with RPC-encoded invocation, complex parameter registration, and Java bindings"

Examples of Web Service Clients that use Dll

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 15–4 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 15–5 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 15–6 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 15–7 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 15–8 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 15–9 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 15–10 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();
```

Tool Support for Assembling J2SE Web Service Clients

Oracle JDeveloper enables you to build client applications that use Web services. It supports OC4J J2SE Web service clients by allowing you to create Java stubs from Web service WSDL descriptions. You can use these stubs to access existing Web services. For more information, see the Oracle JDeveloper on-line help.

Additional Information

For more information on:

- assembling Web services from a WSDL, see Chapter 6, "Assembling a Web Service from WSDL".

- assembling stateful Web services, see Chapter 7, "Assembling a Web Service with Java Classes".
- assembling Web services from EJBs, see Chapter 8, "Assembling a Web Service with EJBs".
- assembling Web services from a JMS queue or topic, see Chapter 9, "Assembling Web Services with JMS Destinations".
- assembling Web services from database resources, see Chapter 10, "Assembling Database Web Services".
- assembling Web services with J2SE 5.0 Annotations, see Chapter 11, "Assembling Web Services with Annotations".
- building J2EE clients, see Chapter 14, "Assembling a J2EE Web Service Client".
- improving performance by data chunking, see "How to Enable Chunked Data Transfer for HTTP 1.1" on page 14-15.
- using the WebServicesAssembler tool to assemble Web services, see Chapter 18, "Using WebServicesAssembler".
- packaging and deploying Web services, see Chapter 19, "Packaging and Deploying Web Services".
- JAR files that are needed to assemble a client, see Appendix A, "Web Service Client APIs and JARs".
- Web services interoperability, see "Ensuring interoperable Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- using quality of service features in Web service clients, see "Managing Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- adding security to a Web service, see the *Oracle Application Server Web Services Security Guide*.
- how to write clients to access Web services secured on the transport level, see "Adding Transport-level Security for Web Services Based on EJBs" and "Accessing Web Services Secured on the Transport Level" in the *Oracle Application Server Web Services Security Guide*.
- adding reliability to a Web service, see "Ensuring Web Service Reliability" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- adding an auditing and logging configuration to a Web service, see "Auditing and Logging Messages" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- processing nonstandard data types, see "Custom Serialization of Java Value Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- the JAX-RPC mapping file, see "JAX-RPC Mapping File Descriptor" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- data types supported by OracleAS Web Services, see "Mapping Java Types to XML and WSDL Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- Oracle JDeveloper tool support for Web service development, see the Oracle JDeveloper on-line help.

16

Using JAX-RPC Handlers

This chapter provides an overview of working with JAX-RPC message handlers.

- Understanding Message Handlers
- How to Write a JAX-RPC Handler
- How to Configure and Register a Server-Side Handler with Ant Tasks
- How to Edit webservices.xml to Register a Server-Side Handler
- Client-Side JAX-RPC Handlers

Understanding Message Handlers

SOAP message handlers can be used to process messages to and from a Web service. There are two kinds of handlers: client and server.

- Client side handlers can intercept messages sent from a client application, the "request", and the corresponding message returned from the service to the client, the "response".
- Server side handlers can intercept messages received by a Web service, the "request", and the corresponding message returned by the service, the "response".

Because handlers gain privileged access to the entire SOAP envelope, they are commonly used for SOAP header processing. Some other common uses for handlers are:

- logging
- auditing
- encryption/decryption

Much of this functionality is provided by the OracleAS Web Services management infrastructure. In many cases, a user-written handler might not be necessary.

For any given Web service or Web service client, there can be zero or more handlers. A collection of handlers constitutes a handler chain. The handler chain is maintained by the JAX-RPC runtime implementation. The default behavior of the runtime implementation is to call each handler in order from the chain. However, a handler can change this processing model based on its implementation of the `javax.xml.rpc.handler.Handler` interface. For example, returning `false` in the `handleRequest` message will halt the runtime from proceeding to the next handler in the chain. Throwing an exception will have a similar effect.

See Also:

The JAX-RPC 1.1 specification provides more information on handlers and the handler model:

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

How to Write a JAX-RPC Handler

To write a JAX-RPC handler, implement the `javax.xml.rpc.handler.Handler` interface.

```
package javax.xml.rpc.handler;
public interface Handler{
    public boolean handleRequest(javax.xml.rpc.handler.MessageContext context);
    public boolean handleResponse(javax.xml.rpc.handler.MessageContext context);
    public boolean handleFault(javax.xml.rpc.handler.MessageContext context);
    public void destroy();
    public void init(javax.xml.rpc.handler.HandlerInfo config);
    public javax.xml.namespace.QName[] getHeaders();
}
```

As an alternative to implementing the `Handler` interface, you can extend the `javax.xml.rpc.handler.GenericHandler` class. This class provides default implementations for all of the interface's methods so there is no need to redefine them in your handler implementation.

See Also:

The API at the following Web site provides more information on the `Handler` interface and `GenericHandler` class.

http://java.sun.com/j2ee/1.4/docs/api/javax/xml/rpc/handler/package-summary.html

How to Configure and Register a Server-Side Handler with Ant Tasks

Handlers are ultimately configured and registered in the Web services deployment descriptor (`webservices.xml`). However, instead of editing the file yourself, you can have WebServicesAssembler generate the proper configuration by specifying the handler classes, that is, classes that implement the `Handler` interface, at development time.

Note: WebServicesAssembler provides Ant tasks that let you configure JAX-RPC message handlers. These tasks are listed in "Ant Tasks that can Configure and Register Handlers" on page 16-3.

Handlers cannot be configured by using the WebServicesAssembler command line.

For example, to add server handlers for a Web service described by the WSDL in Example 17–1, you could use the following Ant task. The server handler appears in bold:

```
<oracle:topDownAssemble appName="hello-service"
                  wsdl="Hello.wsdl"
                  input="./classes"
                  output="build"
```

```
                    ear="dist/hello-service.ear"
                    packageName="oracle.demo"
                    >
                     <oracle:porttype className="oracle.demo.HelloImpl" />
                     <oracle:handler name="ServerHandler"
                             handlerClass="oracle.demo.ServerHelloHandler"/>
</oracle:topDownAssemble>
```

In this example, the server handler `oracle.demo.ServerHelloHandler` is configured for the `hello-service` Web service. Any number of handlers can be added by adding `<handler>` elements, each with unique names. Handlers are added to the chain in the same order in which the handler elements are listed. Although this example is for top down development, handlers can also be added to other Ant tasks using the same `<handler>` element.

The process for adding client-side handlers is almost identical.

See Also:

- "How to Configure Handlers in an Ant Task" on page 18-81 for more information on configuring handlers.
- "Client-Side JAX-RPC Handlers" on page 16-4 for more information on configuring client-side handlers.

Ant Tasks that can Configure and Register Handlers

The following is a list of Ant tasks that can include the `<handler>` element. For more information on these commands, see Chapter 18, "Using WebServicesAssembler".

- aqAssemble
- assemble
- corbaAssemble
- dbJavaAssemble
- ejbAssemble
- genDDs
- genProxy (for client-side generation only)
- jmsAssemble
- plsqlAssemble
- sqlAssemble
- topDownAssemble

How to Edit webservices.xml to Register a Server-Side Handler

If you use Ant tasks to add handlers while generating your Web service, then the appropriate configuration will be added to the `webservices.xml` file for you. There should be no need to add any additional information to this file.

However, if you use the command line, or manually create a Web service deployment descriptor, then you must edit the `webservices.xml` file. You can add handlers to the file by adding `<handler>` subelements to the `<port-component>` element. Example 16–1 illustrates a `<port-component>` element with multiple handlers.

Example 16–1 Sample JAX-RPC Handlers in webservices.xml

```
<port-component>
    ...
<handler>
    <handler-name>First Handler</handler-name>
    <handler-class>oracle.xx.AccountTransactionHandler</handler-class>
    <init-param>
       <param-name>test</param-name>
       <param-value>testValue</param-value>
    </init-param>
</handler>
<handler>
    <handler-name>Second Handler</handler-name>
    <handler-class>oracle.xx.NewAccountHandler</handler-class>
</handler>
   ...
</port-component>
```

Table 16–1 describes the `<handler>` subelements that can be used to specify a server-side handler.

Table 16–1 <handler> Subelements for a Server-Side Handler

Subelement	Description
<handler-class>	The fully-qualified name of the handler's class. The class must implement: `javax.xml.rpc.handler.Handler`.
<handler-name>	A unique name that identifies the handler.
<init-param>	A subelement containing one *param-name*, *param-value* pair. The *param-name*, *param-value* pair represents a single parameter and value that will be passed to the handler's `init` method. There is no limit on the number of `init-param` subelements that can be used in a `<handler>` element.

For more information on the contents of `webservices.xml`, see its schema at the following Web address:

`http://java.sun.com/xml/ns/j2ee/j2ee_web_services_1_1.xsd`

Client-Side JAX-RPC Handlers

On the Web service client, JAX-RPC handlers can intercept and process messages sent from a client application and the corresponding message returned by the service. These handlers can, for example, process the SOAP message. The following sections describe how to register the handlers for use in Web service clients:

- How to Register a JAX-RPC Handler for J2EE Web Service Client
- How to Register a JAX-RPC Handler for a J2SE Web Service Client

How to Register a JAX-RPC Handler for J2EE Web Service Client

For J2EE Web service clients, JAX-RPC handler information appears within the `<service-ref>` element in the deployment descriptor for a given J2EE client. The `<service-ref>` element captures all of the service's J2EE client-related information, such as the location of the WSDL and mapping file, the service interface, the ports the service will run on and their related service endpoint interfaces, and so on.

Unlike server-side handlers, client-side handlers are associated with service references (<service-ref>) instead of port component references (<port-component>). Client-side handlers have a configurable <port-name> parameter that associates a handler with the port of the invoked service. When a service endpoint (WSDL port) is invoked, the value of <port-name> determines which handler is run.

To register a handler for a J2EE Web service client, enter the handler information in the <service-ref> section of its deployment descriptor. The following list identifies the J2EE deployment descriptors for each J2EE component that can act as a Web service client.

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

The contents of <service-ref> element are described by the service-ref (J2EE client) schema, available at the following Web site.

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

Using the handler Element in a J2EE Web Service Client

The <handler> element encapsulates the handler information for a J2EE Web service client. Table 16–2 describes the subelements that it can use.

Table 16–2 <handler> Subelements for a J2EE Web Service Client Handler

Subelement	Description
<handler-class>	The fully-qualified name of the handler's class. The class must implement javax.xml.rpc.handler.Handler.
<handler-name>	The unique name that identifies the handler.
<init-param>	A subelement containing one *param-name*, *param-value* pair. The *param-name*, *param-value* pair represents a single parameter and value that will be passed to the handler's init method. There is no limit on the number of init-param subelements that can be used in a <handler> element.
<port-name>	The name of the port on which the handler will operate.

Enter the handler information at the end of the <service-ref> section, after any port component information. Example 16–2 illustrates a <service-ref> element with two defined handlers. In this example, First Handler is associated with the class oracle.xx.AccountTransactionHandler and runs on portA. Second Handler is associated with the class oracle.xx.NewAccountHandler and runs on portB. First Handler runs only if PortA is invoked and Second Handler runs only if PortB is invoked.

Example 16–2 Sample JAX-RPC Handler for a J2EE Client

```
<service-ref>
    <service-ref-name>service/MyHelloServiceRef</service-ref-name>
    ....
    <port-component-ref>
        ....
    </port-component-ref>
 <handler>
        <handler-name>First Handler</handler-name>
        <handler-class>oracle.xx.AccountTransactionHandler</handler-class>
```

```
        <port-name>portA</port-name>
 </handler>
 <handler>
        <handler-name>Second Handler </handler-name>
        <handler-class>oracle.xx.NewAccountHandler</handler-class>
        <port-name>portB</port-name>
      </handler>
</service-ref>
```

How to Register a JAX-RPC Handler for a J2SE Web Service Client

Client-side JAX-RPC handlers for J2SE clients can be registered using the `genProxy` Ant task for WebServicesAssembler. In Example 16–3, the handler `oracle.demo.ClientHelloHandler` will be available to the J2SE client. Unlike J2EE Web service clients, J2SE clients do not use a deployment descriptor.

Example 16–3 Registering a Handler for a J2SE Web Service Client

```
<oracle:genProxy
            wsdl="http://localhost:8888/hello-service/hello-service?WSDL"
            output="build/src/client"
            packageName="oracle.demo">
            <oracle:handler
                  name="ClientHelloHandler"
                  handlerClass="oracle.demo.ClientHelloHandler" />
      </oracle:genProxy>
```

Limitations

See "Understanding JAX-RPC Handlers" on page F-9.

Additional Information

For more information on:

- processing messages directly, see the SAAJ APIs available from:

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

- OracleAS Web Services extensions to the SAAJ APIs that allow you to work with SOAP 1.2 messages, see "Working with SOAP Messages" on page 5-11.
- assembling Web services from a WSDL, see Chapter 6, "Assembling a Web Service from WSDL".
- assembling stateful Web services, see Chapter 7, "Assembling a Web Service with Java Classes".

  assembling Web services from EJBs, see Chapter 8, "Assembling a Web Service with EJBs".
- assembling Web services from a JMS queue or topic, see Chapter 9, "Assembling Web Services with JMS Destinations".
- assembling Web services from database resources, see Chapter 10, "Assembling Database Web Services".
- assembling Web services with J2SE 5.0 Annotations, see Chapter 11, "Assembling Web Services with Annotations".

17

Processing SOAP Headers

This chapter describes the ways in which you can process SOAP headers:

- How to Process SOAP Headers with Parameter Mapping
- How to Process SOAP Headers by Using Handlers
- How to Process SOAP Headers by Using the ServiceLifecycle Interface

How to Process SOAP Headers with Parameter Mapping

The WebServicesAssembler tool can be used to map SOAP header blocks defined in a `wsdl:binding` element of a WSDL file to method parameters in the generated service endpoint interface (SEI). This allows the SOAP header blocks to be accessed directly inside methods implementing the service endpoint interface.

Example 17–1 illustrates a simple WSDL that explicitly defines a SOAP header.

Example 17–1 Simple WSDL That Explicitly Defines a SOAP Header

```
<definition xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="http://test.com"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"  xmlns:xs="http://www.w3.
org/2001/XMLSchema"
>
   <types/>
   <message name="HelloHeader">
      <part name="header" type="xs:string"/>
   </message>
   <message name="HelloMessage">
      <part name="body" type="xs:string"/>
   </message>
   message name="HelloMessageResponse"/>
   <portType name="HelloPortType">
      <operation name="sayHello">
         <input message="tns:HelloMessage"/>
         <output message="tns:HelloMessageResponse/>
         </operation>
      </portType>
      <binding name="HelloBinding" type="tns:HelloPortType">
         <soap:binding style="rpc" transport="http://schemas.xmlsoap.
org/soap/http" />
         <operation name="sayHello">
            <input>
               <soap:body use="literal" namespace="http://test.com"/>
               <!--the SOAP header must be defined here -->
               <soap:header message=""tns:HelloHeader" part="header"
use="literal"/>
```

```
         </input>
         <output>
            <soap:body use="literal" namespace="http://test.com"/>
         </output>
      </operation>
   </binding>
   <service name="HelloService">
      <port name="HelloPort" binding="tns:HelloBinding">
         <soap:address
location="http://localhost:8888/hello-service/hello-service"/>
      </port>
   </service>
</definition>
```

The WebServicesAssembler tool provides the boolean `mapHeadersToParameters` argument for mapping SOAP headers to parameters. The default for this argument is `true`, so there is no need to explicitly provide it unless you want to suppress the SOAP headers.

You can generate the service endpoint interface with parameter mapping by using either Ant tasks or the WebServicesAssembler tool. The following sample Ant task maps SOAP headers to parameters for the `hello-service` Web service.

```
<oracle:topDownAssemble appName="hello-service"
                   wsdl="Hello.wsdl"
                   input="./classes"
                   output="build"
                   ear="dist/hello-service.ear"
                   packageName="oracle.demo "
                   mapHeadersToParameters="true"
                   >
                   <oracle:porttype className="oracle.demo.HelloImpl" />
</oracle:topDownAssemble>
```

The following is the WebServicesAssembler command line version of the previous example.

```
java -jar wsa.jar -topDownAssemble
                  -wsdl Hello.wsdl
                  -output build
                  -ear dist/hello-services.ear
                  -mapHeadersToParameters true
                  -packageName oracle.demo
```

In this command and Ant task:

- `topDownAssemble`—Creates the required classes and deployment descriptors for a Web service based on a WSDL description. The files can be stored in either an EAR file, a WAR file, or a directory. See topDownAssemble on page 18-22.
- `wsdl`—Specifies the absolute file path, relative file path, or URL to a WSDL document. See "wsdl" on page 18-65.
- `output`—Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. See "output" on page 18-48.
- `ear`—Specifies the name and location of the generated EAR. See "ear" on page 18-44.
- `mapHeadersToParameters`—Indicates if SOAP headers defined in the WSDL should be mapped to parameters for each of the methods in the generated Java code. See "mapHeadersToParameters" on page 18-69.

- `packageName`—Specifies the package name that will be used for generated classes if no package name is declared in the JAX-RPC mapping file. See "packageName" on page 18-48.

  See Also:

  "mapHeadersToParameters" on page 18-69 for more information on this argument.

Ant Tasks and Commands that can Map Headers to Parameters

The following WebServicesAssembler commands and Ant tasks can be used to call the `mapHeaderstoParameters` argument. For more information on these commands, see Chapter 18, "Using WebServicesAssembler".

- genInterface
- genProxy
- topDownAssemble

How to Process SOAP Headers by Using Handlers

JAX-RPC Handlers can be used to process both explicit and implicit SOAP headers. Explicit SOAP headers are those defined in the WSDL document. Example 17–1 illustrates a simple WSDL that defines a SOAP header. Implicit SOAP headers are those that are not necessarily defined in any particular WSDL document, but may be present in a SOAP envelope.

Handlers gain access to a SOAP header using the methods defined in the `javax.xml.rpc.handler.Handler` interface. These methods are:

```
boolean handleRequest(MessageContext context);
boolean handleResponse(MessageContext context);
boolean handleFault(MessageContext context);
```

The `context` argument in each of these messages can be used to view the SOAP headers in the SOAP envelope. The following is an example of a handler implementation viewing the headers on a SOAP request:

```
boolean handleRequest(MessageContext context){
      javax.xml.rpc.handler.soap.SOAPMessageContext smc = (javax.xml.rpc.handler.
soap.SOAPMessageContext)context;
      javax.xml.soap.SOAPHeader sh = smc.getSOAPMessage.getSOAPHeader();
      //the SOAPHeader will contain a list of SOAPHeaderElements (header blocks)
      Iterator it = sh.examineAllHeaderElements();
      //iterate through all the SOAP header elements and print their names
      while(it.hasNext()){
            javax.xml.soap.SOAPHeaderElement elem = (SOAPHeaderElement)it.next();
            System.out.println(elem.getElementName().getQualifiedName());
      }
      return true;
}
```

See Also:

- For more information on the `Handler` interface, see the Java API for XML-Based RPC (JAX-RPC 1.1) specification at the following Web address:

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

- For more information on processing messages directly, see the SOAP with Attachments API for Java (SAAJ) at the following Web address:

  http://java.sun.com/webservices/saaj/index.jsp

How to Process SOAP Headers by Using the ServiceLifecycle Interface

You can manage the life cycle of the service endpoint by implementing the `javax.xml.rpc.server.ServiceLifecycle` interface. The interface has the following methods.

```
void init(Object context);
void destroy();
```

The runtime system will invoke the `init` method and pass a `context` object. The `context` object contains the `ServletEndpointContext` object. From this object, you can extract the `SOAPMessageContext` to process the header upon each invocation.

Example 17–2 demonstrates how to use the `javax.xml.rpc.server.ServiceLifecycle` interface to access SOAP headers:

Example 17–2 Using ServiceLifecycle to Access SOAP Headers

```
public class HelloImpl implements HelloPortType,ServiceLifecycle{
      private Object m_context;

      public void sayHello(String body){
            javax.xml.rpc.server.ServletEndpointContext sec =
(ServletEndpointContext)m_context;
            javax.xml.rpc.handler.soap.SOAPMessageContext mc =
(SOAPMessageContext)sec.getMessageContext();
            javax.xml.soap.SOAPHeader sh = mc.getSOAPMessage().getSOAPHeader();
             // from here you can process all the header
             // blocks in the SOAP header.
}

      //this will be called by the runtime system.
      public void init(Object context){
      m_context = context;
      }
      public void destroy(){
      }
}
```

Implementing this interface enables you to process both implicit and explicit SOAP headers, although it is more useful for the implicit headers.

How to Get HTTP Response and Request Headers

The following sections describe how static stub and Dynamic Invocation Interface (DII) clients can obtain the headers from HTTP response and requests.

- How Stub Clients Can Get Headers with the ServiceLifecycle Interface
- How DII Clients Can Get Headers with the OracleCall Interface
- How DII Clients of WSIF Services Can Get Message Headers

How Stub Clients Can Get Headers with the ServiceLifecycle Interface

J2SE and J2EE static stub clients can use the `HTTP_SERVLET_REQUEST` and `HTTP_SERVLET_RESPONSE` properties in the `oracle.webservices.ServerConstants` class to access the HTTP message header. This property can be used by a service implementation class to get the HTTP servlet request or response when the Web service caller uses HTTP transport.

To use either of these properties, the service implementation must implement `javax.xml.rpc.server.ServiceLifecyle` and store the `Object` passed into the `init` method which is an instance of `javax.xml.rpc.server.ServletEndpointContext`.

When a method in the service implementation class is invoked, the `ServletEndpointContext.getMessageContext` method returns a `javax.xml.rpc.handler.MessageContext`. The `MessageContext.getProperty` method can use either `HTTP_SERVLET_REQUEST` or `HTTP_SERVLET_RESPONSE` as a property name.

If `HTTP_SERVLET_REQUEST` is the requested property, then the returned object is an instance of `javax.servlet.http.HttpServletRequest`. If `HTTP_SERVLET_RESPONSE` is the requested property, then the returned object is an instance of `javax.servlet.http.HttpServletResponse`.

Example 17–3 illustrates how to get an HTTP request header to obtain the IP address. In the example, the `HelloImpl` class implements the `ServiceLifecycle` interface. In this case, the `context` object passed to the `init` method is cast to the `ServletEndpointContext` object. The `destroy` method destroys the service lifecycle. In the implementation of the `getIPAddress` method, the `getMessageContext` method pulls the message context from the `ServletEndpointContext` object. The `getProperty` method uses the `HTTP_SERVLET_REQUEST` property to return the request as an `HttpServletRequest` object. The `getRemoteAddr` method returns the IP address.

Example 17–3 Getting an HTTP Header

```
public class HelloImpl implements ServiceLifecycle  {
   ServletEndpointContext m_context;
   public void init(Object context) throws ServiceException {
       m_context = (ServletEndpointContext)context;
   }

   public void destroy() {
    }

   public String getIPAddress(){
      HttpServletRequest request = (HttpServletRequest)m_context.
          getMessageContext().getProperty(ServerConstants.HTTP_SERVLET_REQUEST);
      return request.getRemoteAddr();
   }

}
```

How DII Clients Can Get Headers with the OracleCall Interface

Dynamic Invocation Interface (DII) clients can use methods in the `oracle.webservices.OracleCall` interface to obtain HTTP request and response headers. To retrieve response headers, use the `getResponseHeaders()` method. This method returns a `Vector`; the elements in the `Vector` are the SOAP response headers.

To retrieve request headers use the `getHeaders()` method. This method also returns a `Vector` where the elements are the SOAP response headers. Typically, the headers are added to the `Vector` with the `addHeaders()` method.

Example 17–4 illustrates DII client code for getting the response headers.

Example 17–4 Retrieving Response Headers in DII Client Code

```
...
OracleCall call = (OracleCall)port.getCall();
Vector headers = call.getResponseHeaders();
if (Vector != null) {
    for(int i = 0;i < headers.size();i++) {
        Element header = (Element) headers.get(i);
        // do something with the header Element
        ...
    }
}
...
```

How DII Clients of WSIF Services Can Get Message Headers

For information on how message headers can be retrieved by DII clients of Web Services Invocation Framework (WSIF) services, see "How DII Clients of WSIF Services Can Get Message Headers" in the *Oracle Application Server Advanced Web Services Developer's Guide*.

Limitations

See "Processing SOAP Headers" on page F-9.

Additional Information

For more information on:

- assembling Web services from a WSDL, see Chapter 6, "Assembling a Web Service from WSDL".
- assembling stateful Web services, see Chapter 7, "Assembling a Web Service with Java Classes".

  assembling Web services from EJBs, see Chapter 8, "Assembling a Web Service with EJBs".
- assembling Web services from a JMS queue or topic, see Chapter 9, "Assembling Web Services with JMS Destinations".
- assembling Web services from database resources, see Chapter 10, "Assembling Database Web Services".
- assembling Web services with J2SE 5.0 Annotations, see Chapter 11, "Assembling Web Services with Annotations".

18

Using WebServicesAssembler

This chapter describes the functionality provided by the WebServicesAssembler tool.

- About the WebServicesAssembler Tool
- Setting Up Ant for WebServicesAssembler
- WebServicesAssembler Commands
- WebServicesAssembler Arguments
- Resolving Name Collisions
- Default Algorithms to Map Between Target WSDL Namespaces and Java Package Names
- How to Establish a Database Connection
- Additional Ant Support for WebServicesAssembler
- How to Assign Multiple Web Services to an EAR or WAR Archive
- How to Represent Java Method Parameter Names in the WSDL

About the WebServicesAssembler Tool

The WebServicesAssembler tool assists in assembling Oracle Application Server Web Services. It enables you to generate the artifacts required to develop and deploy Web services, regardless of whether you are creating the service top down or bottom up. The WebServicesAssembler can also be invoked to create Web service client objects based on a WSDL.

Support for Top Down Web Service Generation

In the top down case, you provide WebServicesAssembler with a WSDL and it creates the service endpoint interfaces. You can then fill in the implementation for the service for any required architecture, such as Java classes.

See Also:

Chapter 6, "Assembling a Web Service from WSDL" for an example of top down Web service assembly that uses WebServicesAssembler.

Support for Bottom Up Web Service Generation

In the bottom up case, you start with existing business logic, such as Java classes, Enterprise Java Beans (EJBs), CORBA objects, JMS queues, or database artifacts such as PL/SQL procedures. WebServicesAssembler uses these artifacts to assemble a WSDL, a mapping file, and the necessary deployment descriptors.

See Also:

See the following chapters for examples of bottom up Web service assembly that use WebServicesAssembler:

- Chapter 7, "Assembling a Web Service with Java Classes"
- Chapter 8, "Assembling a Web Service with EJBs"
- Chapter 9, "Assembling Web Services with JMS Destinations"
- Chapter 10, "Assembling Database Web Services"

Support for XML Schema-Driven Web Service Generation

In the schema-driven case, you start with an XML schema and generate Java beans. Once you have the Java beans, you write the interface that uses the beans as arguments and use the bottom up paradigm to generate the WSDL, mapping file and deployment descriptors.

While you could use JAX-B or Toplink to generate beans from XML schemas, you could also use WebServicesAssembler or Ant tasks.

See Also:

"Using Custom Serialization in Schema-Driven Web Service Development" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on using Ant tasks or WebServicesAssembler to generate a Web service from a schema.

Support for Deployment

Although the OC4J container handles deployment, the WebServicesAssembler tool assists you by ensuring that the application archives it generates are properly prepared for deployment. WebServicesAssembler handles the generation of all relevant deployment descriptors and maps proprietary configuration needed by the applications into Oracle specific deployment files. Applications based on Java classes or EJB 2.1 are deployable across different containers. These Web services are in a J2EE standard deployable form and adhere to industry standards, such as the JAX-RPC, Enterprise Web Services 1.1, and Web Services-Interoperability (WS-I) specifications.

Support for Command Line Invocation and Ant Tasks

The WebServicesAssembler tool can be invoked either on the command line or by Ant tasks. WebServicesAssembler allows you flexibility in how you assemble your Web service. You can break the assembly process into a number of steps that let you more closely control how the Web service is created. For example, you can perform the following tasks.

- Use another mechanism to create deployment descriptors

  To do this, WebServicesAssembler provides command line arguments to add and delete services from deployment descriptors. This means that you can start with a set of hand-coded deployment descriptors that contain information that WebServicesAssembler does not generate. You can then use WebServicesAssembler to append information to these descriptors.

- Control when artifacts are compiled and which classpath to use
- Control when artifacts are packaged into an archive
- Control the content of the archive
- Create platform-independent build files

Command Line Syntax

The WebServicesAssembler tool supports a number of commands. When you invoke WebServicesAssembler, only one command can appear on the command line.

A typical command line has the following syntax.

```
java -jar [OC4J_HOME]/webservices/lib/wsa.jar -[command] -[argument name][argument value]...
```

In this example, *OC4J_HOME* is where OC4J was installed and `wsa.jar` is the name of the WebServicesAssembler JAR file. Note the following command line syntax rules.

- A command must be specified first on the command line.
- Commands and arguments can be preceded with a dash "-", however, it is not required.
- A space must separate an argument name from its value.
- All argument names are case-sensitive when used in Ant tasks. They are not case-sensitive when used on the command line.
- If a white space is required in an argument value, the value must be escaped.

  For example, on Windows and Linux, white space must be double-quoted. Other operating systems may require a different way to pass white space as parameters to a Java executable.
- All arguments that are used after the command can be placed in any order.
- Typically, an argument can appear only once on the command line. Exceptions to this rule are noted in the individual argument descriptions.

Setting Up Ant for WebServicesAssembler

To call WebServicesAssembler commands from Ant tasks, you may need to make some changes and additions to your installation of Ant.

See Also:

"How to Set Up Ant for WebServicesAssembler" on page 4-3 for a description of the changes and additions needed to call Ant tasks.

WebServicesAssembler Commands

This section describes the commands available for the WebServicesAssembler tool. The commands are organized into the following categories based on the functionality they provide.

- Web Service Assembly Commands—assemble Web services. These commands create all of the files necessary to create a deployable archive such as a WAR, an EAR, or an EJB JAR.
- WSDL Management Commands—perform actions on a WSDL, such as generate a WSDL for bottom up development, manage its contents and location, and determine whether it can be processed by WebServicesAssembler.
- Java Generation Commands—generate code to create a Java interface from a WSDL, a proxy/stub, or JAX-RPC value type classes.
- Deployment Descriptor Generation Commands—generate deployment descriptors for EARs, WARs, or EJB JARs.

- Maintenance Commands—return a short description of WebServicesAssembler commands and the version number of the tool.

Web Service Assembly Commands

The following commands can be used to assemble a Web service.

- aqAssemble—Assembles a Web service from a Advanced Queue in the database
- assemble—Assembles a Web service from Java classes and annotated Java classes. (See Chapter 11, "Assembling Web Services with Annotations" for more information on using J2SE 5.0 Web Services Annotations.)
- corbaAssemble—Assembles a Web service endpoint from a CORBA servant object
- dbJavaAssemble—Assembles Web services from a Java class in a database
- ejbAssemble—Assembles an EJB as a Web service
- jmsAssemble—Assembles a JMS Endpoint Web service
- plsqlAssemble—Assembles a Web service from a PL/SQL package
- sqlAssembles—Assembles a Web service from SQL queries and DML statements
- topDownAssemble—Assembles Web service classes and deployment descriptors from a WSDL specification

These commands share the following functionality and behavior:

- Each of the `*Assemble` commands creates all of the files required for a deployable archive.
- Each of the `*Assemble` commands, except `topDownAssemble`, calls the `assemble` command after they generate a Java class.
- The files created by the `*Assemble` commands are placed in the staging directory structure under the directory specified by the `output` argument.

  See Figure 18–1 for an illustration of the staging directory structure created by the `*Assemble` commands.
- The names and contents of the `ear` and `war` directories are affected by the values you provide for the `ear` and `war` arguments.

  See "ear" on page 18-44 and "war" on page 18-51 for more information on these arguments.
- The output of each of the `*Assemble` commands, except `ejbAssemble`, is an EAR and a WAR file, and an optional directory that contains the contents of a WAR that can be deployed into an OC4J instance.
- The contents of the classpath are not copied to the archive. If you specify the `classpath` or `input` arguments to any of the `*Assemble` commands, you must be sure that the classes found in the classpath are also available in the server.
- You can add files to an EAR or WAR file before it is archived by the `*Assemble` commands.

  See "How to Add Files to an Archive" on page 18-84 for more information on this topic.

Figure 18–1 illustrates the staging directory structure created by the `*Assemble` commands. Beneath the specified output directory, the commands create three subdirectories: `ear`, `src`, and `war`.

- The `ear` subdirectory contains the contents of the generated EAR file. It also contains a `META-INF` subdirectory that contains the `application.xml` file.
- The `src` subdirectory contains the generated source files. It also contains a `proxy` subdirectory that contains the Java proxy files. If the Web Service is assembled from a WSDL (top down), `src` also contains subdirectory with a skeleton of the generated service endpoint interface and value type classes.
- The `war` subdirectory contains the contents of the generated WAR file. This subdirectory also contains a `WEB-INF` subdirectory. `WEB-INF` contains the mapping files and the standard and Oracle proprietary deployment descriptors. These files include the *web-service_name*`_java_wsdl_mapping.xml`, `web.xml`, `oracle-webservices.xml` and `webservices.xml` files.

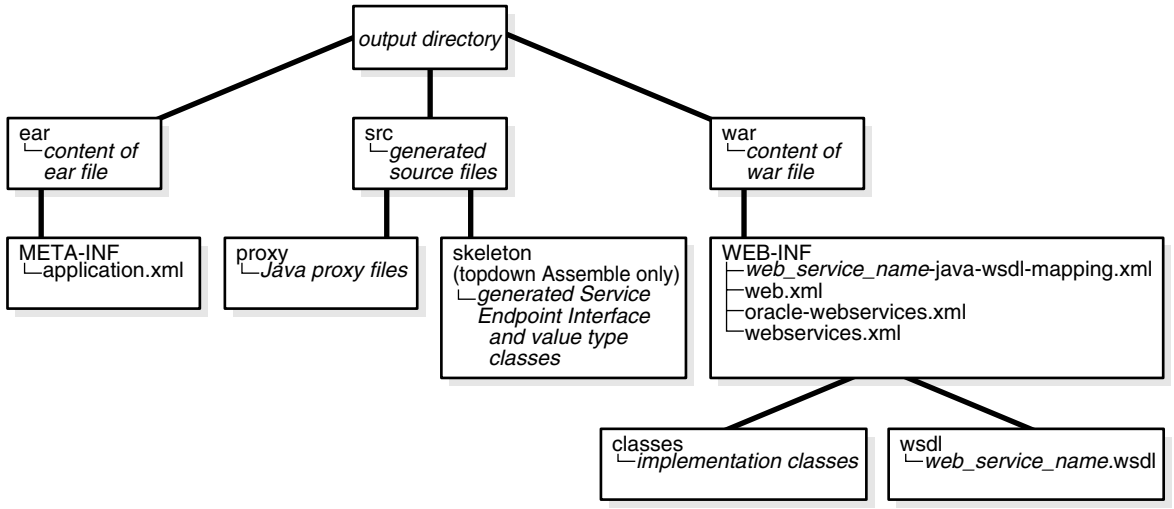
  `WEB-INF` also contains the `classes` and `wsdl` subdirectories. The `classes` subdirectory contains the implementation classes. The `wsdl` subdirectory contains the Web service's WSDL file.

Note: WebServicesAssembler does not remove files from the `war` and `ear` staging directories by default. If you use the same output directory for multiple invocations of WebServicesAssembler, then you may find extra, unwanted files in the WAR and EAR archives. If you want to avoid this behavior, do one of the following:

- specify a different output directory for each invocation of WebServicesAssembler
- delete the contents of the output directory in between calls to the WebServicesAssembler invocations

---

Figure 18–1 Staging Directory Structure Created by the *Assemble Commands



aqAssemble

Use the `aqAssemble` command to generate Web services from an advanced queue in the database. To use this command, you must connect to a database.

The `aqAssemble` command can also add WSIF bindings for an advanced queue to the WSDL. Use the `wsifDbBinding` argument to add WSIF bindings to the WSDL when you are exposing a database resource as a Web service. You must specify the

`className` of the resource's Oracle JPublisher-generated Java class and a database connection.

See Also:

- "How to Assemble a Web Service from an Oracle Streams AQ" on page 10-22 provides more information about using the `aqAssemble` command to expose an advanced queue as a Web service.
- "How to Establish a Database Connection" on page 18-75 describes the arguments that allow you to connect to a database.
- "Configuring a WSIF Endpoint for Database Resources" in the *Oracle Application Server Advanced Web Services Developer's Guide* provides more information on adding bindings for database resources to the WSDL.

Command Line Example:

```
java -jar wsa.jar -aqAssemble
                  -dbUser scott/tiger
                  -sql ToyQueue
                  -dbConnection jdbc:oracle:thin:@dsunrde22:1521:sqlj
                  -dataSource jdbc/OracleManagedDS
                  -appName query
```

Ant Task Example:

```
<oracle:aqAssemble
     dbUser="scott/tiger"
     sql="ToyQueue"
     dbConnection="jdbc:oracle:thin:@dsunrde22:1521:sqlj"
     dataSource="jdbc/OracleManagedDS"
     appName="query"
/>
```

Required Arguments:

To use the `aqAssemble` command, you must specify the following arguments.

- **Database Assembly Arguments:** sql

  You must also connect to the database. To do this, use one of the following combinations of arguments.

  - dataSource, dbConnection, and dbUser
  - dataSource

All Arguments:

The following list identifies all of the arguments that can be used with the `aqAssemble` command.

- **Database Assembly Arguments:** aqConnectionFactoryLocation, aqConnectionLocation, dataSource, dbConnection, dbUser, jpubProp, sql (required), sqlTimeout, sysUser, useDataSource, wsifDbBinding, wsifDbPort

  For more information on these arguments, see "Database Assembly Arguments" on page 18-53.

- **Deployment Descriptor Arguments:** appendToExistingDDs, context, ddFileName, uri

For more information on these arguments, see "Deployment Descriptor Arguments" on page 18-62.

- **General Assembly Arguments:** appName, debug, ear, emptySoapAction, help, interfaceName, mappingFileName, output, packageName, portName, restSupport, schema, serviceName, useDimeEncoding, war

  For more information on these arguments see, "General Web Services Assembly Arguments" on page 18-41.

- **Java Generation Arguments:** valueTypeClassName

  For more information on this argument, see "Java Generation Arguments" on page 18-68.

- **Message Format Arguments:** style, use

  For more information on these arguments, see "Message Format Arguments" on page 18-67.

- **Session Arguments:** timeout

  For more information on this argument, see "Session Arguments" on page 18-51.

- **WSDL Access Arguments:** httpNonProxyHosts, httpProxyHost, httpProxyPort

  For more information on these arguments, see "WSDL Access Arguments" on page 18-64.

- **WSDL Management Arguments:** createOneWayOperations, genQos, qualifiedElementForm, soapVersion, targetNamespace, typeNamespace

  For more information on these arguments, see "WSDL Management Arguments" on page 18-65.

Ant Task Support:

- `<proxy>` tags. For more information, see "How to Configure Proxy Generation in an Ant Task" on page 18-77.
- `<port>` tags. In the `<port>` tag, `aqAssemble` can use these arguments: bindingName, `name` (same as `portName`), portName, sendConnectionFactoryLocation, sendQueueLocation, soapVersion, and uri. For more information, see "How to Configure a Port in an Ant Task" on page 18-78.
- `<handler>` tags. For more information, see "How to Configure Handlers in an Ant Task" on page 18-81.

assemble

Use the `assemble` command to generate a Web service bottom up. The command creates all of the files required to create a deployable archive. These files include the proprietary `oracle-webservices.xml` deployment descriptor. The command can generate stateless Web services whether the transport mechanism is HTTP or JMS.

To find the Java implementation class, you must specify an `input` or `classpath` argument. If you specify either of these arguments, you must be sure that the classes found in the classpath are also available in the server. This is because the contents of the classpath are not copied to the archive.

Assembling WSIF Bindings into the WSDL

In addition to generating a Web service, the `assemble` command can also add WSIF bindings to the WSDL. Use the `wsifJavaBinding` argument to add WSIF bindings

to the WSDL when you are exposing a Java class as a Web service. You must also specify the Java class with the `className` argument.

See Also:

- "Configuring a WSIF Endpoint for Multiple Java Ports" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- "Using Web Services Invocation Framework" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on WSIF.

Assembling a Web Service with J2SE 5.0 Annotations

If you are using the `assemble` command to assemble a Web service from Java classes that contain J2SE 5.0 Annotations, then use the `className` argument to identify the implementation class. The `@WebService` annotation must be present on the class declaration.

J2SE 5.0 annotations require a class name because annotations can appear in either the interface or the implementation class. If the annotations appear only in the interface, WebServicesAssembler can obtain them through its referenced implementation class. If the implementation class also contains annotations, WebServicesAssembler will process them.

The following example illustrates the `assemble` command being used to generate a Web service from a Java class that contains J2SE 5.0 annotations. The `className` argument specifies the `com.mycompany.HelloImpl` class.

```
java -jar wsa.jar assemble -appName myService -className com.mycompany.HelloImpl
-output wsdl
```

If you want WebServicesAssembler to process only the annotations in the interface, then enter the `@WebService` annotation with the `serviceEndpointInterface` property in the implementation class file. J2SE 5.0 annotations will expect all remaining annotations to appear in the service endpoint interface class. For example, if you enter the following annotation in the implementation class file, then WebServicesAssembler will process only the annotations in the `demo.myInterface` interface.

```
@WebService(serviceEndpointInterface="demo.myInterface")
```

See Also:

Chapter 11, "Assembling Web Services with Annotations" for more information on using J2SE 5.0 Annotations to assemble a Web service.

The following command line and Ant task examples demonstrate how to use the `assemble` command to generate a Web service bottom up. The command will create an EAR file named `build/myService.ear`.

Command Line Example:

```
java -jar wsa.jar -assemble
                  -input myservice.jar
                  -className com.mycompany.HelloImpl
                  -interfaceName com.myCompany.myService.Hello
                  -output build
                  -appName myService
```

Ant Task Example:

```
<oracle:assemble appName="myService"
          output="build"
          input="myservice.jar
          >
          <oracle:porttype
            interfaceName="com.myCompany.myService.Hello"
            className="com.mycompany.HelloImpl"/>
</oracle:assemble>
```

Required Arguments:

To use the `assemble` command, you must specify the following arguments.

- **General Assembly Arguments:** className

All Arguments

The following list identifies all of the arguments that can be used with the `assemble` command.

- **Deployment Descriptor Arguments:** appendToExistingDDs, context, ddFileName, uri

  For more information on these arguments, see "Deployment Descriptor Arguments" on page 18-62.

- **General Assembly Arguments:** appName, bindingName, className (required), classpath, debug, ear, emptySoapAction, help, input, interfaceFileName, interfaceName, mappingFileName, output, portName, portTypeName, restSupport, schema, serviceName, strictJaxrpcValidation, useDimeEncoding, war

  For more information on these arguments see, "General Web Services Assembly Arguments" on page 18-41.

- **Java Generation Arguments:** valueTypeClassName, wsifJavaBinding

  For more information on these arguments, see "Java Generation Arguments" on page 18-68.

- **JMS Assembly Arguments:** sendConnectionFactoryLocation, sendQueueLocation

  For more information on these argument, see "JMS Assembly Arguments" on page 18-58.

- **Message Format Arguments:** mtomSupport, style, use

  For more information on these arguments, see "Message Format Arguments" on page 18-67.

- **Session Arguments:** callScope, recoverable, session, timeout

  For more information on these arguments, see "Session Arguments" on page 18-51.

- **WSDL Access Arguments:** httpNonProxyHosts, httpProxyHost, httpProxyPort

  For more information on these arguments, see "WSDL Access Arguments" on page 18-64.

- **WSDL Management Arguments:** createOneWayOperations, genQos, qualifiedElementForm, soapVersion, targetNamespace, typeNamespace

  For more information on these arguments, see "WSDL Management Arguments" on page 18-65.

Ant Task Supports:

- <proxy> tags. For more information, see "How to Configure Proxy Generation in an Ant Task" on page 18-77.
- <port> tags. In the <port> tag, assemble can use these arguments: bindingName, name (same as portName), portName, sendConnectionFactoryLocation, sendQueueLocation, soapVersion, and uri. For more information, see "How to Configure a Port in an Ant Task" on page 18-78.
- <porttype> tags. For more information, see "How to Configure a Port Type in an Ant Task" on page 18-80.
- <handler> tags. For more information, see "How to Configure Handlers in an Ant Task" on page 18-81.
- tags to specify WSIF bindings for multiple ports. For more information, see "Configuring a WSIF Endpoint for Multiple Java Ports" in the *Oracle Application Server Advanced Web Services Developer's Guide*.

corbaAssemble

Use the corbaAssemble command to expose a CORBA servant object as a Web service. The command takes a CORBA IDL file and CORBA naming properties as input. It outputs all of the files required to create a deployable archive.

Internally, WebServicesAssembler will invoke the IDL-to-Java compiler (idlj) that can be found in the environment. The JDK bin directory must be part of the path environment variable. WebServicesAssembler uses the IDL-to-Java compiler to compile the IDL file into Java classes.

Command Line Example

```
java -jar wsa.jar -corbaAssemble
                  -idlInterfaceName oraclecorba.Hello
                  -corbanameURL corbaname::corba.orbd.host:1050#oracle.corba/Hello
                  -idlFile ./Hello.idl
                  -uri /corba_hello
                  -output dist
                  -context corba_hello
                  -targetNamespace http://oracle.j2ee.ws/corba/Hello
                  -typeNamespace http://oracle.j2ee.ws/corba/Hello/types
                  -serviceName Corba_hello
                  -appName corba_hello
                  -style rpc
                  -use literal
```

Ant Task Example:

```
<oracle:corbaAssemble idlInterfaceName="oraclecorba.Hello"
                corbanameURL="corbaname::corba.orbd.host:1050#oracle.corba/Hello"
                idlFile="./Hello.idl"
                output="dist"
                context="corba_hello"
                targetNamespace="http://oracle.j2ee.ws/corba/Hello"
                typeNamespace="http://oracle.j2ee.ws/corba/Hello/types"
                serviceName="Corba_hello"
                appName="corba_hello"
                style="rpc"
                use="literal"
                >
                <oracle:port  uri="/corba_hello" />
```

```
</oracle:corbaAssemble>
```

Required Arguments:

To use the `corbaAssemble` command, you must specify the following arguments.

- **CORBA Assembly Arguments:** idlInterfaceName

All Arguments:

The following list identifies all of the arguments that can be used with the `corbaAssemble` command.

- **CORBA Assembly Arguments:** corbanameURL, corbaObjectPath, idlFile, idlInterfaceName (required), idljPath, ORBInitialHost, ORBInitialPort, ORBInitRef

  For more information on these arguments, see "CORBA Assembly Arguments" on page 18-52.

- **Deployment Descriptor Arguments:** appendToExistingDDs, context, ddFileName, uri

  For more information on these arguments, see "Deployment Descriptor Arguments" on page 18-62.

- **General Assembly Arguments:** appName, bindingName, className, classpath, debug, ear, emptySoapAction, help, mappingFileName, output, packageName, portName, portTypeName, restSupport, schema, serviceName, useDimeEncoding, war

  For more information on these arguments see, "General Web Services Assembly Arguments" on page 18-41.

- **JMS Assembly Arguments:** sendConnectionFactoryLocation, sendQueueLocation

  For more information on these arguments, see "JMS Assembly Arguments" on page 18-58.

- **Message Format Arguments:** style, use

  For more information on these arguments, see "Message Format Arguments" on page 18-67.

- **WSDL Access Arguments:** httpNonProxyHosts, httpProxyHost, httpProxyPort

  For more information on these arguments, see "WSDL Access Arguments" on page 18-64.

- **WSDL Management Arguments:** createOneWayOperations, genQos, qualifiedElementForm, soapVersion, targetNamespace, typeNamespace

  For more information on these arguments, see "WSDL Management Arguments" on page 18-65.

Ant Task Support:

- `<proxy>` tags. For more information, see "How to Configure Proxy Generation in an Ant Task" on page 18-77.
- `<port>` tags. In the `<port>` tag, `corbaAssemble` can use these arguments: bindingName, `name` (same as `portName`), portName, sendConnectionFactoryLocation, sendQueueLocation, soapVersion, and uri. For more information, see "How to Configure a Port in an Ant Task" on page 18-78.
- `<handler>` tags. For more information, see "How to Configure Handlers in an Ant Task" on page 18-81.

dbJavaAssemble

Use the `dbJavaAssemble` command to generate Web services from a Java class inside the Java VM in an Oracle database. To use this command, you must connect to a database.

The `dbJavaAssemble` command can also add WSIF bindings for a Java class inside the Java VM to the WSDL. Use the `wsifDbBinding` argument to add WSIF bindings to the WSDL when you are exposing a database resource as a Web service. You must specify the `className` of the resource's Oracle JPublisher-generated Java class and a database connection.

See Also:

- "How to Assemble a Server-Side Java Class as a Web Service" on page 10-32 provides more information on using the `dbJavaAssemble` command to generate Web services from a Java class inside the Java VM in an Oracle database.
- "How to Establish a Database Connection" on page 18-75 describes the arguments that allow you to connect to a database.
- "Configuring a WSIF Endpoint for Database Resources" in the *Oracle Application Server Advanced Web Services Developer's Guide* provides more information on adding bindings for database resources to the WSDL.

Command Line Example:

```
java -jar wsa.jar -dbJavaAssemble
                  -dbJavaClassName oracle.sqlj.checker.JdbcVersion
                  -dbUser scott/tiger
                  -dbConnection jdbc:oracle:thin:@dsunrde22:1521:sqlj
                  -dataSource jdbc/OracleManagedDS
                  -appName javacallin
```

Ant Task Example:

```
<oracle:dbJavaAssemble
     dbUser="scott/tiger"
     dbJavaClassName="oracle.sqlj.checker.JdbcVersion"
     dbConnection="jdbc:oracle:thin:@dsunrde22:1521:sqlj"
     dataSource="jdbc/OracleManagedDS"
     appName="javacallin"
/>
```

Required Arguments:

To use the `dbJavaAssemble` command, you must specify the following arguments.

- **Database Assembly Arguments:** dbJavaClassName

  The `dbJavaClassName` argument specifies the Java class in the database that is to be published as a Web service.

  You must also connect to the database. To do this, use one of the following combinations of arguments:

  - dataSource, dbConnection, and dbUser
  - dataSource

All Arguments:

The following list identifies all of the arguments that can be used with the `dbJavaAssemble` command.

- **Database Assembly Arguments:** dataSource, dbConnection, dbJavaClassName (required), dbUser, jpubProp, sysUser, useDataSource, wsifDbBinding, wsifDbPort

  For more information on these arguments, see "Database Assembly Arguments" on page 18-53.

- **Deployment Descriptor Arguments:** appendToExistingDDs, context, ddFileName, uri

  For more information on these arguments, see "Deployment Descriptor Arguments" on page 18-62.

- **General Assembly Arguments:** appName, className, classpath, debug, ear, emptySoapAction, help, interfaceName, mappingFileName, output, packageName, portName, restSupport, schema, serviceName, useDimeEncoding, war

  For more information on these arguments see, "General Web Services Assembly Arguments" on page 18-41.

- **Java Generation Arguments:** valueTypeClassName

  For more information on this argument, see "Java Generation Arguments" on page 18-68.

- **Message Format Arguments:** style, use

  For more information on these arguments, see "Message Format Arguments" on page 18-67.

- **Session Arguments:** timeout

  For more information on this argument, see "Session Arguments" on page 18-51.

- **WSDL Access Arguments:** httpNonProxyHosts, httpProxyHost, httpProxyPort

  For more information on these arguments, see "WSDL Access Arguments" on page 18-64.

- **WSDL Management Arguments:** createOneWayOperations, genQos, qualifiedElementForm, soapVersion, targetNamespace, typeNamespace

  For more information on these arguments, see "WSDL Management Arguments" on page 18-65.

Ant Task Support:

- `<proxy>` tags. For more information, see "How to Configure Proxy Generation in an Ant Task" on page 18-77.
- `<port>` tags. In the `<port>` tag, `dbJavaAssemble` can use these arguments: bindingName, portName (or `name`), sendConnectionFactoryLocation, sendQueueLocation, soapVersion, and uri. For more information, see "How to Configure a Port in an Ant Task" on page 18-78.
- `<handler>` tags. For more information, see "How to Configure Handlers in an Ant Task" on page 18-81.

ejbAssemble

Use the `ejbAssemble` command to create an EAR or EJB JAR that can expose an EJB as a Web service. You must specify a valid version 2.1 EJB JAR as input; the system will create a WSDL and the proprietary `oracle-webservices.xml` deployment descriptor.

By default, this command creates an EAR file that contains a version 2.1 EJB file that can be deployed directly. The `oracle-webservices.xml` file specifies the context and URL pattern that can be used to access the EJB as a Web service.

If you do not want to deploy the EJB as an EAR file, you can create an EJB JAR file instead. For example, this enables you to deploy the EJB as a JAR file to a J2EE container, or to use other J2EE application deployment tools, such as Ant. To save the EJB as a JAR file, specify a directory for the `ear` argument.

Assembling WSIF Bindings into the WSDL

The `ejbAssemble` command can also add WSIF bindings to the WSDL. Use the `wsifEjbBinding` argument to add WSIF bindings when you are exposing an EJB as a Web service. You must specify the EJB's home interface in the `className` argument and its JNDI name in the `jndiName` argument.

Note: You cannot use the `ejbAssemble` command or WebServicesAssembler to assemble Web services from version 3.0 EJBs. To do this, you must use J2SE 5.0 Annotations. For more information, see "Steps to Use Annotations to Assemble a Web Service from a Version 3.0 EJB" on page 11-12.

See Also:

- "ear" on page 18-44 for a description of the different ways in which you can specify this parameter.
- "Using Web Services Invocation Framework" in the *Oracle Application Server Advanced Web Services Developer's Guide* provides more information on WSIF.
- "Configuring a WSIF Endpoint for EJBs" in the *Oracle Application Server Advanced Web Services Developer's Guide* provides information on adding WSIF bindings for individual and for multiple ports.
- Chapter 8, "Assembling a Web Service with EJBs" provides more information on using the `ejbAssemble` command to expose a version 2.1 EJB as a Web service.

The following command line and Ant task examples create the EAR file `build/myService.ear`.

Command Line Example:

```
java -jar wsa.jar -ejbAssemble
                  -output build
                  -input myEjb.jar
                  -ejbName myEjb
                  -appName myService
```

Ant Task Example:

```
<oracle:ejbAssemble output="build"
             input="myEjb.jar"
             ejbName="myEjb"
             appName="myService"
             />
```

Required Arguments:

To use the `ejbAssemble` command, you must specify the following arguments.

- **General Assembly Arguments:** ejbName, input

All Arguments:

The following list identifies all of the arguments that can be used with the `ejbAssemble` command.

- **Deployment Descriptor Arguments:** appendToExistingDDs, context, ddFileName, uri

  For more information on these arguments, see "Deployment Descriptor Arguments" on page 18-62.

- **General Assembly Arguments:** appName, bindingName, className, classpath, debug, ear, ejbName (required), emptySoapAction, help, initialContextFactory, input (required), interfaceName, jndiName, jndiProviderURL, mappingFileName, output, portName, portTypeName, restSupport, schema, serviceName, strictJaxrpcValidation, useDimeEncoding

  For more information on these arguments see, "General Web Services Assembly Arguments" on page 18-41.

- **Java Generation Arguments:** valueTypeClassName, wsifEjbBinding

  For more information on these arguments, see "Java Generation Arguments" on page 18-68.

- **JMS Assembly Arguments:** sendConnectionFactoryLocation, sendQueueLocation

  For more information on these arguments, see "JMS Assembly Arguments" on page 18-58.

- **Message Format Arguments:** style, use

  For more information on these arguments, see "Message Format Arguments" on page 18-67.

- **WSDL Access Arguments:** httpNonProxyHosts, httpProxyHost, httpProxyPort

  For more information on these arguments, see "WSDL Access Arguments" on page 18-64.

- **WSDL Management Arguments:** createOneWayOperations, genQos, qualifiedElementForm, soapVersion, targetNamespace, typeNamespace

  For more information on these arguments, see "WSDL Management Arguments" on page 18-65.

Additional Ant Support:

- `<proxy>` tags. For more information, see "How to Configure Proxy Generation in an Ant Task" on page 18-77.

- <port> tags. In the <port> tag, ejbAssemble can use these arguments: bindingName, name (same as portName), portName, sendConnectionFactoryLocation, sendQueueLocation, soapVersion, and uri. For more information, see "How to Configure a Port in an Ant Task" on page 18-78.
- <handler> tags. For more information, see "How to Configure Handlers in an Ant Task" on page 18-81.
- tags to specify WSIF bindings for multiple ports. For more information, see "Configuring a WSIF Endpoint for Multiple EJB Ports" in the *Oracle Application Server Advanced Web Services Developer's Guide*.

jmsAssemble

Use the jmsAssemble command to expose a JMS destination (queue or topic) as a Web service. JMS Web services have two types of operations: send and receive. The send operation sends a message to the JMS destination. The receive operation retrieves a message from the destination. Some of the JMS message properties (for example, correlation ID) can be exposed as SOAP headers.

See Also:

Chapter 9, "Assembling Web Services with JMS Destinations" provides more information on using the jmsAssemble command to expose a JMS destination as a Web service.

Command Line Example:

```
java -jar wsa.jar -jmsAssemble
           -sendConnectionFactoryLocation jms/ws/mdb/theQueueConnectionFactory
           -sendQueueLocation jms/ws/mdb/theQueue
           -replyToConnectionFactoryLocation jms/ws/mdb/logQueueConnectionFactory
           -replyToQueueLocation jms/ws/mdb/logQueue
           -linkReceiveWithReplyTo true
           -targetNamespace http://oracle.j2ee.ws/jms-doc
           -typeNamespace http://oracle.j2ee.ws/jms-doc/types
           -serviceName JmsService
           -appName jms_service
           -context jms_service
           -input ./demo/build/mdb_service.jar
           -uri JmsService
           -output ./demo/dist
```

Ant Task Example:

```
<oracle:jmsAssemble
       linkReceiveWithReplyTo="true"
       targetNamespace="http://oracle.j2ee.ws/jms-doc"
       typeNamespace="http://oracle.j2ee.ws/jms-doc/types"
       serviceName="JmsService"
       appName="jms_service"
       context="jms_service"
       input="./demo/build/mdb_service.jar"
       output="./demo/dist"
       >
       <oracle:port  uri="JmsService"
          sendConnectionFactoryLocation="jms/ws/mdb/theQueueConnectionFactory"
          sendQueueLocation="jms/ws/mdb/theQueue"
          replyToConnectionFactoryLocation="jms/ws/mdb/logQueueConnectionFactory"
          replyToQueueLocation="jms/ws/mdb/logQueue"/>
</oracle:jmsAssemble>
```

Required Arguments:

To use the `jmsAssemble` command, you must specify the following arguments.

- **JMS Assembly Arguments:** either replyToConnectionFactoryLocation or sendConnectionFactoryLocation

All Arguments:

The following list identifies all of the arguments that can be used with the `jmsAssemble` command.

- **Deployment Descriptor Arguments:** appendToExistingDDs, context, ddFileName, uri

  For more information on these arguments, see "Deployment Descriptor Arguments" on page 18-62.

- **General Assembly Arguments:** appName, bindingName, debug, ear, emptySoapAction, help, input, output, portName, portTypeName, serviceName, useDimeEncoding, war

  For more information on these arguments see, "General Web Services Assembly Arguments" on page 18-41.

- **JMS Assembly Arguments:** deliveryMode, genJmsPropertyHeader, jmsTypeHeader, linkReceiveWithReplyTo, payloadBindingClassName, priority, receiveConnectionFactoryLocation, receiveQueueLocation, receiveTimeout, receiveTopicLocation, replyToConnectionFactoryLocation, replyToQueueLocation, replyToTopicLocation, sendConnectionFactoryLocation, sendQueueLocation, sendTopicLocation, timeToLive, topicDurableSubscriptionName

  For more information on these arguments, see "JMS Assembly Arguments" on page 18-58.

- **WSDL Management Arguments:** createOneWayOperations, genQos, qualifiedElementForm, soapVersion, targetNamespace, typeNamespace

  For more information on these arguments, see "WSDL Management Arguments" on page 18-65.

Ant Task Supports:

- `<proxy>` tags. For more information, see "How to Configure Proxy Generation in an Ant Task" on page 18-77.
- `<port>` tags. In the `<port>` tag, `jmsAssemble` can use these arguments: bindingName, `name` (same as `portName`), portName, sendConnectionFactoryLocation, sendQueueLocation, soapVersion, and uri. For more information, see "How to Configure a Port in an Ant Task" on page 18-78.
- `<handler>` tags. For more information, see "How to Configure Handlers in an Ant Task" on page 18-81.

plsqlAssemble

Use the `plsqlAssemble` command to generate Web services from a PL/SQL package containing stored procedures and functions. To use this command, you must connect to a database.

See Also:

- "How to Establish a Database Connection" on page 18-75 describes the arguments that allow you to connect to a database.
- "How to Assemble a Web Service from a PL/SQL Package" on page 10-9 provides more information on using the `plsqlAsemble` command to expose PL/SQL packages as Web services.

Assembling WSIF Bindings into the WSDL

The `plsqlAssemble` command can also add WSIF bindings for a PL/SQL package to the WSDL. Use the `wsifDbBinding` argument to add WSIF bindings to the WSDL when you are exposing a database resource as a Web service. You must specify the `className` of the resource's Oracle JPublisher-generated Java class and a database connection.

See Also:

"Configuring a WSIF Endpoint for Database Resources" in the *Oracle Application Server Advanced Web Services Developer's Guide* provides more information on adding bindings for database resources to the WSDL.

Command Line Example:

```
java -jar wsa.jar -plsqlAssemble
                  -appName query
                  -dbUser scott/tiger
                  -sql Company
                  -dbConnection jdbc:oracle:thin:@dsunrde22:1521:sqlj
                  -dataSource jdbc/OracleManagedDS
```

Ant Task Example:

```
<oracle:plsqlAssemble
     dbUser="scott/tiger"
     appName="query"
     sql="Company"
     dbConnection="jdbc:oracle:thin:@dsunrde22:1521:sqlj"
     dataSource="jdbc/OracleManagedDS"
/>
```

Required Arguments:

To use the `plsqlAssemble` command, you must specify the following arguments.

- **Database Assembly Arguments:** sql

  The `sql` argument specifies the name of the PL/SQL package that is to be published as a Web service.

  You must also connect to the database. To do this, use one of the following combinations of arguments.

  - dataSource, dbConnection, and dbUser
  - dataSource

All Arguments:

The following list identifies all of the arguments that can be used with the `plsqlAssemble` command.

- **Database Assembly Arguments:** dataSource, dbConnection, dbUser, jpubProp, sql, sysUser, useDataSource, wsifDbBinding, wsifDbPort

  For more information on these arguments, see "Database Assembly Arguments" on page 18-53.

- **Deployment Descriptor Arguments:** appendToExistingDDs, context, ddFileName, uri

  For more information on these arguments, see "Deployment Descriptor Arguments" on page 18-62.

- **General Assembly Arguments:** appName, className, classpath, debug, ear, emptySoapAction, help, interfaceName, mappingFileName, output, packageName, portName, restSupport, schema, serviceName, useDimeEncoding, war

  For more information on these arguments see, "General Web Services Assembly Arguments" on page 18-41.

- **Java Generation Arguments:** valueTypeClassName

  For more information on this argument, see "Java Generation Arguments" on page 18-68.

- **Message Format Arguments:** style, use

  For more information on these arguments, see "Message Format Arguments" on page 18-67.

- **Session Arguments:** timeout

  For more information on this argument, see "Session Arguments" on page 18-51.

- **WSDL Access Arguments:** httpNonProxyHosts, httpProxyHost, httpProxyPort

  For more information on these arguments, see "WSDL Access Arguments" on page 18-64.

- **WSDL Management Arguments:** createOneWayOperations, genQos, qualifiedElementForm, soapVersion, targetNamespace, typeNamespace

  For more information on these arguments, see "WSDL Management Arguments" on page 18-65.

Ant Task Support:

- `<proxy>` tags. For more information, see "How to Configure Proxy Generation in an Ant Task" on page 18-77.
- `<port>` tags. In the `<port>` tag, `plsqlAssemble` can use these arguments: bindingName, `name` (same as `portName`), portName, sendConnectionFactoryLocation, sendQueueLocation, soapVersion, and uri. For more information, see "How to Configure a Port in an Ant Task" on page 18-78.
- `<handler>` tags. For more information, see "How to Configure Handlers in an Ant Task" on page 18-81.

sqlAssemble

Use the `sqlAssemble` command to generate Web services from SQL statements, including SQL queries and DMLs (Data Manipulation Language). To use this command, you must connect to a database.

See Also:

- "How to Assemble a Web Service from a SQL Query or DML Statement" on page 10-15 provides a full example of generating Web services from SQL statements.
- "How to Establish a Database Connection" on page 18-75 describes the arguments that allow you to do this.

Adding WSIF Bindings to the WSDL

The `sqlAssemble` command can also add WSIF bindings for SQL queries to the WSDL. Use the `wsifDbBinding` argument to add WSIF bindings to the WSDL when you are exposing a database resource as a Web service. You must specify the `className` of the resource's Oracle JPublisher-generated Java class and a database connection.

See Also:

"Configuring a WSIF Endpoint for Database Resources" in the *Oracle Application Server Advanced Web Services Developer's Guide* provides more information on adding bindings for database resources to the WSDL.

The following command line and Ant task examples establish a database connection and run two SQL commands.

Command Line Example:

```
java -jar wsa.jar -sqlAssemble
  -dbUser scott/tiger
  -sqlstatement "getEmp=select ename, sal from emp where empno=:{id NUMBER}"
  -sqlstatement "getEmpBySal=select ename, sal from emp where sal>:{mysal NUMBER}"
  -dbConnection jdbc:oracle:thin:@dsunrde22:1521:sqlj
  -dataSource jdbc/OracleManagedDS
```

Ant Task Example:

```
<oracle:sqlAssemble
   dbUser="scott/tiger"
   dbConnection="jdbc:oracle:thin:@dsunrde22:1521:sqlj"
   dataSource="jdbc/OracleManagedDS"
   appName="query">
       <sqlstatement value="getEmp=select ename, sal from emp where empno=:{id
NUMBER}"/>
       <sqlstatement value="getEmpBySal=select ename, sal from emp where
sal>:{mysal NUMBER}"/>
</oracle:sqlAssemble>
```

Required Arguments:

To use the `sqlAssemble` command, you must specify the following arguments.

- **Database Assembly Arguments:** sqlstatement

  You must also connect to the database. To do this, use one of the following combinations of arguments.

  - dataSource, dbConnection, and dbUser
  - dataSource

All Arguments:

The following list identifies all of the arguments that can be used with the `sqlAssemble` command.

- **Database Assembly Arguments:** dataSource, dbConnection, dbUser, jpubProp, sql, sqlstatement, sysUser, useDataSource, wsifDbBinding, wsifDbPort

  For more information on these arguments, see "Database Assembly Arguments" on page 18-53.

- **Deployment Descriptor Arguments:** appendToExistingDDs, context, ddFileName, uri

  For more information on these arguments, see "Deployment Descriptor Arguments" on page 18-62.

- **General Assembly Arguments:** appName, className, classpath, debug, ear, emptySoapAction, help, interfaceName, mappingFileName, output, packageName, portName, restSupport, schema, serviceName, useDimeEncoding, war

  For more information on these arguments see, "General Web Services Assembly Arguments" on page 18-41.

- **Java Generation Arguments:** valueTypeClassName

  For more information on this argument, see "Java Generation Arguments" on page 18-68.

- **Message Format Arguments:** style, use

  For more information on these arguments, see "Message Format Arguments" on page 18-67.

- **Session Arguments:** timeout

  For more information on this argument, see "Session Arguments" on page 18-51.

- **WSDL Access Arguments:** httpNonProxyHosts, httpProxyHost, httpProxyPort

  For more information on these arguments, see "WSDL Access Arguments" on page 18-64.

- **WSDL Management Arguments:** createOneWayOperations, genQos, qualifiedElementForm, soapVersion, targetNamespace, typeNamespace

  For more information on these arguments, see "WSDL Management Arguments" on page 18-65.

Ant Task Support:

- `<proxy>` tags. For more information, see "How to Configure Proxy Generation in an Ant Task" on page 18-77.
- `<port>` tags. In the `<port>` tag, `sqlAssemble` can use these arguments: bindingName, `name` (same as `portName`), portName, sendConnectionFactoryLocation, sendQueueLocation, soapVersion, and uri. For more information, see "How to Configure a Port in an Ant Task" on page 18-78.
- `<handler>` tags. For more information, see "How to Configure Handlers in an Ant Task" on page 18-81.

topDownAssemble

Use the `topDownAssemble` command to create the required classes and deployment descriptors for a Web service based on a WSDL description. The files can be stored in either an EAR file, a WAR file, or a directory. You must specify a value for either the `input` or `classpath` arguments to allow for the proper loading of the specified implementation class.

This command is typically used with `genInterface` to generate a Web service top down. If these commands are used together to generate a Web service, then they must share the same value for the `unwrapParameters` argument.

Only one service element can be implemented. WebServicesAssembler enables you to generate the artifacts for only one service at a time. If more than one service is described in the WSDL, a command line argument, `serviceName`, enables you to specify the service you want to use.

Note that if there is any conflict between the names of generated classes, then they will be resolved according to JAX-RPC rules as described in section 4.3.12, "Name Collisions" in the JAX-RPC specification. For more information, see "Resolving Name Collisions" on page 18-71.

See Also:

Chapter 6, "Assembling a Web Service from WSDL" provides more information on using the `topDownAssemble` command to generate a Web service based on a WSDL description.

WSDL Limitations

The following list describes the limitations on WSDLs that can be consumed by WebServicesAssembler.

- If the WSDL contains references to multiple port types, then the `topDownAssemble` command must specify a `<porttype>` tag for each port type.
- National Language Support (also known as "NLS" or "Globalization Support") characters that occur in names in the WSDL, such as in the name of a service, port type, operation, binding or port, are not supported. This may also result in errors in the Web Services Test Page.
- In top down Web service development, you cannot use WebServicesAssembler to change the message format. You can do this only by editing the WSDL.
- The WSDL cannot contain multiple message formats. Remove any ports from the WSDL that reference a binding with a message format that you do not want to use.
- WebServicesAssembler cannot consume WSDLs that contain the `xsd:choice` or `xsd:group` XML types. If you want to consume a WSDL that contains these XML types, set the WebServicesAssembler `dataBinding` argument to `false` and code the `SOAPElement` so that the payload conforms to the schema definition in the WSDL file.

The following command line and Ant task create required classes and deployment descriptors for a Web service. It creates an EAR `build/myService.ear`. The target of the `input` argument should contain the implementation classes. These classes will be copied to the generated archive for you.

Command Line Example:

```
java -jar wsa.jar -topDownAssemble
                  -output build
```

```
                -wsdl my.wsdl
                -input myClasses
                -className com.mycompany.HelloImpl
                -appName myService
```

Ant Task Example:

```
<oracle:topDownAssemble output="build"
                wsdl="my.wsdl"
                input="myClasses"
                appName="myService">
        <porttype
          className="com.mycompany.HelloImpl"/>
        </oracle:topDownAssemble>
```

Required Arguments:

To use the `topDownAssemble` command, you must specify the following arguments.

- **General Assembly Arguments:** className; (input and/or classpath)
- **WSDL Management Arguments:** wsdl

All Arguments:

The following list identifies all of the arguments that can be used with the `topDownAssemble` command.

- **Deployment Descriptor Arguments:** appendToExistingDDs, context, ddFileName, uri

  For more information on these arguments, see "Deployment Descriptor Arguments" on page 18-62.

- **General Assembly Arguments:** appName, classFileName, className (required), classpath, debug, ear, emptySoapAction, help, input, interfaceName, mappingFileName, output, packageName, portName, restSupport, searchSchema, serviceName, useDimeEncoding, war

  For more information on these arguments see, "General Web Services Assembly Arguments" on page 18-41.

- **Java Generation Arguments:** dataBinding, mapHeadersToParameters, overwriteBeans, unwrapParameters, valueTypePackagePrefix

  For more information on these arguments, see "Java Generation Arguments" on page 18-68.

- **Message Format Arguments:** mtomSupport

  For more information on this argument, see "Message Format Arguments" on page 18-67.

- **WSDL Access Arguments:** fetchWsdlImports, httpNonProxyHosts, httpProxyHost, httpProxyPort, wsdl (required)

  For more information on these arguments, see "WSDL Access Arguments" on page 18-64.

- **WSDL Management Arguments:** wsdlTimeout

  For more information on this argument, see "WSDL Management Arguments" on page 18-65.

Additional Ant Support:

- `<proxy>` tags. For more information, see "How to Configure Proxy Generation in an Ant Task" on page 18-77.
- `<port>` tags. In the `<port>` tag, `topDownAssemble` can use these arguments: `name` (same as `portName`), portName, and uri. For more information, see "How to Configure a Port in an Ant Task" on page 18-78.
- `<porttype>` tags. For more information, see "How to Configure a Port Type in an Ant Task" on page 18-80.
- `<handler>` tags. For more information, see "How to Configure Handlers in an Ant Task" on page 18-81.

WSDL Management Commands

The following commands perform actions on a WSDL. The `fetchWsdl` and `genQosWsdl` commands are used in top down Web service development to manage the contents and location of the WSDL. The `genWsdl` command is used to generate a WSDL for bottom up Web service development. The `analyze` command can be used at any time to determine whether WebServicesAssembler supports the functionality described in the WSDL.

- analyze—Determines whether WebServicesAssembler supports the functionality described in the WSDL
- fetchWsdl—Copies the WSDL and its imports to an output directory
- genConcreteWsdl—Creates a concrete WSDL by determining the message format from the abstract part of the WSDL
- genQosWsdl—Inserts assertions about Quality of Service (capability assertions) into the WSDL
- genWsdl—Generates a WSDL based on a Java interface

analyze

Use the `analyze` command to confirm whether the WSDL can be processed by this version of the WebServicesAssembler. The `analyze` command determines whether the specified WSDL can be used to generate a proxy or create an interface for top down assembly. The command also checks that the WSDL uses valid XML and whether it complies with the JAX-RPC requirements of OC4J.

This command returns a message if the WSDL cannot be processed.

> **Note:** The `analyze` command does not check conformance to the Web Services Interoperability (WS-I) specification or general interoperability of your WSDL. You may want to use the WS-I Analyzer tool either directly, or from inside Oracle JDeveloper to check conformance. For more information on tools that you can use to check interoperability, see "Ensuring Interoperable Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide*.

The following command line and Ant task examples demonstrate using `analyze` to see if the specified WSDL can be processed.

Command Line Example:

```
java -jar wsa.jar -analyze
                  -wsdl myservice.wsdl
```

Ant Task Example:

```
<oracle:analyze wsdl="myservice.wsdl"
         />
```

Required Arguments:

To use the `analyze` command, you must specify the following argument.

- **WSDL Access Arguments:** wsdl

All Arguments:

The following list identifies all of the arguments that can be used with the `analyze` command.

- **General Assembly Arguments:** debug, help

  For more information on these arguments, see "General Web Services Assembly Arguments" on page 18-41.

- **WSDL Access Arguments:** httpNonProxyHosts, httpProxyHost, httpProxyPort, wsdl (required)

  For more information on this argument, see "WSDL Access Arguments" on page 18-64.

- **WSDL Management Arguments:** wsdlTimeout

  For more information on this argument, see "WSDL Management Arguments" on page 18-65.

Additional Ant Support:

None

fetchWsdl

Use the `fetchWsdl` command in top down Web service generation to copy the base (or top level) WSDL file and all of its imported and included WSDLs and schemas into a specified output directory.

All of the WSDLs and schemas are downloaded into the same directory. Any naming conflicts are resolved by appending a number to the name of the file before the extension. For example, if three `myschema.xsd` files are downloaded, they will be named `myschema.xsd`, `myschema1.xsd`, and `myschema2.xsd`.

The following command line and Ant task examples will fetch the base WSDL specified by the URL, and any other WSDL fragments and schemas it imports. The results are then stored in the `wsdl` directory.

Command Line Example:

```
java -jar wsa.jar -fetchWdsl
                  -wsdl http://someserver/services/aservice?WSDL
                  -output wsdl
```

Ant Task Example:

```
<oracle:fetchWsdl wsdl="http://someserver/services/aservice?WSDL"
```

```
output="wsdl"
/>
```

Required Arguments:

To use the `fetchWsdl` command, you must specify the following argument.

- **WSDL Access Arguments:** wsdl

All Arguments:

The following list identifies all of the arguments that can be used with the `fetchWsdl` command.

- **General Assembly Arguments:** debug, help, output

  For more information on these arguments, see "General Web Services Assembly Arguments" on page 18-41.

- **WSDL Access Arguments:** httpNonProxyHosts, httpProxyHost, httpProxyPort, wsdl (required)

  For more information on these arguments, see "WSDL Access Arguments" on page 18-64.

- **WSDL Management Arguments:** wsdlTimeout

  For more information on this argument, see "WSDL Management Arguments" on page 18-65.

Additional Ant Support:

None

genConcreteWsdl

While an abstract WSDL is enough to define the API for a Web service, WebServicesAssembler needs a concrete WSDL to deploy a Web service or to generate client proxies that can communicate with a Web service.

If you have only an abstract WSDL, use the `genConcreteWsdl` command. In top down Web service development, this command enables you to generate a concrete WSDL given an abstract WSDL. The command does this by analyzing the `wsdl:portType` part of the WSDL and determining whether the bindings for the Web service (that is, the `use` and `style` values) should be document/literal or RPC/literal. The command writes these values into the `binding` element of the WSDL, and saves it with the name determined by the value of the `output` argument.

The following command line and Ant task examples take an abstract WSDL `myAbstract.wsdl` as input and generate a concrete WSDL `myConcrete.wsdl` in the `outputDir` directory.

Command Line Example:

```
java -jar wsa.jar -genConcreteWsdl
                  -output outputDir/myConcrete.wsdl
                  -wsdl myAbstract.wsdl
```

Ant Task Example:

```
<oracle:genConcreteWsdl output="outputDir/myConcrete.wsdl"
                 wsdl="myAbstract.wsdl" />
```

Required Arguments:

To use the `genConcreteWsdl` command, you must specify the following argument.

- **WSDL Access Arguments:** wsdl

All Arguments:

The following list identifies all of the arguments that can be used with the `genConcreteWsdl` command.

- **Deployment Descriptor Arguments:** ddFileName

  For more information on this argument, see "Deployment Descriptor Arguments" on page 18-62.

- **General Assembly Arguments:** debug, help, output

  For more information on these arguments, see "General Web Services Assembly Arguments" on page 18-41.

- **WSDL Access Arguments:** httpNonProxyHosts, httpProxyHost, httpProxyPort, importAbstractWsdl, wsdl (required)

  For more information on these arguments, see "WSDL Access Arguments" on page 18-64.

- **WSDL Management Arguments:** singleService, wsdlTimeout

  For more information on these arguments, see "WSDL Management Arguments" on page 18-65.

Additional Ant Support:

None.

genQosWsdl

In top down Web service development, use the `genQosWsdl` command to add capability assertions for security and reliability into a specified WSDL. Capability assertions are descriptions of Web service management policies that allow consumers of Web services to discover which management policies are enabled for the Web service.

Usually, the capability assertions are defined in the deployment descriptor. Use the `ddFileName` argument to specify the file that contains the capability assertions and the `wsdl` argument to specify the name of the WSDL into which they will be inserted. The `output` argument specifies where the modified WSDL file will be stored. If you do not specify the `output` argument, then the original WSDL will be overwritten.

> **See Also:**
>
> "Working with Capability Assertions" in the *Oracle Application Server Advanced Web Services Developer's Guide* describes how capability assertions are derived and how they are inserted into the WSDL.

The following command line and Ant task examples add assertions to `my.wsdl` and store the results in the `build` directory.

Command Line Example:

```
java -jar wsa.jar -genQosWsdl
                  -wsdl my.wsdl
                  -ddFileName oracle-webservices.xml
```

```
                    -output build
```

Ant Task Example:

```
<oracle:genQosWsdl wsdl="my.wsdl"
            ddFileName="oracle-webservices.xml"
            output="build"
            />
```

Required Arguments:

To use the `genQosWsdl` command, you must specify the following arguments.

- **Deployment Descriptor Arguments:** ddFileName
- **WSDL Management Arguments:** wsdl

All Arguments:

The following list identifies all of the arguments that can be used with the `genQosWsdl` command.

- **Deployment Descriptor Arguments:** ddFileName (required)

  For more information on this argument, see "Deployment Descriptor Arguments" on page 18-62.

- **General Assembly Arguments:** debug, help, output

  For more information on these arguments see, "General Web Services Assembly Arguments" on page 18-41.

- **WSDL Access Arguments:** httpNonProxyHosts, httpProxyHost, httpProxyPort, wsdl (required)

  For more information on these arguments, see "WSDL Access Arguments" on page 18-64.

- **WSDL Management Arguments:** wsdlTimeout

  For more information on this argument, see "WSDL Management Arguments" on page 18-65.

Additional Ant Support:

None

genWsdl

Use the `genWsdl` command to generate a WSDL and a JAX-RPC mapping file for assembling Web services bottom up from a Java interface. This command requires either an `interfaceName` argument or a `className` argument that points to an annotated Java class to provide the WSDL with a value for the service endpoint interface.

The following example illustrates the `genWsdl` command. The `interfaceName` argument specifies the `oracle.j2ee.demo.HelloIntf` interface.

```
java -jar wsa.jar genWSDL -interfaceName oracle.j2ee.demo.HelloIntf -output wsdl
-classpath classes
```

Generating WSDLs with WSIF Bindings

The `genWsdl` command can use the following arguments to add WSIF bindings to the generated WSDL.

- the `wsifJavaBinding` argument adds WSIF bindings to the WSDL when you are exposing a Java class as a Web service. You must also specify the Java class with the `className` argument.
- the `wsifEjbBinding` argument adds WSIF bindings to the WSDL when you are exposing an EJB as a Web service. You must specify the EJB's home interface in the `className` argument and its JNDI name in the `jndiName` argument.
- the `wsifDbBinding` argument adds WSIF bindings to the WSDL when you are exposing a database resource as a Web service. You must specify the `className` of the resource's Oracle JPublisher-generated Java class and a database connection.

You can also use the `genWsdl` command to add WSIF bindings for multiple ports in the WSDL.

See Also:

"Using Web Services Invocation Framework" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on WSIF.

Generating WSDLs for use with J2SE 5.0 Annotations

If you are generating a WSDL from J2SE 5.0 Annotations, then use the `className` argument instead of the `interfaceName` argument. The `className` argument must identify the implementation class and the `@WebService` annotation must be present on the class declaration.

Use the `interfaceName` argument if the specified `className` does not contain any J2SE 5.0 Annotations.

J2SE 5.0 Annotations require a class name because annotations can appear in either the interface or the implementation class. If the annotations appear only in the interface, WebServicesAssembler can obtain them through its referenced implementation class. If the implementation class also contains annotations, WebServicesAssembler will process them.

In the following example, the `genWsdl` command is used to generate a WSDL for use with J2SE 5.0 annotations. The `className` argument specifies the `oracle.j2ee.demo.HelloImpl` class.

```
java -jar wsa.jar genWsdl -className oracle.j2ee.demo.HelloImpl -output wsdl
-classpath classes
```

If you want WebServicesAssembler to process only the J2SE 5.0 annotations in the interface, then enter the `@WebService` annotation with the `serviceEndpointInterface` property in the implementation class file. J2SE 5.0 Annotations will expect all remaining annotations to appear in the service endpoint interface class. For example, if you enter the following annotation in the implementation class file, then WebServicesAssembler will process only the annotations in the `demo.myInterface` interface.

```
@WebService(serviceEndpointInterface="demo.myInterface")
```

The following command line and Ant task output a JAX-RPC mapping file and a WSDL that corresponds to the Java interface specified by `interfaceName`. The results are stored in the `etc` directory.

Command Line Example:

```
java -jar wsa.jar -genWsdl
                  -classpath myservice.jar
```

```
                -output etc
                -interfaceName com.mycompany.myservice.Hello
```

Ant Task Example:

```
<oracle:genWsdl output="etc"
    >
    <oracle:porttype interfaceName="com.mycompany.myservice.Hello"/>
    <oracle:classpath>
        <pathelement path="myservice.jar" />
    </oracle:classpath>
</oracle:genWsdl>
```

Required Arguments:

To use the `genWsdl` command, you must specify the following arguments.

- **General Assembly Arguments:** classpath; when generating a WSDL using J2SE 5.0 Annotations, `genWsdl` requires either interfaceName or a className that points to an annotated Java class

All Arguments:

The following list identifies all of the arguments that can be used with the `genWsdl` command.

- **Database Assembly Arguments:** dataSource, dbConnection, dbUser, wsifDbBinding, wsifDbPort

  For more information on these arguments, see "Database Assembly Arguments" on page 18-53.

- **Deployment Descriptor Arguments:** ddFileName

  For more information on this argument, see "Deployment Descriptor Arguments" on page 18-62.

- **General Assembly Arguments:** bindingName, className (required for J2SE 5.0 Annotations), classpath (required), debug, emptySoapAction, help, initialContextFactory, interfaceName (required for J2SE 5.0 Annotations), jndiName, jndiProviderURL, mappingFileName, output, portName, portTypeName, schema, serviceName, strictJaxrpcValidation

  For more information on these arguments see, "General Web Services Assembly Arguments" on page 18-41.

- **Java Generation Arguments:** valueTypeClassName, wsifEjbBinding, wsifJavaBinding

  For more information on these arguments, see "Java Generation Arguments" on page 18-68.

- **JMS Arguments:** sendConnectionFactoryLocation, sendQueueLocation

  For more information on these arguments, see "JMS Assembly Arguments" on page 18-58.

- **Message Format Arguments:** style, use

  For more information on these arguments, see "Message Format Arguments" on page 18-67.

- **WSDL Management Arguments:** createOneWayOperations, genQos, qualifiedElementForm, soapVersion, targetNamespace, typeNamespace

For more information on these arguments, see "WSDL Management Arguments" on page 18-65.

Additional Ant Support:

- `<port>` tags. In the `<port>` tag, `genWsdl` can use these arguments: bindingName, `name` (same as `portName`), portName, sendConnectionFactoryLocation, sendQueueLocation, soapVersion, and uri. For more information, see "How to Configure a Port in an Ant Task" on page 18-78.
- `<porttype>` tags. For more information, see "How to Configure a Port Type in an Ant Task" on page 18-80.
- tags to specify WSIF bindings for multiple ports. For more information, see "Configuring a WSIF Endpoint for Multiple Java Ports", "Configuring a WSIF Endpoint for Multiple EJB Ports", and "Configuring a WSIF Endpoint for Multiple Database Resource Ports", in the *Oracle Application Server Advanced Web Services Developer's Guide*.

Java Generation Commands

The following commands generate code to create a Java interface, a proxy/stub, or JAX-RPC value type classes.

- genInterface—Generates a Java interface from a WSDL file
- genProxy—Generates a proxy/stub from a WSDL file
- genValueTypes—Generates JAX-RPC value type classes from an XML schema

genInterface

For top down Web service development, this command creates a service endpoint interface for each port type and a Java value type class (beans) for any complex type defined in a WSDL. It also creates a JAX-RPC mapping file that describes the mapping between the XML schema types and the Java value type classes. These files can then be used to construct a J2EE Web service client or to create an implementation that can run on the server.

If you are assembling a J2EE Web service client and want to pass in an Oracle specific client configuration, then use the `ddFileName` argument.

In addition to a JAX-RPC mapping file, the `genInterface` command generates the files listed in Table 18–1 from a WSDL document. The list and descriptions assume that no JAX-RPC mapping file was used to change the default naming conventions of these files. Also note that if there is any conflict between the names of generated Java classes, then they will be resolved according to JAX-RPC rules as described in section 4.3.12, "Name Collisions" in the JAX-RPC specification. For more information, see "Resolving Name Collisions" on page 18-71.

Note: Changing any of these generated files is not recommended because they may be overwritten if a WebServicesAssembler command is invoked again. WebServicesAssembler will not overwrite the files generated with the `genInterface` command for the schema types under these conditions:

- the class name, including the package name, already exists in the classpath argument given to the WebServicesAssembler command, and
- the command line argument `overwriteBeans` is set to false (the default value). See "overwriteBeans" on page 18-69.

Table 18–1 genInterface Generated Files

File Name	Description
<*derived_name*>.java	One file is generated for each type defined in the schemas in the specified WSDL. The name of the Java class file, *derived_name*, is derived from the name of the complex type or element in the schema. The JAX-RPC specification sections 4.2.3, "XML Struct and Complex Type", 4.3.6, "WSDL Fault", and 6.5, "SOAP Fault", describe this class.
<*portTypeName*>.java	The service endpoint interface file. This file contains a Java method for every WSDL operation in the port type. The *portTypeName* is the value of the `<portTypeName>` element in the WSDL document. Section 4.3.3, "WSDL Port Type", in the JAX-RPC 1.1 specification, describes this class.

Note: The `genInterface` command cannot process a WSDL file correctly if it uses multibyte characters in element names. The command will produce class files but they will not compile correctly.

To work around this limitation, set the `dataBinding` argument to `false` when generating the interface. For more information, see "genInterface, genProxy, and topDownAssemble Cannot Process WSDL Files Correctly that Contain Multibyte Characters in Element Names" on page F-12.

See Also:

- Chapter 14, "Assembling a J2EE Web Service Client" for information on how to use `genInterface` to construct a J2EE Web service client.
- Chapter 6, "Assembling a Web Service from WSDL" for information on how to use `genInterface` to create a server-side implementation.
- "Default Algorithms to Map Between Target WSDL Namespaces and Java Package Names" on page 18-72 for information on how WebServicesAssembler performs package-to-namespace mappings.
- The JAX-RPC 1.1 specification provides more information on the mapping which WebServicesAssembler uses to map special characters in the WSDL, such as periods (`.`) and dashes (`-`).

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

The following command line and Ant task examples create a service endpoint interface in the `src` directory (`src/oracle/demo/service`).

Command Line Example:

```
java -jar wsa.jar -genInterface
                  -output src
                  -wsdl myservice.wsdl
                  -packageName oracle.demo.service
```

Ant Task Example:

```
<oracle:genInterface output="src"
              wsdl="myservice.wsdl"
              packageName="oracle.demo.service"
              />
```

Required Arguments:

To use the `genInterface` command, you must specify the following argument.

- **WSDL Access Arguments:** wsdl

All Arguments

The following list identifies all of the arguments that can be used with the `genInterface` command.

- **Deployment Descriptor Arguments:** ddFileName

  For more information on this argument, see "Deployment Descriptor Arguments" on page 18-62.

- **General Assembly Arguments:** classpath, debug, help, mappingFileName, packageName, output, searchSchema, serviceName

  For more information on these arguments see, "General Web Services Assembly Arguments" on page 18-41.

- **Java Generation Arguments:** dataBinding, mapHeadersToParameters, overwriteBeans, unwrapParameters, valueTypePackagePrefix

For more information on these arguments see, "Java Generation Arguments" on page 18-68.

- **WSDL Access Arguments:** httpNonProxyHosts, httpProxyHost, httpProxyPort, wsdl (required)

  For more information on these arguments see, "WSDL Access Arguments" on page 18-64.

- **WSDL Management Arguments:** wsdlTimeout

  For more information on this argument see, "WSDL Management Arguments" on page 18-65.

Additional Ant Support:

None

genProxy

Use the `genProxy` command to create a static proxy stub that can be used by a J2SE Web service client. This command creates all of the Java files required to contact the ports specified in the WSDL. If you have an Oracle specific client configuration, use the `ddFileName` argument to pass it to the command.

You must compile the proxy code before you can use it.

The `genProxy` command generates the files Table 18–2 from a WSDL document. The list and descriptions assume that no JAX-RPC mapping file was used to change the default naming conventions of these files. Also note that if there is any conflict between the names of generated Java classes, then they will be resolved according to JAX-RPC rules as described in section 4.3.12, "Name Collisions" in the JAX-RPC specification. For more information, see "Resolving Name Collisions" on page 18-71.

Note that changing any of these generated files is not recommended because they may be overwritten if the WebServicesAssembler command is invoked again.

Note: WebServicesAssembler will not overwrite the files generated with the `genProxy` command for the schema types under these conditions:

- the class name, including the package name, already exists in the classpath argument given to the WebServicesAssembler command, and
- the command line argument `overwriteBeans` is set to `false` (the default value). See "overwriteBeans" on page 18-69.

Table 18–2 genProxy Generated Files

File Name	Description
<*derived_name*>.java	One file is generated for each type defined in the schemas in the specified WSDL. The name of the Java class file, *derived_name*, is derived from the name of the complex type or element in the schema. The JAX-RPC specification sections 4.2.3, "XML Struct and Complex Type", 4.3.6, "WSDL Fault", and 6.5, "SOAP Fault", describe this class.

Table 18–2 (Cont.) genProxy Generated Files

File Name	Description
<*portName*>Client.java	A utility class file is generated for each port in the WSDL. This file contains code that shows you how to call your service. You should use this file only to test your service; not as the vehicle in which your clients call the service. It is example code. The *portName* is the value of the `<portName>` element in the WSDL document. **Note:** this file is not mandated by the JAX-RPC specification.
<*portTypeName*>.java	The service endpoint interface file. This file contains a Java method for every WSDL operation in the port type. The *portTypeName* is the value of the `<portTypeName>` element in the WSDL document. Section 4.3.3, "WSDL Port Type", in the JAX-RPC 1.1 specification, describes this class.
<*serviceName*>.java	A service interface file with this name is generated for every service in the WSDL document. The *serviceName* is the value of the `<serviceName>` element in the WSDL. Sections 4.3.9, "WSDL Service", and 4.3.10, "Service Interface" describe this class.
Runtime directory/package	These directories contain the files used internally by the Web services proxy code. You should not use any of these files directly because they may change or be removed in a future release.

See Also:

- Chapter 15, "Assembling a J2SE Web Service Client" for more information on using `genProxy` to create proxy stub code.
- "WSDL Limitations" on page 18-22 describes the limitations on WSDLs that can be used as input to the WebServicesAssembler tool.

The following command line and Ant task examples create all of the proxy code and store it in the `proxysrc` directory.

Command Line Example:

```
java -jar wsa.jar -genProxy
                  -output proxysrc
                  -wsdl myservice.wsdl
```

Ant Task Example:

```
<oracle:genProxy output="proxysrc"
          wsdl="myservice.wsdl"
          />
```

Required Arguments:

To use the `genProxy` command, you must specify the following argument.

- **WSDL Access Arguments:** wsdl

All Arguments:

The following list identifies all of the arguments that can be used with the `genProxy` command.

- **Deployment Descriptor Arguments:** ddFileName

For more information on this argument, see "Deployment Descriptor Arguments" on page 18-62.

- **General Assembly Arguments:** classpath, debug, help, mappingFileName, packageName, output, searchSchema, serviceName, useDimeEncoding

  For more information on these arguments see, "General Web Services Assembly Arguments" on page 18-41.

- **Java Generation Arguments:** dataBinding, mapHeadersToParameters, overwriteBeans, unwrapParameters, valueTypePackagePrefix

  For more information on these arguments see, "Java Generation Arguments" on page 18-68.

- **JMS Assembly Arguments:** replyToConnectionFactoryLocation, replyToQueueLocation

  For more information on these arguments see, "JMS Assembly Arguments" on page 18-58.

- **Message Format Arguments:** mtomSupport

  For more information on this argument, see "Message Format Arguments" on page 18-67.

- **Proxy Arguments:** endpointAddress, genJUnitTest

  For more information on these arguments see, "Proxy Arguments" on page 18-61.

- **WSDL Access Arguments:** httpNonProxyHosts, httpProxyHost, httpProxyPort, wsdl (required)

  For more information on these arguments, see "WSDL Access Arguments" on page 18-64.

- **WSDL Management Arguments:** wsdlTimeout

  For more information on this argument see, "WSDL Management Arguments" on page 18-65.

Additional Ant Support:

- `<handler>` tags—For more information, see "How to Configure Handlers in an Ant Task" on page 18-81.
- `<port>` tags—In the `<port>` tag, `genProxy` can use these arguments: endpointAddress, `name` (same as `portName`), portName, replyToConnectionFactoryLocation, and replyToQueueLocation. For more information, see "How to Configure a Port in an Ant Task" on page 18-78.

genValueTypes

Use the `genValueTypes` command to create JAX-RPC value type classes (beans) from the specified schemas. This command creates the beans for schema-driven Web service development.

WebServicesAssembler can create more than one value type class for each `genValueTypes` invocation. The command supports more than one `schema` argument on the command line or `<schema value="">` line in an Ant task.

In addition to the beans, this command creates the `custom-type-mappings.xml` and `jaxrpc-mappings.xml` files. The `custom-type-mappings.xml` file makes it easier to configure the custom serializer. The generated custom type mapping file

conforms to the service side schema, but it can be modified slightly to be used on the client side.

The `jaxrpc-mappings.xml` file is a partial JAX-RPC mapping file that can be supplied when generating the WSDL for bottom up Web service development.

See Also:

- "Editing the Server Side Custom Type Mapping File" in the *Oracle Application Server Advanced Web Services Developer's Guide* for an example of an edited service side custom type mapping file.
- "Custom Serialization of Java Value Types" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on the custom serializer.

The following command line and Ant task examples create a value type (bean) for every complex type defined in the schemas `mySchema.xsd` and `otherSchema.xsd` and creates the files `custom-type-mappings.xml` and `jaxrpc-mappings.xml`. The beans and files are stored in the `build` directory.

Command Line Example:

```
java -jar wsa.jar -genValueTypes
                  -schema mySchema.xsd
                  -schema otherSchema.xsd
                  -packageName com.mycompany
                  -output build
```

Ant Task Example:

```
<oracle:genValueTypes packageName="com.mycompany" output="build">
                <oracle:schema value="otherSchema.xsd"/>
                <oracle:schema value="mySchema.xsd"/>
 </oracle:genValueTypes>
```

Required Arguments:

To use the `genValueTypes` command, you must specify the following argument.

- **General Assembly Arguments:** schema

All Arguments:

The following list identifies all of the arguments that can be used with the `genValueTypes` command.

- **General Assembly Arguments:** debug, help, packageName, output, schema (required)

  For more information on these arguments see, "General Web Services Assembly Arguments" on page 18-41.

- **WSDL Access Arguments:** httpNonProxyHosts, httpProxyHost, httpProxyPort

  For more information on these arguments see, "WSDL Access Arguments" on page 18-64.

Additional Ant Support:

None

Deployment Descriptor Generation Commands

The following commands create deployment descriptors or files that are used in generating descriptors for the EAR.

- genApplicationDescriptor—Creates an `application.xml` file
- genDDs—Creates deployment descriptors

genApplicationDescriptor

Use the `genApplicationDescriptor` command to create an `application.xml` file that can be used when generating an EAR.

The `input` argument must be a directory that contains the WARs and EJB JARs that will be placed in the EAR. The generated `application.xml` will contain tags for each of the WARs and EJB JARs found in the specified `input` directory.

Command Line Example:

```
java -jar wsa.jar -genApplicationDescriptor
                  -input src/ejb
                  -output build
```

Ant Task Example:

```
<oracle:genApplicationDescriptor
        input="src/ejb"
        output="build"
/>
```

Required Arguments:

To use the `genApplicationDescriptor` command, you must specify the following argument.

- **General Assembly Arguments:** input

All Arguments:

The following list identifies all of the arguments that can be used with the `genApplicationDescriptor` command.

- **Deployment Descriptor Arguments:** context

  For more information on this argument, see "Deployment Descriptor Arguments" on page 18-62.

- **General Assembly Arguments:** debug, help, input (required), output

  For more information on these arguments see, "General Web Services Assembly Arguments" on page 18-41.

Additional Ant Support:

None

genDDs

Use the `genDDs` command in top down or bottom up Web service generation to create the `web.xml`, `webservices.xml`, and `oracle-webservices.xml` deployment descriptors.

The following command line and Ant task examples create the deployment descriptors and store them in the `WEB-INF` directory.

Command Line Example:

```
java -jar wsa.jar -genDDs
                  -output WEB-INF
                  -wsdl myservice.wsdl
                  -classpath myservice.jar
                  -interfaceName com.mycompany.myservice.Hello
                  -className com.mycompany.myservice.HelloImpl
```

Ant Task Example:

```
<oracle:genDDs output="WEB-INF"
        wsdl="myservice.wsdl"
        classpath="myservice.jar"
        >
        <oracle:porttype
           interfaceName="com.mycompany.myservice.Hello"
           className="com.mycompany.myservice.HelloImpl"/>
        </oracle:genDDs>
```

Required Arguments

To use the `genDDs` command, you must specify the following arguments.

- **General Assembly Arguments:** className, interfaceName
- **WSDL Access Arguments:** wsdl

All Arguments:

The following list identifies all of the arguments that can be used with the `genDDs` command.

- **Deployment Descriptor Arguments:** appendToExistingDDs, context, ddFileName, uri

  For more information on these arguments, see "Deployment Descriptor Arguments" on page 18-62.

- **General Assembly Arguments:** className (required), classpath, debug, ejbName, help, interfaceName (required), mappingFileName, output, serviceName, strictJaxrpcValidation, useDimeEncoding

  For more information on these arguments see, "General Web Services Assembly Arguments" on page 18-41.

- **JMS Assembly Arguments:** sendConnectionFactoryLocation, sendQueueLocation

  For more information on these arguments see, "JMS Assembly Arguments" on page 18-58.

- **WSDL Access Arguments:** httpNonProxyHosts, httpProxyHost, httpProxyPort, wsdl (required)

  For more information on these arguments see, "WSDL Access Arguments" on page 18-64.

- **WSDL Management Arguments:** wsdlTimeout

  For more information on this argument see, "WSDL Management Arguments" on page 18-65.

Additional Ant Support:

- `<port>` tags. In the `<port>` tag, `genDDs` can use these arguments: `name` (same as `portName`), portName, sendConnectionFactoryLocation, sendQueueLocation, and uri. For more information, see "How to Configure a Port in an Ant Task" on page 18-78.
- `<porttype>` tags. For more information, see "How to Configure a Port Type in an Ant Task" on page 18-80.
- `<handler>` tags. For more information, see "How to Configure Handlers in an Ant Task" on page 18-81.

Maintenance Commands

The following commands return a short description of WebServicesAssembler commands and the version number of the tool.

- help—Returns a list of WebServicesAssembler commands
- version—Returns the version of the WebServicesAssembler tool

help

Use the `help` command to return a list of WebServicesAssembler commands and a brief description. WebServicesAssembler will also return help when it is run:

- without any commands or arguments. For example,

```
java -jar wsa.jar
```

- with incorrect commands or arguments. For example,

```
java -jar wsa.jar -myProxy -ootput proxysrc -wsdl myservice.wsdl
```

You can get help on a specific command by running WebServicesAssembler with the command followed by `-help`. For example, the following command returns detailed help on the `genProxy` command and its required and optional arguments.

```
java -jar wsa.jar -genProxy -help
```

The `help` command is supported only on the command line. There is no comparable Ant task. The following command line example returns help text on the commands and arguments supported by WebServicesAssembler.

Command Line Example:

```
java -jar wsa.jar -help
```

Ant Task Example:

No Ant support provided.

Required Arguments:

None

All Arguments:

The following argument can be used with the `help` command:

- **General Assembly Arguments:** debug

For more information on this argument see, "General Web Services Assembly Arguments" on page 18-41.

Additional Ant Support:

None

version

Use the `version` command to obtain the version number of the WebServicesAssembler tool.

Command Line Example:

```
java -jar wsa.jar -version
```

Ant Task Example:

No Ant support provided.

Required Arguments:

None

All Arguments:

- **General Assembly Arguments:** debug, help

  For more information on these arguments see, "General Web Services Assembly Arguments" on page 18-41.

Additional Ant Support:

None

WebServicesAssembler Arguments

This section describes the arguments that can be called for WebServicesAssembler commands.

- General Web Services Assembly Arguments
- Session Arguments
- CORBA Assembly Arguments
- Database Assembly Arguments
- JMS Assembly Arguments
- Proxy Arguments
- Deployment Descriptor Arguments
- WSDL Management Arguments
- Message Format Arguments
- Java Generation Arguments

General Web Services Assembly Arguments

This section describes common arguments that can be used by many of the WebServicesAssembler commands. They include file-related input and output arguments, WSDL-related arguments, and mapping-related arguments.

- appName
- bindingName
- classFileName
- className
- classpath
- debug
- ear
- ejbName
- emptySoapAction
- help
- initialContextFactory
- input
- interfaceFileName
- interfaceName
- jndiName
- jndiProviderURL
- mappingFileName
- output
- packageName
- portName
- portTypeName
- schema
- searchSchema
- serviceName
- strictJaxrpcValidation
- useDimeEncoding
- war

appName

`appName <String>`

Specifies the name of an application. Usually, this name is used as a base value for other arguments like `context` and `uri`. The value of this argument is also used to name the EAR and the WAR files if the `ear` and `war` arguments are not specified.

bindingName

`bindingName <String>`

The name of the binding to use in the generated WSDL.

classFileName

`classFileName <String>`

Identifies the Java file name of the implementation class specified in the `className` argument. If necessary, this file will be compiled during an assembly.

className

`className <String>`
Specifies the name of the class (including the package name) that is the implementation class for the Web service.

Using this argument adds the `<servlet-class>` element to `web.xml`.

If you are exposing an EJB as a Web service, the value of the `className` argument must be the EJB's home interface.

> **See Also:**
>
> "ejbAssemble" on page 18-14 for more information on using this command with the `className` argument

If this argument is used with the `wsifJavaPort` argument, a `classname` attribute is added to the `java:address` element in the `port` component of the WSDL.

If this argument is used with the `wsifEjbPort` argument, a `classname` attribute is added to the `ejb:address` element in the `port` component of the WSDL.

classpath

`classpath <String>`
Specifies the classpath that contains any user-created classes given to WebServicesAssembler. One reason to specify this argument is if you have already created some value type classes or exceptions and you do not want WebServicesAssembler to overwrite them.

For commands that generate a WAR file, such as the `*Assemble` commands, this argument enables you to point to classes the Web service has a dependency on and that should not be placed in the generated WAR.

This argument does not cause any additional classes to be copied to a generated WAR file. Any classes needed at runtime or deployment time must either be copied to the WAR or EAR manually or made available on the application server classpath using server configuration options.

To include classes in the WAR file, you also have the option of using the `input` argument.

> **See Also:**
>
> "input" on page 18-46 for more information on this argument.

If you list multiple paths on the classpath, separate them with a colon (:) if you are using the UNIX operating system. If you are using the Windows operating system, separate multiple paths with a semicolon (;).

debug

`debug <true|false>`
Indicates whether all detailed diagnostic messages should be printed for a given command. Default value is `false`.

The following command line and Ant task examples will display diagnostic messages on the performance of the `assemble` command.

Command Line Example:

```
java -jar -wsa.jar -assemble -debug true...
```

Ant Task Example:

```
<oracle:assemble debug="true"
    ....
 />
```

ear

`ear <file name>`

Specifies the name and location of the generated EAR. The following example creates the EAR file `myService.ear` in the `dist` directory.

```
-ear dist/myService.ear
```

The following list describes how WebServicesAssembler interprets the value of the `ear` argument.

- If the value of the `ear` argument ends with a file name with the extension "`.ear`" (as in the preceding example), then an EAR file with the specified name is created at the location specified by the `ear` argument. The `output` argument is not used to determine the location of the EAR file.
- In all other cases, the name is assumed to be a directory name. An EAR file is not created. Instead, the contents of the EAR will be written to the indicated directory.
- If you specify a value for the `output` argument and do not provide the `ear` argument, then the EAR file will be given the value of the `appName` argument as its default name. For example, if the value of `output` is `build` and the value of `appName` is `myService`, and `ear` is not specified, then the EAR file will be created as `build/myService.ear`.

> **Note:** If you specify the same directory for the `ear` and the `output` argument, then WebServicesAssembler will return an error.

The `ear` and `war` arguments can be used together in the same command line or Ant task. Table 18–3 describes the behavior of these arguments based on whether they specify a file or a directory.

Table 18–3 Behavior of ear and war Arguments for File and Directory Values

ear and war Argument Combination	Behavior
`ear` *directory1* `war` *directory2*	If the `ear` and `war` arguments are set to different directories, the following behavior will occur: ■ no `.ear` file will be created ■ the contents of the `.ear` file will be generated into *directory1* ■ a `.war` file will be created in *directory1* ■ the contents of the `.war` file will be generated into *directory2*

Table 18–3 (Cont.) Behavior of ear and war Arguments for File and Directory Values

ear and war Argument Combination	Behavior
`ear` *directory1* `war` *file*	If the `ear` argument is set to a directory and the `war` argument is set to a file, the following behavior will occur: ■ no `.ear` file will be created ■ the contents of the `.ear` file will be generated into *directory1* ■ a `.war` file will be created in *directory1* ■ a `.war` file will be created with the user-specified name (the value of the *file* parameter)
`ear` *file* `war` *directory2*	If the `ear` argument is set to a file and the `war` argument is set to a directory, the following behavior will occur: ■ a `.ear` file will be created with the user-specified name (value of the *file* parameter) ■ a `.war` file will be created in the `.ear` file ■ the contents of the `.war` will be generated into *directory2*
`ear` *file1* `war` *file2*	If the `ear` argument is set to a file and the `war` argument is set to a file, the following behavior will occur: ■ a `.ear` file will be created with the user-specified name (value of the *file1* parameter) ■ a `.war` file will be created in the `.ear` file ■ a `.war` file will be created with the user-specified name (the value of the *file2* parameter)
`ear` argument not specified `war` argument not specified	If neither the `ear` nor the `war` argument are specified, the following behavior will occur: ■ a `.ear` file will be generated with the default name *application_name*`.ear` ■ a `.war` file will be generated with the default name *application_name*`_web.war` In these examples, *application_name* represents the value of the `appName` argument.

ejbName

`ejbName <`*String*`>`

The name of the EJB to be exposed as a Web service. Note that this is not a class name; it is the unique name of the EJB that is specified in the `<ejb-name>` tag in the `ejb-jar.xml` file.

If the EJB is version 2.1, using this argument adds the `<ejb-link>` element to `webservices.xml`.

emptySoapAction

`emptySoapAction <true|false>`

If `true`, the value of the `soapAction` attribute for each SOAP binding operation in the generated WSDL will be set to an empty string. If `false` (default), the target namespace and the operation name (`<`*target-namespace*`>/<`*operation-name*`>`) is used as the value of the `soapAction` attribute.

Tools from other vendors that use OracleAS Web Services WSDLs to generate the client-side proxy may not be able to recognize or honor the default `soapAction` value. Setting this argument to `true` increases the chances of interoperability with these tools.

help

`help`
Displays a help message for a given command. The `help` argument can either precede or follow the command.

The following command line and Ant task examples will display a help message on the `assemble` command.

Command Line Example:

```
java -jar -wsa.jar -assemble -help ...
```

or

```
java -jar -wsa.jar -help -assemble ...
```

Ant Task Example:

```
<oracle:assemble debug="help"
    ....
 />
```

initialContextFactory

`initialContextFactory <`*`String`*`>`
Specifies the name of the factory that will provide the initial context. This is an optional argument that can be called by the `genWsdl` or `ejbAssemble` command when configuring an EJB WSIF port with `wsifEjbBinding` or `wsifEjbPort`. If you do not provide a value for this attribute, the value in the `jndi.properties` file will be used.

input

`input <`*`String`*`>`
Specifies the directory or JAR containing the classes that should be copied to `WEB-INF/classes`. This argument will be added to the classpath used by the WebServicesAssembler. The `ejbAssemble` command assumes the value of the input is an EJB JAR file or a directory that contains the contents of an EJB JAR.

If this argument specifies a JAR, then it will be expanded and its contents copied to the `WEB-INF/classes` directory.

The `input` argument can be used more than once on the command line or in an Ant task. In an Ant task, write the multiple occurrences in separate tags and include the `value` attribute. Example 18–1 illustrates multiple instances of `input` in an Ant task.

Example 18–1 Multiple Instances of input in an Ant Task

```
<oracle:jmsAssemble
       linkReceiveWithReplyTo="true"
       targetNamespace="http://oracle.j2ee.ws/jms-doc"
       typeNamespace="http://oracle.j2ee.ws/jms-doc/types"
       serviceName="JmsService"
       appName="jms_service"
       context="jms_service"
       input="./demo/build/mdb_service.jar"
       output="./demo/dist"
       input="first.jar"
       >
             <oracle:input value="second.jar"/>
             <oracle:input value="third.jar"/>
</oracle:jmsAssemble>
```

interfaceFileName

`interfaceFileName <String>`
Specifies the path and name of the service endpoint interface (SEI) Java source code file. If the specified file name cannot be found, WebServicesAssembler will stop processing.

The presence of the `interfaceFileName` argument can play a role in how WebServicesAssembler represents the parameter names of Java methods in the generated WSDL.

> **See Also:**
>
> "How to Represent Java Method Parameter Names in the WSDL" on page 18-87 for more information on this topic.

interfaceName

`interfaceName <String>`
Specifies the name of a Java class (including the package name) that contains the service endpoint interface (SEI).

Using this argument adds a `<service-endpoint-interface>String</service-endpoint-interface>` element to `WEB-INF/webservices.xml`, where *`String`* is the value provided for `interfaceName`.

Using this argument with any of the commands which assemble a Web service from database resources (`plsqlAssemble`, `sqlAssemble`, `dbJavaAssemble`, or `aqAssemble`), adds the `<service-endpoint-interface>String</service-endpoint-interface>` element to the *`serviceName`*`-java-wsdl-mapping.xml` file, where *`String`* is the value provided for `interfaceName` and *`serviceName`* is the value of the `serviceName` argument.

jndiName

`jndiName <String>`
Specifies a JNDI name for an EJB.

jndiProviderURL

`jndiProviderURL <String>`
Specifies the URL for the JNDI Provider. This is an optional argument that can be called by the `genWsdl` or `ejbAssemble` command when configuring an EJB WSIF port with `wsifEjbBinding` or `wsifEjbPort`. If you do not provide a value for this attribute, the value in the `jndi.properties` file will be used.

mappingFileName

`mappingFileName <String>`
Specifies a file location that points to a JAX-RPC mapping file. If the specified file name cannot be found, WebServicesAssembler will stop processing.

Using this argument adds the `<jaxrpc-mapping-file>` element to `webservices.xml`. Note that the location and name of the file may be changed when it is written to the deployment descriptor. The contents of the file may be modified before being placed in the archive if some mappings were not defined in the original file.

See Also:

"JAX-RPC Mapping File Descriptor" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on the contents of the JAX-RPC mapping file and how it is used.

output

`output <String>`

Specifies the directory where generated files will be stored. If the directory does not exist, it will be created. If you do not specify the `output` argument, then the output will be stored in the current directory.

Except when used with the `genConcreteWsdl` command, the target of the `output` argument is always assumed to be a directory. In the case of the `genConcreteWsdl` command, the target is assumed to be a file.

The `output` argument is typically used with the `ear` and `war` arguments.

See Also:

"ear" on page 18-44 and "war" on page 18-51 for more information on the behavior of `output` when it is used with these arguments.

Note: If you specify the same directory for the `output` and the `ear` (or `war`) argument, then WebServicesAssembler will return an error.

---

packageName

`packageName <String>`

Specifies the package name that will be used for generated classes if no package name is declared in the JAX-RPC mapping file. If `packageName` is not specified and not declared in the mapping file, then the package name will be derived from the target namespace of the WSDL.

See Also:

"Default Algorithms to Map Between Target WSDL Namespaces and Java Package Names" on page 18-72 for more information on namespace to package mappings.

Note: The `packageName` argument affects only the package name for the 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. For information on generating the code into a single package, see "Generating Code into a Single Package from a WSDL with Multiple Namespaces" in the *Oracle Application Server Advanced Web Services Developer's Guide*.

---

portName

`portName <String>`

Specifies the name of a port in a WSDL document. This argument populates the `<port name="...">` WSDL element.

If a port name is not specified, then the default will be based on either the transport and SOAP version, or on the WSIF type. The default port name will be one of the following values.

- `HttpSoap11`
- `HttpSoap12`
- `JmsSoap11`
- `JmsSoap12`
- `WsifEjb`
- `WsifJava`

portTypeName

`portTypeName <String>`
Specifies the name of the port type to use in the generated WSDL. This argument populates the `<portType name="...">` WSDL element.

restSupport

`restSupport <true|false>`
Specifies whether REST (Representational State Transfer) support will be enabled for this Web service. The default value is `false`.

REST services allow you to get and modify service objects with the HTTP GET and POST commands. Another feature of REST Web services is the use of XML documents, not SOAP envelopes, for sending messages.

> **See Also:**
>
> Chapter 12, "Assembling REST Web Services" for more information on implementing Web services with REST support.

schema

`schema <String>`
Specifies the relative path or URL to an XML schema document. This argument enables you to specify a schema for value types instead of letting WebServicesAssembler generate them. This argument must be used in conjunction with a JAX-RPC mapping file.

> **See Also:**
>
> "mappingFileName" on page 18-47 for information on specifying a mapping file.

The `schema` argument can be used more than once on the command line or in an Ant task. In an Ant task, write the multiple occurrences in separate tags and include the `value` attribute. Example 18–2 illustrates multiple instances of `schema` in an Ant task.

Example 18–2 Multiple Instances of schema in an Ant Task

```
<oracle:genValueTypes packageName="com.mycompany"
              output="build">
                   <oracle:schema value="otherSchema.xsd"/>
                   <oracle:schema value="mySchema.xsd"/>
 </oracle:genValueTypes>
```

searchSchema

`searchSchema <true|false>`
Indicates whether all of the types in the schemas listed in the WSDL should be processed. When this argument is set to `true` (default), all of the types will be processed. This is useful when using a Web service implementation that uses types found in the schemas, but are not directly referenced by the WSDL. When this argument is set to `false`, the types that are not referenced by the WSDL will not be processed.

serviceName

`serviceName <`*`String`*`>`
Specifies the service name. The `serviceName` argument is used to identify the generated or packaged WSDL file (*serviceName*`.wsdl`) and the mapping file (*serviceName*`-java-wsdl-mapping.xml`). In bottom up Web service assembly, this argument also provides a value for the `<service name="...">` WSDL element.

When this argument is used in top down Web service and proxy assembly, the WSDL will expect to find a service with this name. In this case, the value of the `<service name="...">` WSDL element will not be changed.

strictJaxrpcValidation

`strictJaxrpcValidation <true|false>`
Determines whether the service endpoint interface, exceptions, and value types will be validated according to all of the JAX-RPC validation rules. If this argument is not specified or set to `true` (default), the following validation checks are performed:

- the service endpoint interface implements `java.rmi.Remote`
- all methods throw `java.rmi.RemoteException`
- all value types and properties in exceptions follow JAX-RPC rules

If any of these validation checks fail, then processing stops and an error is thrown describing what is wrong with the interface.

If this argument is set to `false`, then a service endpoint interface does not have to implement `java.rmi.Remote` and methods do not have to throw `java.rmi.RemoteException`. However, if any method has parameters or exceptions that do not follow JAX-RPC rules, then the method will be ignored and will not be processed.

In the case of Beans and exceptions, if this argument is set to false, any invalid property in the Bean (or exception) will be ignored. Valid properties will be processed and included in the WSDL. Note that this behavior might produce a wire-level format that the user is not expecting. For invalid properties, WebServicesAssembler will throw a warning.

useDimeEncoding

`useDimeEncoding <true|false>`
Specifies whether DIME encoding will be used for streaming SOAP messages with attachments over the wire. If `useDimeEncoding` is set to `true`, DIME encoding is used for attachments instead of MIME. The default value is `false`.

Using this argument adds the `<use-dime-encoding>` element to `oracle-webservices.xml`.

Note: 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 Also:

"Working with DIME Attachments" in the *Oracle Application Server Advanced Web Services Developer's Guide* provides more information about working with DIME encoding and attachments.

war

`war <file name or directory name>`

Specifies the name of the WAR to generate. If a file is specified, it must have the `.war` extension. If a directory is specified, the contents of the WAR will be written to the indicated directory. The following example creates the WAR file `myService.war` in the `dist` directory.

```
-war dist/myService.war
```

The following list describes how WebServicesAssembler interprets the value of the `war` argument.

- If the value of the `war` argument ends with a file name with the extension "`.war`" (see the preceding example), then a WAR file with the specified name is created at the location specified by the `war` argument. The `output` argument is not used to determine the location of the WAR file.
- In all other cases, the value is assumed to be a directory name. The WAR file will be written to the indicated directory.
- If you specify a value for the `output` argument and do not provide the `war` argument, then the WAR will be placed inside of an EAR. The name of the WAR file inside of the EAR will be *`appName`*`.war`.

Note: If you specify the same directory for the `war` and the `output` argument, then WebServicesAssembler will return an error.

See Also:

- "ear" on page 18-44 for a description of how WebServicesAssembler interprets values for the `ear` argument.
- Table 18–3 on page 18-44 for a description of the behavior of the `war` and `ear` arguments when they are used together in the same command line or Ant task.

Session Arguments

These arguments can be used to control the behavior of session state.

- callScope
- recoverable
- session

- timeout

callScope

`callScope <true|false>`
Indicates that the servant is to be created for each call and is garbage-collected after each call. Default value is `false`. An error will be thrown if both `callScope` and `session` are set to `true`.

Using this argument with the value set to `true` adds the `<param name="scope">call</param>` element to `oracle-webservices.xml`.

recoverable

`recoverable <true|false>`
Indicates whether applications with session state are recoverable. This argument can be used only when the service is exposed as a stateful Web service with session scope. The default, `true`, causes session state to be preserved. If it is recoverable, then a boolean `<distributable>` element is added to `web.xml`. Recoverable means that you want the service to be able to recover in a distributed environment if the node that you are interacting with goes down. This means that the Web service state must be also be distributable.

If `recoverable` is `false`, then the `<distributable>` element will not be added to `web.xml`.

session

`session <true|false>`
Indicates that the servant instance is to be preserved for the duration of the HTTP session. This argument is valid only for HTTP transport. Session timeout can be tuned by the `timeout` argument. Default value is `false`. If `timeout` is set, then `session` is set to `true` by default. An error will be thrown if both `callScope` and `session` are set to `true`.

Using this argument adds the `<param name="scope">session</param>` element to `oracle-webservices.xml`.

timeout

`timeout <int>`
Specifies the number of seconds a session should last before it times out. The default value is 60 seconds. If the value is 0 or a negative number, then the session will not time out.

If a value is set for this argument, then the `session` argument is automatically set to `true`. If the `session` argument is `true` and `timeout` is not set, then the session will time out after 60 seconds.

Using this argument adds the `<param name="session-timeout">`*`value`*`</param>` element to `oracle-webservices.xml`, where *`value`* is the integer value given to `timeout`.

CORBA Assembly Arguments

The following arguments can be used by the `corbaAssemble` command. They can be used to control how a Web service is assembled using CORBA servant objects.

- corbanameURL
- corbaObjectPath

- idlFile
- idlInterfaceName
- idljPath
- ORBInitialHost
- ORBInitialPort
- ORBInitRef

corbanameURL

`corbanameURL <String>`
Specifies a URL for locating the CORBA object using a CORBA Naming Service.

corbaObjectPath

`corbaObjectPath <String>`
Specifies a path to the CORBA object.

idlFile

`idlFile <String>`
Specifies the location of the CORBA `idl` file.

idlInterfaceName

`idlInterfaceName <String>`
Specifies the name of the interface in the `idl` from which the Web service will be generated.

idljPath

`idljPath <String>`
Specifies the path to the directory containing the IDL-to-Java compiler (`idlj`) if it is not already specified in the `path`.

ORBInitialHost

`ORBInitialHost <String>`
Specifies the host name of the ORB.

ORBInitialPort

`ORBInitialPort <String>`
Specifies the port for the ORB.

ORBInitRef

`ORBInitRef <String>`
Specifies an ORB initial reference.

Database Assembly Arguments

These arguments are used by the commands that assemble a Web service from database artifacts such as PL/SQL stored procedures, SQL statements, Oracle Advanced Queues (AQ), and Java classes inside the Java VM in an Oracle database.

See Also:

Chapter 10, "Assembling Database Web Services" for more information on assembling Web services from database artifacts.

- aqConnectionFactoryLocation
- aqConnectionLocation
- dataSource
- dbConnection
- dbJavaClassName
- dbUser
- jpubProp
- sql
- sqlstatement
- sqlTimeout
- sysUser
- useDataSource
- wsifDbBinding
- wsifDbPort

aqConnectionFactoryLocation

`aqConnectionFactory <String>`
Specifies the JNDI location of the Oracle Streams AQ JMS queue connection factory for the exposed AQ.

aqConnectionLocation

`aqConnectionLocation <String>`
Specifies the JDNI location of the Oracle Streams AQ JMS queue connection connecting to the exposed AQ.

dataSource

`dataSource <String>`
Specifies the JNDI location of the data source used by the Web services at runtime.

Using this argument adds the following `param name` attribute to the `<implementor>` element in `oracle-webservices.xml`. The *`dataSource value`* variable is the path specified with the `dataSource` argument.

```
<param name="databaseJndiName">dataSource value</param>
```

For example, if you specify `dataSource="ws/dbws/src/query/datasource"` in an Ant task, the `<implementor>` element will be written as follows.

```
<implementor type="database">
     <param name="databaseJndiName">ws/dbws/src/query/datasource</param>
</implementor>
```

dbConnection

`dbConnection <String>`

Specifies the JDBC URL for the database. This argument is used with the `dbUser` argument to connect to a database at code-generation time.

dbJavaClassName

`dbJavaClassname <String>`
Specifies the name of the server-side Java class to be published as a Web service.

dbUser

`dbUser <String>`
Specifies the database schema and password in the form of *user/password*. This argument is used with the `dbConnection` argument to connect to a database at code-generation time.

If you attempt to access the database by invoking WebServicesAssembler on the command line and do not specify a password with `-dbUser`, you will be prompted for a password. This is to protect the password being shown in clear text.

jpubProp

`jpubProp <String>`
Specifies the name of a file with properties to control the Oracle JPublisher translation process.

See Also:

- "Changing the SQL to XML Mapping for Numeric Types" on page 10-6 for examples of how you might use the `jpubProp` argument.
- *Oracle Database JPublisher User's Guide* for more information on Oracle JPublisher.

sql

`sql <String>`
The value of the `sql` argument has a different meaning depending on whether it is used with `aqAssemble` or `plsqlAssemble`.

- `aqAssemble`—When used with `aqAssemble`, the value of `sql` is an AQ queue name.
- `plsqlAssemble`—When used with `plsqlAssemble`, the value of `sql` is a PL/SQL package name.

The `sql` argument can handle case-sensitive names in SQL statements. For example, the use of quotes in the following SQL statement indicates that the package name, `SIMple`, is case sensitive.

```
create package "SIMple" as
procedure foo;
end;
```

To use the `SIMple` package name as the target of the `sql` argument on the command line, enter:

```
-sql '"SIMple"'
```

To use it in an Ant task, enter:

```
sql="&quot;SIMple&quot;"
```

Note that in each case, the quotations enclosing `SIMple` are required. The quotations and the package name are passed to Oracle JPublisher. Oracle JPublisher uses the quotations to resolve the identifier against the name in the database.

sqlstatement

`sqlstatement <String>`
Specifies the DML statement or SQL query to be published as a Web service.

- SQL statements must be either queries or DML statements.
- SQL statements are tagged with the name that will be used in the WSDL and in generated Java code.

  `methodName=<statement>`

  The *`methodName`* indicates the operation name for the SQL statement in the generated Web services.
- SQL statements may contain embedded host variables with a parameter name and corresponding SQL type. The parameter name is used in the WSDL and in the generated Web service operations. The parameter is given in the following form.

  `:{`*`param_name param_sql_type`*`}`

  In this example, *`param_name`* defines the identifier for the parameter in the generated Web service, and *`param_sql_type`* defines the SQL type name for that parameter.
- The `sqlstatement` argument cannot be used to specify stored procedures.

When you pass a SQL statement on the WebServicesAssembler Ant task, you must ensure that it is correctly quoted. The value of `sqlstatement` can be quoted using standard XML quotations as shown in Table 18–4:

Table 18–4 Valid Quoting Symbols for the sqlstatement Ant Task

Symbol	Quotation
&	&
"	"
<	<
>	>
'	'

Multiple `sqlstatement` arguments can be specified on the command line or Ant task. In an Ant task, write the multiple occurrences in separate tags and use the `value` attribute. Example 18–3 illustrates multiple instances of the `sqlstatement` argument in an Ant task.

Example 18–3 Multiple Instances of sqlstatement in an Ant Task

```
<oracle:sqlAssemble
   dbUser="scott/tiger"
   dbConnection="jdbc:oracle:thin:@dsunrde22:1521:sqlj"
   dataSource="jdbc/OracleManagedDS"
   appName="query">
       <sqlstatement value="getEmp=select ename, sal from emp where empno=:{id
NUMBER}"/>
       <sqlstatement value="getEmpBySal=select ename, sal from emp where
sal&gt;:{mysal NUMBER}"/>
```

```
</oracle:sqlAssemble>
```

sqlTimeout

```
sqlTimeout <int>
```

Specifies timeout, in seconds, for the database operation. Database operations can include PL/SQL stored function or procedure invocations, SQL queries, and DML statements. The default is 0, which means the service never times out.

sysUser

```
sysUser <String>
```

Specifies the name and password of a user with `SYS` privileges in the form of `dbSysUser/syspassword`. Using this argument allows PL/SQL and Java wrapper code to be installed automatically into the database at code-generation time.

useDataSource

```
useDataSource <true|false>
```

The `useDataSource` argument can be specified on the command line or in the Ant task only when the `wsifDbBinding` or `wsifDbPort` argument is also present. If `useDataSource` is specified and neither of these arguments are present, then an error is thrown.

The `useDataSource` argument determines whether the value of the `dataSource` argument will be used in the WSDL port tag. If `useDataSource` is not specified, or specified with the value of `true`, then the `dataSource` value used in the command line or Ant task is used in the WSDL port tag. If `useDataSource` is specified with the value of `false`, then the `dbConnection` value used in the command line or Ant task is used in the WSDL port tag.

wsifDbBinding

```
wsifDbBinding <true|false>
```

This argument is used in bottom up Web services assembly to add WSIF SQL bindings to the WSDL. WebServicesAssembler will generate native WSIF SQL bindings in addition to SOAP bindings in the WSDL. This argument's default value is `false`.

See Also:

- "WSIF SQL Extensions to the WSDL" in the *Oracle Application Server Advanced Web Services Developer's Guide* for information on the extensions the `wsifDbBinding` argument adds to the WSDL.
- "Configuring a WSIF Endpoint for Multiple Database Resource Ports" in the *Oracle Application Server Advanced Web Services Developer's Guide* for information on arguments that allow you to more closely control the definition of the WSIF SQL bindings or to specify multiple ports.

wsifDbPort

```
wsifDbPort <true|false>
```

This argument is used in bottom up Web services assembly to add WSIF bindings for database resources for multiple ports to the WSDL. It can be used only in Ant tasks.

See Also:

"Configuring a WSIF Endpoint for Multiple Database Resource Ports" in the *Oracle Application Server Advanced Web Services Developer's Guide* provides more information on how to use `wisfDbPort`.

JMS Assembly Arguments

The following arguments can be used by the `jmsAssemble` command to assemble a JMS Endpoint Web service EAR or WAR directory.

See Also:

- Chapter 9, "Assembling Web Services with JMS Destinations" for more information assembling a JMS Endpoint Web service.
- The JMS specification at the following Web site:

  `http://java.sun.com/products/jms/docs.html`

Of the following arguments, `replyToConnectionFactoryLocation`, `replyToQueueLocation`, `sendConnectionFactoryLocation`, and `sendQueueLocation` can be used by JMS transport as well as by JMS assembly.

- deliveryMode
- genJmsPropertyHeader
- jmsTypeHeader
- linkReceiveWithReplyTo
- payloadBindingClassName
- priority
- receiveConnectionFactoryLocation
- receiveQueueLocation
- receiveTimeout
- receiveTopicLocation
- replyToConnectionFactoryLocation
- replyToQueueLocation
- replyToTopicLocation
- sendConnectionFactoryLocation
- sendQueueLocation
- sendTopicLocation
- timeToLive
- topicDurableSubscriptionName

deliveryMode

`deliveryMode <String>`

Specifies the default value of the `JMSDeliveryMode` header field. This field contains the delivery mode specified when the message was sent. The values for `deliveryMode` supported by the JMS specification are `PERSISTENT` and `NON_PERSISTENT`.

genJmsPropertyHeader

`genJmsPropertyHeader <true|false>`
By default, JMS Endpoint Web services always publish the following JMS message properties of SOAP headers defined on the generated WSDL.

- message ID
- correlation ID
- the ReplyTo destination, including its name, type, and factory

The corresponding schema definition of the JMS-specific header and its binding will be included in the WSDL. Set this argument to `false` and the `OracleJmsProperties` on the wire SOAP message will be ignored at runtime. Default value is `true`.

jmsTypeHeader

`jmsTypeHeader <String>`
The value of this argument is used to specify the default value of the `JMSType` header field for the JMS messages that are propagated by the `send` operation to a JMS destination. For example, a possible value for `jmsTypeHeader` could be `My Quote Message`. No default value is provided.

linkReceiveWithReplyTo

`linkReceiveWithReply <true|false>`
Determines whether the `receive` operation will be linked to the reply-to destination. When this argument is `true`, either receive destination/connection-factory or reply-to destination/connection-factory must be set. Default value is `false`.

payloadBindingClassName

`payloadBindingClassname <String>`
The fully-qualified Java class name of the content object of `javax.jms.ObjectMessage` instance if no data binding is used. The content is not a Java value type but an XML fragment. The only valid values are `java.lang.String` and `javax.xml.soap.SOAPElement`. Default value is `java.lang.String`.

priority

`priority <int>`
Specifies the default integer value of the JMS message priority header field for the JMS messages that are propagated by the `send` operation to a JMS destination. Values can range from 0 to 9, with 0 as lowest priority and 9 as highest.

receiveConnectionFactoryLocation

`receiveConnectionFactoryLocation <String>`
JNDI name of the JMS `ConnectionFactory` used for the `receive` operation. The type of `ConnectionFactory` must be consistent with the `receive` destination. No default value is provided.

receiveQueueLocation

`receiveQueueLocation <String>`
JNDI name of the JMS Queue used for the `receive` operation. The `receiveQueueLocation` and `receiveTopicLocation` arguments are mutually exclusive and cannot both be set. No default value is provided.

receiveTimeout

`receiveTimeout <int>`
The value of this argument is used to specify the default value for the amount of time that the `receive` method is blocked to get a message. The `receiveTimeout` value is expressed in seconds. The default value, 0, means that blocking never expires and the call blocks indefinitely.

receiveTopicLocation

`receiveTopicLocation <String>`
JNDI name of the JMS Topic used for the `receive` operation. No default value is provided. The `receiveQueueLocation` and `receiveTopicLocation` arguments are mutually exclusive and cannot both be set.

replyToConnectionFactoryLocation

`replyToConnectionFactoryLocation <String>`
Specifies the JNDI name of the JMS connection factory to be used as the default reply-to of all `send` operation JMS messages. The type of `ConnectionFactory` must be consistent with the reply-to destination. No default value is provided. This argument can be used for JMS Web service assembly or for JMS transport.

When this argument is used for JMS transport, it specifies the JNDI name of the JMS `ConnectionFactory` to be used as the default reply-to of all Web service operations that send SOAP messages over the wire.

replyToQueueLocation

`replyToQueueLocation <String>`
Specifies the JNDI name of the JMS queue to be used as the default reply-to of all `send` operation JMS messages. No default value is provided. The `replyToQueueLocation` and `replyToTopicLocation` arguments are mutually exclusive and cannot both be set. This argument can be used for JMS Web service assembly or for JMS transport.

When this argument is used for JMS transport, it specifies the JNDI name of the JMS queue to be used as the default reply-to of all Web service operations that send SOAP messages over the wire.

replyToTopicLocation

`replyToTopicLocation <String>`
Specifies the JNDI name of the JMS Topic to be used as the default reply-to of all `send` operation JMS messages. No default value is provided. The `replyToQueueLocation` and `replyToTopicLocation` arguments are mutually exclusive and cannot both be set.

sendConnectionFactoryLocation

`sendConnectionFactoryLocation <String>`
Specifies the JNDI name of the JMS `ConnectionFactory` used to obtain a connection for the JMS `send` operation. The type of `ConnectionFactory` must be consistent with the `send` destination. No default value is provided. This argument can be used for JMS Web service assembly or for JMS transport.

When this argument is used for JMS transport, it specifies the JNDI name of the JMS `ConnectionFactory` used to obtain a connection for the Web service operation that sends the SOAP messages over the wire.

sendQueueLocation

`sendQueueLocation <String>`
Specifies the JNDI name of the JMS queue to be used for the JMS `send` operation. No default value is provided. The `sendQueueLocation` and `sendTopicLocation` arguments are mutually exclusive and cannot both be set. This argument can be used for JMS Web service assembly or for JMS transport.

When this argument is used for JMS transport, it specifies the JMS queue to be used for the Web service operation that sends the SOAP messages over the wire.

sendTopicLocation

`sendTopicLocation <String>`
JNDI name of JMS Topic used for `send` operation. No default value is provided. The `sendQueueLocation` and `sendTopicLocation` arguments are mutually exclusive and cannot both be set.

timeToLive

`timeToLive <int>`
The value of this argument specifies the default time-to-live value of a JMS `send` operation. The value of `timeToLive`, expressed in seconds, is a relative value—not absolute. When a message is sent, its expiration time is calculated as the sum of the time-to-live value specified on the JMS `send` method and the current GMT value. If the argument is not set or if it is set to 0, then the message will never expire.

topicDurableSubscriptionName

`topicDurableSubscriptionName <String>`
The value of this argument is used as a unique identity to register a durable subscription for the Topic `receive` operation. No default value is provided.

See Also:

The JMS specification for more information on durable subscriptions:

`http://java.sun.com/products/jms/docs.html`

Proxy Arguments

These arguments can be used to control the contents of the generated proxy code.

- endpointAddress
- genJUnitTest

endpointAddress

`endpointAddress <URL>`
Specifies the URL of the endpoint address used to contact the Web service. If you use this argument, then the endpoint address in the WSDL is ignored.

genJUnitTest

`genJUnitTest <true|false>`
If `genJUnitTest` is set to `true`, then a JUnit test will be created for each exposed method in the Web service and stored in the same directory as the service endpoint interface. The JUnit test which is created is only a template; you must provide the actual test code for the method. The default value is `false`.

Deployment Descriptor Arguments

The following arguments can be used to control how Web service deployment is performed.

- appendToExistingDDs
- context
- ddFileName
- uri

appendToExistingDDs

`appendToExistingDDs <true|false>`

Determines whether entries for a new Web service will be appended to the existing deployment descriptors or whether they will be overwritten. The deployment descriptors that can be affected by this argument are `oracle-webservices.xml`, `web.xml`, and `webservices.xml`.

If the argument is set to `true`, any deployment descriptors in the output location will be updated with the entries needed for a new service. If `false`, any existing deployment descriptors in the output location will be overwritten. Default value is `false`.

See Also:

"How to Assign Multiple Web Services to an EAR or WAR Archive" on page 18-85 for information on how to use `appendToExistingDDs` to append a new Web service to an existing deployment descriptor.

Note: The `appendToExistingDDs` and the `ddFileName` argument can be used in the same invocation or Ant task. The `ddFileName` argument points to a file that contains management configuration. All of the management configuration found in this file that corresponds to the service that is generating deployment descriptors will be included in the generated `oracle-webservices.xml` file. When `appendToExistingDDs` is `false` (default), then any deployment descriptor file (`web.xml`, `oracle-webservices.xml`, or `webservices.xml`) in the output directory will be overwritten. When `appendToExistingDDs` is `true`, WebServicesAssembler will add the entries for the current service to the existing deployment descriptor file.

Note that WebServicesAssembler does not try to resolve conflicts. If you append the same service to a deployment descriptor file twice, then an invalid deployment descriptor may be the result.

---

context

`context <String>`

Specifies the root context for the Web application. You cannot specify an empty string for `context`. If you do not provide a value for `context`, then its default value is determined as follows:

- If the command supports the `appName` argument, then the default value for `context` is the value of the `appName` argument. If the command does not support

the appName argument, then the default value is the name of the first service in the WSDL.

- If the command does not use a WSDL, for example genApplicationDescriptor, then the default value for context is the name of the WAR file without the -web.war or .war extension.

For version 2.1 EJBs, the value of context is placed in the <context-root> element in the oracle-webservices.xml deployment descriptor file.

For Web applications, the value of context is placed in the <context-root> element in the application.xml deployment descriptor file.

The URL is built from the protocol, the host and the port. For example,

http://localhost:8888/*host*/*port*

The protocol, http://localhost:8888/ is set at deployment time and is used to contact the Web service. The second part, the *host* is provided by the context argument. The third part, the *port*, is provided by the uri argument. The full URL is created by concatenating the protocol with the values of the context and uri arguments.

See Also:

"uri" on page 18-63 for an example of using the context and uri arguments to form an HTTP URL.

ddFileName

ddFileName <*String*>
Specifies the oracle-webservices.xml deployment descriptor that contains the settings you want to assign to the Web service. If this argument is not used, then an oracle-webservices.xml deployment descriptor will be generated.

When this argument is used with the *Assemble commands, management and custom serialization information will be copied from the specified file to the oracle-webservices.xml file in the archive.

If the specified file name cannot be found, WebServicesAssembler will stop processing.

The appendToExistingDDs and the ddFileName arguments can be used in the same command line invocation or Ant task.

See Also:

- Appendix C, "oracle-webservices.xml Deployment Descriptor Schema" for more information on this file.
- "appendToExistingDDs" on page 18-62 for more information on the impact of using this argument and ddFileName in the same command line or Ant task.

uri

uri <*String*>
Specifies the URI to use for the Web service. This is the third part of the URL used to contact the Web service. The first part is the protocol, which is set at deployment time. The second part is provided by the context argument. The full URL is created by concatenating the protocol, the context, and the uri arguments.

The default value for uri is the value of the appName argument. You cannot specify an empty string for uri.

Using this argument adds the `<url-pattern>` element to `web.xml` (for Web applications) and the `<endpoint-address-uri>` element to `oracle-webservices.xml` for EJB 2.1.

WSDL Access Arguments

The following arguments can be used to access the WSDL. The `Http*` arguments help to access the WSDL if it is a URL and resides on a network that uses an HTTP proxy server.

- fetchWsdlImports
- httpNonProxyHosts
- httpProxyHost
- httpProxyPort
- importAbstractWsdl
- wsdl

fetchWsdlImports

`fetchWsdlImports <true|false>`
Indicates if you want to make a local copy of the WSDL and everything it imports. When `fetchWsdlImports` is `true`, the specified WSDL and everything it imports will be copied to the WAR. When `false`, only the specified WSDL will be copied to the WAR. Default value is `false`.

This argument is typically used when the WSDL must be packaged with the deployment. This avoids the overhead, un-reliability, security, or threat-of-change issues that may arise when calling out for each runtime invocation requires a remote schema. The only time you would package a WSDL and its schema imports with a Web service is in a top down assembly.

httpNonProxyHosts

`httpNonProxyHosts <`*`String`*`>`
Specifies the name of the host that should *not* use an HTTP proxy. If there are multiple hosts they should be separated by a '|', for example `localhost|127.0.0.1`. This argument can help to get the WSDL if it is a URL that resides on a network that uses an HTTP proxy server.

httpProxyHost

`httpProxyHost <`*`String`*`>`
Specifies the name of the HTTP proxy host. This argument can help to get the WSDL if it is a URL that resides on a network that uses an HTTP proxy server.

httpProxyPort

`httpProxyPort <`*`int`*`>`
Specifies the port number for the HTTP proxy. This argument can help to get the WSDL if it is a URL that resides on a network that uses an HTTP proxy server.

importAbstractWsdl

`importAbstractWsdl <true|false>`
Specifies whether the generated WSDL will import the WSDL specified with the `wsdl` argument. If `true`, the generated WSDL will import the WSDL file. If `false`, the

generated WSDL elements and the elements in the original WSDL will be written to a single output file. Default value is `false`.

wsdl

`wsdl <String>`
Specifies the absolute file path, relative file path, or URL to a WSDL document. For example, `http://host:80/services/myservice?WSDL` and `myservice.wsdl` are valid values for this argument.

If the network uses an HTTP proxy server, and the WSDL is a URL, you may need to set the `httpNonProxyHosts`, `httpProxyHost` and `httpProxyPort` arguments.

WSDL Management Arguments

The following options can be used to control the contents of the generated WSDL.

- createOneWayOperations
- genQos
- qualifiedElementForm
- singleService
- soapVersion
- targetNamespace
- typeNamespace
- wsdlTimeout

createOneWayOperations

`createOneWayOperations <true|false>`
When this argument is set to `true`, methods that return `void` will have no response message. If it is not specified or set to `false`, the response message will be empty. Default value is `false`.

genQos

`genQos <true|false>`
Determines whether QOS (quality of service) information will be added to the WSDL in the form of capability assertions. Capability assertions are derived from the Web service management configuration in the `oracle-webservices.xml` deployment descriptor. If you set `genQos` to `true`, then capability assertions will be generated into the WSDL. You must also set the `ddFilename` argument to an `oracle-webservices.xml` deployment descriptor that contains the Web services management configuration. The default value of this argument is `false`.

See Also:

"Working with Capability Assertions" in the *Oracle Application Server Advanced Web Services Developer's Guide* provides more information on providing capability assertions to the server- and client-side of a Web service.

qualifiedElementForm

`qualifiedElementForm <true|false>`
Determines whether the `elementFormDefault` element in each generated schema in the WSDL is set to `qualified` or `unqualified`.

The value of the `elementFormDefault` element indicates whether locally declared elements must be qualified by the target namespace in an instance document. If the value of this element is `unqualified`, then locally declared elements should not be qualified by the target namespace. If the value is `qualified`, then locally declared elements must be qualified by the target namespace.

To set the value of the `elementFormDefault` element to `unqualified`, set the `qualifiedElementForm` argument to `false`. To set the value of the element to `qualified`, then set the argument to `true`. The default value for the `qualifiedElementForm` argument is `true`.

singleService

`singleService <true|false>`
Determines whether a single port or multiple ports will be generated for each service defined in the WSDL. If `true`, WebServicesAssembler generates a single service with multiple ports (one for each port type). If `false`, WebServicesAssembler generates multiple services, each with a single port. Default is `false`.

soapVersion

`soapVersion <1.1|1.2|1.1,1.2>`
Specifies the SOAP version to the WSDL document. Values can be `1.1`, `1.2`, or `1.1,1.2`. Default value is `1.1`.

The "`1.1,1.2`" value means that WebServicesAssembler will create two ports with two bindings. One port and binding will support version 1.1; the other port and binding will support version 1.2. Each port must be bound to a different URL. That is, you cannot support both versions concurrently with the same URL address.

targetNamespace

`targetNamespace <`*`String`*`>`
Specifies the target namespace to be used in the generated WSDL.

The value of `targetNamespace` can be either a conforming HTTP URL, a non-conforming HTTP URL, or even a URI. The value of `targetNamespace` maps to a package name in the following ways:

- When the target namespace is a conforming HTTP URL, then the order of the elements in it will be reversed when mapping to a Java package name. For example:

  The target namespace `http://hello.demo.oracle.com` will map to the package name `com.oracle.demo.hello`.

- When the target namespace is a non-conforming HTTP URL, then the order of the elements in it will be unchanged. For example:

  The target namespace `http://hello.demo.oracle` will map to the package name `hello.demo.oracle`.

- When the target namespace is a URN, then the value is mapped into a single identifier, with components separated by underscores ("_"). For example:

  The target namespace `urn:oracle.demo.hello` will map to the identifier `oracle_demo_hello`.

For best coding practices, follow these suggestions:

- When the code's package name maps to a conforming HTTP URL (when reversed), then use this URL as the target namespace. For example:

For package name `com.oracle.demo.hello`, use `-targetNamespace http://hello.demo.oracle.com`.

- When the code's package name does not map to a conforming HTTP URL (when reversed), then use the package name unchanged as the target namespace. For example:

  For package name `oracle.demo.hello`, use `-targetNamespace http://oracle.demo.hello`.

  Note: Avoid using a URN as the value for the `targetNamespace` argument, such as:

  `-targetNamespace urn:oracle.demo.hello`

  This will cause the service and the service elements to be mapped to two different locations. The service will be mapped to `oracle_demo_hello` and the service elements to `oracle.demo.hello`.

  For other ways in which the service and service elements can be generated into separate directories (either intentionally or unintentionally), see "Web Service and Message Parts are Generated into Separate Directories" on page F-11.

typeNamespace

`typeNamespace <String>`

Specifies the type namespace to be used in the schema types in the generated WSDL. The name that you specify will always be used and it will not be reversed.

See Also:

"Default Algorithms to Map Between Target WSDL Namespaces and Java Package Names" on page 18-72 for more information on this argument and the default namespace-to-package name mapping conventions.

wsdlTimeout

`wsdlTimeout <int>`

Specifies the number of seconds WebServicesAssembler should wait for a response to an HTTP or HTTPS request for a remote WSDL resource. The default value is 60 seconds.

By default, WebServicesAssembler will wait as long as one minute to receive the response to a HTTP request for a WSDL definition from a host. This argument is used to override the default wait limit. A value of zero indicates that WebServicesAssembler will wait for a period of time determined by the platform.

This argument applies only to requests made through HTTP or HTTPS. It will be ignored if the request is made through another protocol.

Message Format Arguments

The following arguments can be used to control the message format used by the generated WSDL, and hence, the Web service.

- mtomSupport
- style

- use

mtomSupport

`mtomSupport <true|false>`
For bottom up or top down Web service assembly, this attribute specifies whether messages with appropriate content (`base64binary`) will be encoded as MTOM formatted attachments. The default value for this attribute is `false`.

Using this attribute adds an `<mtom-support>` element to the `oracle-webservices.xml` file.

Note: This attribute can be used only with the `assemble`, `genProxy`, and `topDownAssemble` Ant tasks. It cannot be used as an argument on the WebServicesAssembler command line.

See Also:

See "Working with MTOM Encoded Attachments" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on how to use this argument.

style

`style <document-bare|document-wrapped|rpc>`
For bottom up Web service assembly, this argument specifies the `style` attribute of the message format in the generated WSDL. Possible values are `document-bare`, `document-wrapped`, and `rpc`. The default value is `document-wrapped`.

See Also:

"OracleAS Web Services Message Formats" on page 5-1 provides more information on the RPC and document (wrapped and bare) message styles.

use

`use <literal|encoded>`
For bottom up Web service assembly, this argument specifies the `use` attribute of the message format in the generated WSDL. Possible values are `literal` and `encoded`. The default value is `literal`.

See Also:

"OracleAS Web Services Message Formats" on page 5-1 provides more information on the literal and encoded message uses.

Java Generation Arguments

The following options can be used to control how Java files are generated.

- dataBinding
- mapHeadersToParameters
- overwriteBeans
- unwrapParameters
- valueTypeClassName

- valueTypePackagePrefix
- wsifEjbBinding
- wsifJavaBinding

dataBinding

`dataBinding <true|false>`

If `true`, WebServicesAssembler will attempt to generate Java value types that follow the JAX-RPC default type mapping rules for every element in the schema. If WebServicesAssembler encounters an unsupported type, it will use the `javax.xml.soap.SOAPElement` to represent it. If `false`, WebServicesAssembler uses a `SOAPElement` for every schema type it encounters. Default value is `true`.

> **Note:** 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.

mapHeadersToParameters

`mapHeadersToParameters <true|false>`

Indicates if SOAP headers defined in the WSDL should be mapped to parameters for each of the methods in the generated Java code. Default value is `true`.

overwriteBeans

`overwriteBeans <true|false>`

Indicates if beans should be generated for schema types even if the class already exists in the classpath. Default value is `false`.

unwrapParameters

`unwrapParameters <true|false>`

This argument can be set only for document-literal operations and will be ignored for other message formats. Typically a document-literal operation has a single input schema type and a single output schema type. The schema types are sometimes called wrappers. When `unwrapParameters` is set to `false` the generated service endpoint interface will be generated with wrappers around the input parameter and the return type. For example, a method with wrapped parameters may look like this.

```
public EchoResponse echo(EchoRequest p) throws RemoteException;
```

When `unwrapParameters` is set to `true`, which is the default, the return type and response type will be unwrapped. This is usually easier to use, especially if the types are simple. For example, a method with unwrapped parameters may look like this.

```
public String echo(String string) throws RemoteException;
```

valueTypeClassName

`valueTypeClassName <String>`

Specifies the fully-qualified class name of the JAX-RPC value type which is used for `java.util.Collection` and `java.util.Map`. This argument enables you to generate schemas for these classes since the service endpoint interface does not refer to them directly.

See Also:

"Oracle Specific Type Support" on page 5-6 provides an example of using `valueTypeClassName` to declare non-built-in value types as items in a `Map` or `Collection`.

The `valueTypeClassName` argument can be used multiple times on the command line or in an Ant task. In an Ant task, write the multiple occurrences in separate tags and use the `name` attribute. Example 18–4 illustrates the use of multiple instances of `valueTypeClassName` to generate schemas for the `tClass1`, `tClass2`, and `tClass3` type classes.

Example 18–4 Multiple Instances of valueTypeClassName in an Ant Task

```
<oracle:assemble appName="myService"
   output="build"
   input="myservice.jar"
   style="rpc"
   use="encoded">
            <oracle:porttype
              interfaceName="com.myCompany.myService.Hello"
              className="com.mycompany.HelloImpl"/>
            <oracle:valueTypeClassName name="tClass1"/>
            <oracle:valueTypeClassName name="tClass2"/>
            <oracle:valueTypeClassName name="tClass3"/>
</oracle:assemble>
```

valueTypePackagePrefix

`valueTypePackagePrefix <String>`

Specifies the package name prefix for all value types that are to be created from a WSDL. This argument enables you to group value type package names by service.

All value type classes will start with the specified package name. By default, the package name for value types is based on the namespace of the complex type in the schema. This argument can be used to prefix the package name that is derived from the namespace.

For example, if `valueTypePackagePrefix` is set to `myapp`, and the type's target namespace is `http://ws-i.org/`, then the package name will be `myapp.org.ws-i`.

If you need more control over the WSDL namespace to Java package mapping, then use a JAX-RPC mapping file. The `valueTypePackagePrefix` argument is ignored if there is a namespace to package mapping defined in the JAX-RPC mapping file.

See Also:

- "How to Specify a Root Package Name" on page 18-75 for more information on how to use the `valueTypePackagePrefix` argument.
- "Default Algorithms to Map Between Target WSDL Namespaces and Java Package Names" on page 18-72. for more information on the mapping between WSDL namespaces and Java package names.

wsifEjbBinding

`wsifEjbBinding <true|false>`

This argument is used in bottom up Web services assembly to add WSIF EJB bindings to the WSDL. If `true`, you must also specify the EJB's home interface `className` and `jndiName`. WebServicesAssembler will generate native WSIF EJB bindings in addition to SOAP bindings in the WSDL. The default value is `false`.

See Also:

- "WSIF EJB Extensions to the WSDL" in the *Oracle Application Server Advanced Web Services Developer's Guide* for information on the extensions that the wsifEjbBinding argument adds to the WSDL.
- "Configuring a WSIF Endpoint for Multiple EJB Ports" in the *Oracle Application Server Advanced Web Services Developer's Guide* for information on arguments that allow you to more closely control the definition of the WSIF EJB bindings or to specify multiple ports.

The `wsifEjbBinding`, `wsifJavaBinding`, and `wsifDbBinding` arguments are mutually exclusive. An exception will be thrown if two or more of these arguments are used in a single command line or Ant task.

wsifJavaBinding

`wsifJavaBinding <true|false>`
This argument is used in bottom up Web services assembly to add WSIF Java bindings to the WSDL. If `true`, you must also specify the `className` of the Java implementation class. WebServicesAssembler will generate native WSIF Java bindings in addition to SOAP bindings in the WSDL. The default value is `false`.

See Also:

- "WSIF Java Extensions to the WSDL" in the *Oracle Application Server Advanced Web Services Developer's Guide* for information on the extensions this argument adds to the WSDL.
- "Configuring a WSIF Endpoint for Multiple Java Ports" in the *Oracle Application Server Advanced Web Services Developer's Guide* for information on arguments that allow you to more closely control the definition of the WSIF Java bindings or to specify multiple ports.

The `wsifEjbBinding`, `wsifJavaBinding`, and `wsifDbBinding` arguments are mutually exclusive. An exception will be thrown if two or more of these arguments are used in a single command line or Ant task.

Resolving Name Collisions

The WSDL 1.1 specification allows various element definitions to have the same name within a specified namespace in the WSDL document. This can potentially cause name conflicts when mapping from WSDL to Java.

Section 4.3.12 of the JAX-RPC specification outlines the rules to resolve name collisions by appending suffixes to the mapped Java identifiers. The following commands make use of these rules when generating Java files:

- genInterface
- genProxy

- topDownAssemble

Table 18–5, contains content copied from the JAX-RPC specification for your convenience. It specifies rules and suffixes for avoiding name collisions in the WSDL to Java mapping. If there are no name collisions, then there is no requirement to use these suffixes.

Table 18–5 Name Collision Rules

Java Definition mapped from WSDL/XML	Appended Suffix in the Mapped Java Identifier	Examples
Java class based on the WSDL/XML to Java type mapping whose name is derived from that of a `xsd:simpleType` or an `xsd:complexType`	*`_Type`*	XML: `<xsd:complexType name="shared">` Java: `Shared_`*`Type`*`.java`

Java enumeration class (see Section 4.2.4 of the JAX-RPC specification)	*`_Enumeration`*	XML:
		`<xsd:simpleType name="shared">`
		`<xsd:restriction base="xsd:string">`
		`<xsd:enumeration value="foo"/>`
		`</xsd:restriction>`
		`</xsd:simpleType>`
		Java: `Shared_`*`Enumeration`*`.java`

Java class based on the WSDL/XML to Java type mapping whose name is derived from that of a `xsd:element`	*`_Element`*	XML: `<xsd:element name="shared">` Java: `Shared_`*`Element`*`.java`
service endpoint interface	*`_PortType`*	XML: `<wsdl:portType name="shared">` `<wsdl:binding name="shared"...>` Java: service endpoint interface: `Shared_`*`PortType`*`.java`
generated service interface	*`_Service`*	XML: `<wsdl: service name="shared" ... >` Java: `Shared_`*`Service`*`.java`
exception class	*`_Exception`*	Java: `Shared_`*`Exception`*`.java`

Default Algorithms to Map Between Target WSDL Namespaces and Java Package Names

The following sections describe the default algorithms that WebServicesAssembler uses to map between namespaces and package names.

- Java Package Name to WSDL Namespace Mapping Algorithm
- WSDL Namespace to Java Package Name Mapping Algorithm
- How to Specify a Namespace
- How to Specify a Root Package Name

Java Package Name to WSDL Namespace Mapping Algorithm

The WebServicesAssembler constructs a default type namespace from the package name. If the package name starts with a standard top-level domain name or an International Organization for Standardization (ISO) country code, the WebServicesAssembler "reverses" the package name and places it into an HTTP URL. Otherwise, the package name is directly placed into an HTTP URL.

> **Note:** The standard top level domain names are defined by the Internet Corporation For Assigned Names and Numbers (http://www.icann.org/tlds/).
>
> ISO Country codes are defined by the International Organization for Standardization (http://www.iso.org/iso/en/ISOOnline.frontpage).

For example, the package `com.oracle.mytypes` will have the type namespace `http://mytypes.oracle.com/`. The package `examples.chapter1` will have the type namespace `http://examples.chapter1/`.

The following sections describe the order in which the mapping algorithm in WebServicesAssembler attempts to map Java artifacts and types to WSDL artifacts and XML schema types.

- Mapping Java Artifacts to WSDL Artifacts
- Mapping Java Types to XML Schema Types

Mapping Java Artifacts to WSDL Artifacts

The following steps represent the order in which the WebServicesAssembler tool attempts to map Java artifacts to WSDL artifacts.

1. Use the value of the `targetNamespace` WebServicesAssembler argument to get the target namespace to be used in the generated WSDL. See "targetNamespace" on page 18-66 for more information on this argument.
2. Look up the value of the Java `<package-mapping>` element in the JAX-RPC mapping file.
3. Derive the name of the WSDL namespace from the Java package name of the service endpoint interface.

Mapping Java Types to XML Schema Types

The following steps represent the order in which the WebServicesAssembler tool attempts to map Java types to XML schema types.

1. Look up the value of the Java `<package-mapping>` element in the JAX-RPC mapping file.
2. Use the value of `typeNamespace` WebServicesAssembler argument to get the type namespace for the schema types in the generated WSDL. See "typeNamespace" on page 18-67 for more information on this argument.
3. Derive the WSDL namespace from the Java package of the value type.
4. Use the value of the `targetNamespace` attribute defined in the WSDL.

WSDL Namespace to Java Package Name Mapping Algorithm

The WSDL namespace to Java package name mapping is based on the algorithm in the Java Architecture for XML Data Binding (JAXB) specification, version 2.0. This specification can be found at the following Web site:

http://www.jcp.org/en/jsr/detail?id=222/

The following are exceptions to the algorithm defined in the specification:

- The algorithm in the JAXB specification states that the scheme part of the URI should be removed only if it is either `http` or `urn`. The current implementation has expanded the set of schemes to remove. The set includes the following: `http`, `https`, `ftp`, `mailto`, `file`, `nntp`, `telnet`, `ldap`, `nfs`, `urn`, and `tftp`.
- The algorithm in the JAXB specification does not specify what action to take when Globalization Support characters are present in the URI. In the current implementation, any non-ASCII characters are encoded in the form `u`*xxxx*, where *xxxx* is the four-digit UTF8 encoding.
- The JAXB specification states that Step 6 of the algorithm should be skipped if the top-level domain name is not a standard country code or top level domain name. The specification does not further clarify what processing should be done in this case. In the current implementation, the host string of the URI is tokenized with the delimiter ".", even when the string does not end with a top-level domain name or country code.

Table 18–6 lists some examples of package name to namespace mappings.

Table 18–6 Examples of Namespace to Package Name Mappings

Namespace	Package Name
http://army.mil/	mil.army
http://oracle/j2ee/ws_example/	oracle.j2ee.ws_example
http://somecompany.jp/	jp.somecompany
http://toplink.oracle.com	com.oracle.toplink
http://www.acme.com/go/espeak.xsd	com.acme.go.espeak
urn:dime/types.xsd	dime.types_xsd

The following sections describe the order in which the mapping algorithm in WebServicesAssembler attempts to map the service endpoint interface, value types, and their related artifacts in the WSDL to Java names and types.

Mapping the WSDL Service Endpoint Interface and Related Endpoint Artifacts to Java Package and Class Names

The following steps represent the order in which the WebServicesAssembler tool attempts to map the service endpoint interface and its related artifacts in the WSDL to Java names.

1. For the service endpoint interface, look up the value of the `<service-interface-mapping>` element in the JAX-RPC mapping file. See "JAX-RPC Mapping File Descriptor" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on this file.
2. Use the value of the `packageName` WebServicesAssembler argument to get the Java package name for the generated classes. See "packageName" on page 18-48 for more information on this argument.
3. Look up the value of the `<package-mapping>` element in the JAX-RPC mapping file.
4. Derive the Java package name from WSDL namespace.

Mapping WSDL Value Types and Related Artifacts to Java Names and Types

The following steps represent the order in which the WebServicesAssembler tool attempts to map the value types and their related artifacts in the WSDL to Java names and types.

1. Look up the `<java-xml-type-mapping>` element in the JAX-RPC mapping file. See "JAX-RPC Mapping File Descriptor" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on this file.
2. If the schema type is defined under the same namespace as the `targetNamespace` of the targeting document (a WSDL or schema), then use the value of the `packageName` WebServicesAssembler argument. See "packageName" on page 18-48 for more information on this argument.
3. Look up the value of the Java `<package-mapping>` element in the JAX-RPC mapping file.
4. Derive the Java package name from the WSDL namespace and the value of the `valueTypePackagePrefix` WebServicesAssembler argument as described in "valueTypePackagePrefix" on page 18-70. See also "How to Specify a Root Package Name" on page 18-75 for more information on this argument.

How to Specify a Namespace

Use the `typeNamespace` argument to explicitly specify a namespace. The name that you specify will always be used and it will not be reversed.

How to Specify a Root Package Name

Use the `valueTypePackagePrefix` argument to specify a root package name for all types in the schema(s) in the specified WSDL. All package names will start with this value. The specified value will be used only if there is no type namespace-to-package mapping declared in the JAX-RPC mapping file.

If the generated package name starts with the same value, it will not be added to the package name. This will avoid packages looking like `com.oracle.mytypes.com.oracle.mytypes`.

How to Establish a Database Connection

The aqAssemble, dbJavaAssemble, plsqlAssemble, and sqlAssemble commands require a database connection to generate a Web service. A connection can be made to the database at Web service assembly time and at runtime. At Web service assembly time, the WebServicesAssembler connects to the database to get information on the database entities such as PL/SQL packages or AQ queues. At Web service runtime, the user application uses the data source JNDI location to get the JDBC connection for database operations. WebServicesAssembler ensures that this information will be provided to the Web service runtime code.

The following arguments can be used to provide connection information to a Web service.

- dbUser
- dbConnection
- dataSource

To provide assembly time and runtime access to the database, use either the `dataSource` argument or the `dbConnection` and `dbUser` combination on the command line or in an Ant task.

- The `dbConnection` and `dbUser` arguments usually appear together. They provide the JDBC URL and the database schema and password to WebServicesAssembler for assembly time and runtime access to the database.
- The `dataSource` argument provides the JNDI location that allows WebServicesAssembler to provide assembly time and runtime access to the database.
- If all three arguments appear in an Ant task or on the command line, then the values of `dbConnection` and `dbUser` will be used for assembly time access and `dataSource` will be used for runtime access to the database.

Additional Ant Support for WebServicesAssembler

This section describes additional functionality that is available for WebServicesAssembler through Ant tasks. This functionality is not available on the WebServicesAssembler command line.

- How to Use Multiple Instances of an Argument in Ant
- How to Configure Proxy Generation in an Ant Task
- How to Configure a Port in an Ant Task
- How to Configure a Port Type in an Ant Task
- How to Configure Handlers in an Ant Task
- How to Add Files to an Archive
- How to Control a WebServicesAssembler Build

How to Use Multiple Instances of an Argument in Ant

The following arguments to WebServicesAssembler commands can be declared more than once on the command line or in an Ant task.

- input
- schema
- sqlstatement
- valueTypeClassName

In an Ant task, you must list multiple instances of any of these arguments as separate tags. When they are listed as separate tags, the `input`, `schema`, and `sqlstatement` arguments require a `value` attribute. The `valueTypeClassName` argument requires a `name` attribute.

For example, the following `assemble` command uses multiple instances of the `input` argument to generate an EAR file which includes `first.jar`, `second.jar`, and `third.jar`.

```
<oracle:assemble
          appName="myService"
          output="build">
             <oracle:PORTTYPE
               interfaceName="com.myCompany.myService.Hello"
               className="com.mycompany.HelloImpl">
```

```
            </oracle:porttype>
            <oracle:input value="first.jar"/>
            <oracle:input value="second.jar"/>
            <oracle:input value="third.jar"/>
</oracle:assemble>
```

The following `jmsAssemble` command uses multiple instances of the `valueTypeClassName` argument to identify class names of the JAX-RPC value types which can be items in `java.util.Collection` and `java.util.Map`.

```
<oracle:jmsAssemble
       linkReceiveWithReplyTo="true"
       targetNamespace="http://oracle.j2ee.ws/jms-doc"
       typeNamespace="http://oracle.j2ee.ws/jms-doc/types"
       serviceName="JmsService"
       appName="jms_service"
       context="jms_service"
       input="./demo/build/mdb_service.jar"
       output="./demo/dist"
       >
         <oracle:valueTypeClassName name="tClass1"/>
         <oracle:valueTypeClassName name="tClass2"/>
         <oracle:valueTypeClassName name="tClass3"/>
</oracle:jmsAssemble>
```

How to Configure Proxy Generation in an Ant Task

The `proxy` subtag enables you to generate a client proxy at the same time the server code is assembled. This is most useful during bottom up assembly when the name of the WSDL is not necessarily known ahead of time. You can use the `proxy` subtag in Ant tasks for the following commands.

- aqAssemble
- assemble
- corbaAssemble
- dbJavaAssemble
- ejbAssemble
- jmsAssemble
- plsqlAssemble
- sqlAssemble
- topDownAssemble

The `proxy` subtask provides functionality similar to the `genProxy` WebServicesAssembler command. Except for `wsdl`, any of the supported `genProxy` arguments can be used as attributes in the `proxy` subtag.

The `output` argument is supported as an attribute, but is not required as it is for `genProxy`. If `output` is not specified in the `proxy` subtag, then the value of `output` provided for the parent tag will be used with the `src/proxy` directory path appended to it. For example if the parent tag sets `output="build"`, then the output for the proxy subtag will be placed in `build/src/proxy`.

The following arguments can be used as attributes to the proxy subtag.

- classpath

- dataBinding
- ddFileName
- endpointAddress
- genJUnitTest
- mapHeadersToParameters
- mappingFileName
- output
- overwriteBeans
- packageName
- replyToConnectionFactoryLocation
- replyToQueueLocation
- searchSchema
- unwrapParameters
- useDimeEncoding
- valueTypePackagePrefix

Example 18–5 illustrates the use of a <proxy> subtag with a packageName attribute. In this example, assemble is used as the parent tag.

Example 18–5 Using the proxy Subtag in an Ant Task

```
<oracle:assemble...>   (or any command that supports the proxy tag)
  <oracle:proxy packageName="myproxy"/>
</oracle:assemble>
```

How to Generate Handler and Port Information into a Proxy

You can add message processing information and port-specific information to a proxy by using the <handler> and <port> tags.

- <handler>—to examine and potentially modify a request before it is sent to the remote host, or before the client processes the response, you can configure a <handler> tag as a sub tag of <proxy>. "How to Configure Handlers in an Ant Task" on page 18-81 provides more information on the <handler> tag.
- <port>—to generate a proxy for a particular port, you can configure a <port> tag as a sub tag of <proxy>. "How to Configure a Port in an Ant Task" on page 18-78 provides more information on the <port> tag.

How to Configure a Port in an Ant Task

The port identifies the network address of the endpoint hosting the Web service. For some WebServiceAssembler commands, Ant tasks allow you to specify a port tag as a child. This enables you to have a different configuration for each port. For example, you can assign two different transports or two different SOAP versions.

These Ant tasks allow you to specify a port tag.

- aqAssemble
- assemble
- corbaAssemble

- dbJavaAssemble
- ejbAssemble
- jmsAssemble
- plsqlAssemble
- sqlAssemble
- topDownAssemble
- genWsdl
- genDDs

These arguments can be used in a `port` tag (note: arguments in the context of Ant tasks are "attributes").

- bindingName
- endpointAddress
- portName (or name)
- replyToConnectionFactoryLocation
- replyToQueueLocation
- sendConnectionFactoryLocation
- sendQueueLocation
- soapVersion
- uri

In addition to these arguments, the `port` tag also enables you to specify a `name` argument which is the local part of the port name.

Within the `port` tag, each command can specify zero or more optional arguments. Each command supports a different subset of arguments. For the list of arguments that can be used for a particular command, see the command's "Additional Ant Support" section.

Example 18–6 illustrates using the `port` tag to specify a JMS and an HTTP transport to different ports. Note that the `port` declaration for the HTTP transport uses the `name` attribute. The `name` attribute is optional if you specify only one `port` in an Ant task. If you specify multiple `port` tags in an Ant task, then the `name` attribute is required.

Example 18–6 Assigning Different Transports to Different Ports

```
<oracle:port
      uri="/echo"
      sendQueue="jms/senderQueue"
      sendConnectionFactoryLocation="jms/senderQueueConnectionFactory"
      replyToConnectionFactoryLocation="jms/receiverQueueConnectionFactory"
      replyToQueue="jms/receiverQueue"/>
<oracle:port uri="echo2" name="EchoHttpPort"/>
```

Example 18–7 illustrates using the `port` tag to specify different SOAP message versions to different ports. When you specify multiple ports in the same Ant task, then each `port` tag will require a `name` attribute:

Example 18–7 Assigning Different SOAP Message Versions to Different Ports

```
<oracle:assemble
```

```
   ...
      <oracle:port uri="soap11" soapVersion="1.1" name="httpSoap11Port" />
      <oracle:port uri="soap12" soapVersion="1.2" name="httpSoap12Port" />
   ...
/>
```

How to Configure a Port Type in an Ant Task

Some Ant tasks allow you to specify a `porttype` tag as a child task. This enables you to configure different interfaces to a Web service. The Ant tasks for these WebServicesAssembler commands allow you to specify `porttype` as a child task:

- assemble
- genDDs
- genWsdl
- topDownAssemble

A `porttype` tag must have an `interfaceName` argument to identify the interface to the Web service. If you are specifying multiple `<porttype>` tags, then each tag must contain a `<port>` subtag.

> **See Also:**
>
> "How to Configure a Port in an Ant Task" on page 18-78 for more information on `port` tags.

When configuring a `porttype` with the `assemble`, `topDownAssemble`, `genDDs`, and `genWsdl` commands, you must specify a `className` attribute. For the `assemble` and `topDownAssemble` commands, you can specify a `classFileName` attribute instead of a `className` attribute.

Additionally, for the `topDownAssemble` command, if the WSDL contains references to multiple port types, then you must specify a `<porttype>` tag for each port type.

Table 18–7 summarizes the valid attributes and subtags that can be used with the `<porttype>` tag.

Table 18–7 Attributes and Subtags for the <porttype> Tag

Attributes and Subtags	Description
classFileName attribute	This attribute can be specified for the `assemble` and `topDownAssemble` commands if a `className` is not specified. It identifies the Java file name of the implementation class specified in the `className` argument.
className attribute	This attribute is required for the `assemble`, `topDownAssemble`, `genDDs`, and `genWsdl` commands. It specifies the name of the class (including the package name) that is the implementation class for the Web service.
interfaceName attribute	An `interfaceName` attribute is required. It identifies the interface to the Web service.
<port> subtag	A `<port>` subtag is required if you are specifying multiple `<porttype>` tags. It identifies the port to which the Web service interface applies. If you are specifying only one `<porttype>` tag, then a `<port>` tag is not required.

Example 18–8 illustrates assigning a port type to a Web service.

Example 18–8 Associating a Port Type to a Web Service

```
...
<oracle:porttype
   interfaceName="my.company.MyInterface"
   className="my.company.MyImpl">
</oracle:porttype>
...
```

How to Configure Handlers in an Ant Task

This section has the following subsections.

- Attributes and Child Tags for handler Tags
- Sample Handler Configuration
- Ant Tasks that Can Configure Handlers
- Configuring Multiple Handlers in an Ant Task

WebServicesAssembler lets you configure handlers and client handlers in Ant tasks. You can use the Ant tasks to configure all of the information defined for handlers by the Enterprise Web Services 1.1 specification.

> **Note:** Handlers and client handlers can be declared and configured only in Ant tasks. They cannot be declared or configured on the command line.

A handler can examine and potentially modify a request, response, or fault before it is processed by a Web service component. It can also examine and potentially modify the response or fault after the component has processed the request. A client handler, as its name implies, runs on the client before the request is sent to the remote host, and before the client processes the response.

The `handler` tag defines a handler in a WebServicesAssembler Ant task. Attributes define details about the handler. These tags and attributes correspond to the `<handler>` tag and its subelements defined by the Enterprise Web Services 1.1 specification. Declaring a handler and its attributes in an Ant task will set corresponding elements in `webservices.xml`.

Attributes and Child Tags for handler Tags

The following sections describe the attributes and child tags that can be used with the `handler` tag.

- class (attribute)
- initparam (child tag)
- name (attribute)
- soapheader (child tag)
- soaprole (attribute)

class (attribute)

`class="`*`class_name`*`"`

This attribute defines a fully-qualified class name for the handler implementation.

Using this attribute sets the <handler-class>*class_name*</handler-class> element in webservices.xml.

initparam (child tag)

<oracle:initparam name="*myName*" value="*myValue*" />
This child tag defines a name-value pair that a handler can use as an initialization parameter. The name attribute contains the name of a parameter. Each parameter name must be unique in the Web application. The value attribute contains the value of the corresponding parameter.

You can define multiple initparam declarations as children of a handler Ant task invocation.

Using the initparam child tag sets the following <init-param> structure in webservices.xml.

```
<init-param>
    <param-name>myName</param-name>
    <param-value>myValue</param-value>
</init-param>
```

name (attribute)

name="*handler_name*"
This attribute defines the name of the handler. The name must be unique within the module.

Using this attribute sets the <handler-name>*handler_name*</handler-name> element in webservices.xml.

soapheader (child tag)

<oracle:soapheader value="{*namespace_URI*}*local_part*"/>
This child tag defines the QName of a SOAP header that will be processed by the handler. The value attribute takes a local part of a QName and a namespace URI. You can define multiple soapheader declarations as children of a handler Ant task.

The value of the soapheader child tag is written to the soap-header tag in webservices.xml. For example, if the namespace URI is http://oracle.j2ee.ws/Header and the local part is authenticateHeader, then the soapheader child tag has the following value.

```
<oracle:soapheader value="{http://oracle.j2ee.ws/Header}authenticateHeader"/>
```

This value will have the following representation in the webservices.xml file.

```
<soap-header
xmlns:wsa1="http://oracle.j2ee.ws/Header">wsa1:authenticateHeader</soap-header>
```

In this example, wsa1 is a unique prefix for the given namespace.

soaprole (attribute)

soaprole ="*Some_SoapRole*"
This attribute defines a SOAP actor that the handler will play as a role.

Using this attribute sets the <soap-role>*Some_SoapRole*</soap-role> element in webservices.xml.

soapRole (child tag)

This child tag has the same behavior as the `soaprole` attribute: it defines a SOAP-actor that the handler will play as a role. However, unlike the attribute, the `soapRole` child tag enables you to specify more than one SOAP role for the handler.

For example, the following Ant task fragment uses the `soapRole` subtag to define two SOAP actors that the handler will play as a role.

```
...
<oracle:handler soapRole="SomeSOAPRole" name="StaticStateHandlerName"
                class="oracle.j2ee.ws.tools.wsa.handler.StaticStateHandler">
   <oracle:soapRole name="another-soap-role"/>
   <oracle:soapRole name="yet-another-soap-role"/>
</oracle:handler>
...
```

Sample Handler Configuration

To illustrate how to construct a handler configuration that can be used in an Ant task, Example 18–9 provides a sample configuration for the `StaticStateHandlerName` handler. The handler in implemented by the `myApp.handler.StaticStateHandler` class, and it requires two initialization parameters, `SCOTT` and `TIGER`. The handler will process two types of SOAP headers: `authenticateHeader` and `authenticateHeader2`.

Example 18–9 A Sample Handler Configuration

```
<some tag that supports handlers>
 <oracle:handler soaprole="Some_SoapRole"
   name="StaticStateHandlerName"
   class="myApp.handler.StaticStateHandler">
        <oracle:initparam name="id" value="SCOTT"/>
        <oracle:initparam name="password" value="TIGER"/>
   <oracle:soapheader value="{http://oracle.j2ee.ws/Header}authenticateHeader"/>
   <oracle:soapheader value="{http://oracle.j2ee.ws/Header2}
authenticateHeader2"/>
 </oracle:handler>
</some tag that supports handlers>
```

Ant Tasks that Can Configure Handlers

The Ant tasks for the following WebServiceAssembler commands allow you to specify a `handler` tag as a child task.

- aqAssemble
- assemble
- corbaAssemble
- dbJavaAssemble
- ejbAssemble
- jmsAssemble
- plsqlAssemble
- sqlAssemble
- topDownAssemble
- genProxy

- genDDs (can specify a `handler` tag for the server side only)

Configuring Multiple Handlers in an Ant Task

You can specify multiple handlers for any command that supports handlers. Each handler can have a different configuration.

Example 18–10 illustrates a multiple handler configuration. The parent tag can be any tag which supports handlers.

- The first handler tag configures the `StaticStateHandlerName` handler. The handler is implemented by the `myApp.handler.StaticStateHandler` class, and it requires two initialization parameters, `SCOTT` and `TIGER`. The handler will process two types of SOAP headers: `authenticateHeader` and `authenticateHeader2`.
- The second handler tag shows a very basic handler tag. It configures the `MyOtherHandler` handler that is implemented by the `myapp.handler.MyOtherHandler` class.

Example 18–10 Sample Multiple Header Configuration

```
<some tag that supports handlers>
   <handler soaprole="Some_SoapRole" name="StaticStateHandlerName"
        class="myApp.handler.StaticStateHandler">
     <initparam name="id" value="SCOTT"/>
     <initparam name="password" value="TIGER"/>
     <soapheader namespace="http://oracle.j2ee.ws/Header"
          localpart="authenticateHeader"/>
     <soapheader namespace="http://oracle.j2ee.ws/Header2"
         localpart="authenticateHeader2"/>
   </handler>
   <handler name="MyOtherHandler" class="myApp.handler.MyOtherHandler"/>
</some tag that supports handlers>
```

How to Add Files to an Archive

To add a file to an EAR or WAR archive, copy the file to the directory that WebServicesAssembler uses as a staging area where it jars the archive before calling the `*Assemble` task. Figure 18–1 illustrates the layout of the staging area.

The default staging area is *<output>*`/war` for a WAR archive and *<output>*`/ear` for an EAR archive. The *<output>* variable indicates the value of the `output` argument. For example, if the value of the `output` argument is `outputDir`, then the files are stored in `outputDir/war` and `outputDir/ear`.

For example, the following Ant task will place a key store `oraks.jks` into the generated EAR.

```
<copy file="oraks.jks" todir="build/ear/META-INF/>
<oracle:assemble output="build"
                ...
 />
```

How to Control a WebServicesAssembler Build

WebServicesAssembler provides a boolean `failonerror` argument that will allow you to continue a build even though there are errors. If the value of `failonerror` is `true`, the build fails if WebServicesAssembler encounters an error. If the value is `false`, then the build will continue. The default value is `true`.

The `failonerror` argument can be used with any WebServicesAssembler command. In the following example, WebServicesAssembler will continue to process if the assemble command encounters an error.

```
<oracle:assemble failonerror="false"
          ...
/>
```

How to Assemble Support for MTOM Encoded Attachemnts into a Web Service

The `mtomSupport` Ant attribute enables you to assemble support for MTOM encoded messages into a Web service and Web service client. You can use this attribute only in Ant tasks; is is not available on the WebServicesAssembler command line.

The `mtomSupport` attribute can be used with these Ant tasks:

- assemble
- genProxy
- topDownAssemble

See Also:

For more information on the `mtomSupport` attribute and how to use it, see the following sections:

- "mtomSupport" on page 18-68
- "Working wih MTOM Enoded Attachments" in the *Oracle Application Server Advanced Web Services Developer's Guide*

How to Assign Multiple Web Services to an EAR or WAR Archive

To assign multiple Web services in a single archive (EAR or WAR) the `assemble` or `topDownAssemble` task must be called for each Web service that is going in the WAR. The arguments defined for each assemble task must follow these rules.

- All `output` arguments must have the same value.
- Only the last `assemble` task can define an `ear` argument. The reason for this is when an `ear` argument is specified, the contents of the WAR staging directory (`/war`) will be archived and put in the EAR. When an archive is created, the files in the staging area are deleted.
- The `war` argument for all tasks except the last, must have the same value as the `output` argument with the directory name `/war` appended to it. This is because WebServicesAssembler's default behavior is to use the output directory, *`directory`*`/war`, as the staging area for the WAR. For example, if the `output` argument is set to `dist` then the `war` argument should be set to `dist/war`.
- All tasks except the first must set `appendToExistingDDs` to `true`. The first `assemble` task can set `appendToExistingDDs` to `true` only if the output *`directory`*`/war`, already contains deployment descriptors that you want WebServicesAssembler to modify (for example a `web.xml` with resource references).
- The last `assemble` task should not define a `war` argument unless you want WebServicesAssembler to create a WAR instead of an EAR.
- The `appName` argument must be unique in all of the `assemble` tasks.
- If a `uri` argument is specified it must be unique in all of the `assemble` tasks.

Example 18–11 displays the Ant tasks to assign two Web services, `firstApp` and `nextApp` to the `myApps.ear` EAR file. Note the following details in the example code.

- the `output` arguments for the two assemble commands are assigned the same value for the output directory path: `${out.dir}`.
- the first `assemble` command stores its output in a `/war` directory, while the last `assemble` argument specifies the EAR file that will store the two Web services.
- the last Web service specifies the `appendToExistingDDs` so that its deployment descriptor will be appended to the descriptor generated by the previous command.

The ellipsis indicates additional Ant commands.

Example 18–11 Ant Tasks to Assign Two Web Services to an EAR

```
<oracle:assemble appName="firstApp"
          output="${out.dir}"
          war="${out.dir}/war"
            ...
          />
      <oracle:assemble appName="nextApp"
          output="${out.dir}
          appendToExistingDDs="true"
          ear="myApps.ear"
            ...
          />
```

Example 18–12 displays the Ant tasks to assign three Web services, `firstApp`, `nextApp`, and `lastApp` to the `myApps.ear` EAR file. Note the following details in the example code.

- the `output` arguments for the three `assemble` commands are assigned the same value for the output directory path: `${out.dir}`.
- the first two `assemble` commands store their output in a `/war` directory, while the last `assemble` argument specifies the EAR file that will store the three Web services.
- the second and last Web services also specify the `appendToExistingDDs` so that their deployment descriptors will be appended to the descriptor generated by the previous command.

The ellipsis indicates additional Ant commands.

Example 18–12 Ant Tasks to Assign Three Web Services to an EAR

```
<oracle:assemble appName="firstApp"
          output="${out.dir}"
          war="${out.dir}/war"
            ...
          />
      <oracle:assemble appName="nextApp"
          output="${out.dir}
          appendToExistingDDs="true"
          war="${out.dir}/war"
            ...
          />
      <oracle:assemble appName="lastApp"
          output="${out.dir}
```

```
appendToExistingDDs="true"
ear="myApps.ear"
  ...
/>
```

Limitations on Assigning Multiple Web Services to a WAR File

WebServicesAssembler does not check for conflicts when adding multiple Web services to a single WAR. The following conflicts will cause an invalid assembly that will not be detected until deployment time or runtime.

- The different Web services contain WSDL files that have the same name.

  If the WSDL files in different Web services have the same name, then one of the WSDL files will overwrite the others. To avoid this conflict, ensure that WSDL file names in different Web services are unique.

- The different Web services contain class files that have the same name.

  If class files use the same name (including package name) in different Web services, then one of the classes will overwrite the other. In some cases, these files really are the same. For example, value type classes from the same schema types can be shared between two different services. This type of conflict will not cause a problem.

  Classes that cannot be the same include the service endpoint interface and the implementation class. If the class files are not the same, then they should either be renamed or placed in a different package.

How to Represent Java Method Parameter Names in the WSDL

When you are generating a Web service bottom up from Java classes or EJBs, WebServicesAssembler generates a WSDL. For a Java method's `<element name="..." />` attribute in the generated WSDL, WebServicesAssembler attempts to use the method's actual parameter names. Using the method's actual parameter names in the WSDL and SOAP messages makes it easier for you to determine which parameter corresponds to which element.

WebServicesAssembler uses these techniques to attempt to retrieve the method's parameter names in the following order.

1. If the interface source file is provided, WebServicesAssembler will parse the code for parameter names. To do this, the full path name of the Java interface file must be specified with the `interfaceFileName` argument.
2. If the interface source file is not provided, WebServicesAssembler will attempt to load and parse the Java class file that implements the methods in the interface. To be able to extract the parameter names, the class file must have been compiled with the `-g` option to `javac`.

If these techniques fail to retrieve the parameter names, or if the class is not loadable or obfuscated, then WebServicesAssembler uses the parameter's datatype and a number (for example, `string_1`) by default.

Example 18–13 and Example 18–14 provide WSDL fragments that represent the following method.

```
public String sayHello(String name)
```

Example 18–13 illustrates a WSDL fragment where WebServicesAssembler was able to retrieve the `sayHello` method's parameter name. The `<element name="..."/>` attribute and its value are highlighted in bold.

Example 18–13 WSDL Fragment With a Generated Parameter Name

```
<definitions name="HelloService" targetNamespace="http://hello.demo.oracle/">
    <types>
    <schema elementFormDefault="qualified"
targetNamespace="http://hello.demo.oracle/">
    <complexType name="sayHello">
       <sequence>
          <element name="name" nillable="true" type="string"/>
       </sequence>
    </complexType>
...
```

Example 18–14 illustrates a WSDL fragment where the `sayHello` method's parameter name could not be determined. WebServicesAssembler uses `string_1` value for the element name attribute. The `<element name="..."/>` attribute and its value are highlighted in bold.

Example 18–14 WSDL Fragment With a Default Parameter Name

```
<definitions name="HelloService" targetNamespace="http://hello.demo.oracle/">
    <types>
    <schema elementFormDefault="qualified"
targetNamespace="http://hello.demo.oracle/">
    <complexType name="sayHello">
       <sequence>
          <element name="string_1" nillable="true" type="string"/>
       </sequence>
    </complexType>
...
```

Limitations

See "Using WebServicesAssembler" on page F-10.

Additional Information

For more information on:

- assembling Web services from a WSDL, see Chapter 6, "Assembling a Web Service from WSDL".
- assembling stateful Web services, see Chapter 7, "Assembling a Web Service with Java Classes".
- assembling Web services from EJBs, see Chapter 8, "Assembling a Web Service with EJBs".
- assembling Web services from a JMS queue or topic, see Chapter 9, "Assembling Web Services with JMS Destinations".
- assembling Web services from database resources, see Chapter 10, "Assembling Database Web Services".
- assembling Web services with J2SE 5.0 Annotations, see Chapter 11, "Assembling Web Services with Annotations".

- assembling REST Web services, see Chapter 12, "Assembling REST Web Services".
- building J2EE clients, see Chapter 14, "Assembling a J2EE Web Service Client".
- building J2SE clients, see Chapter 15, "Assembling a J2SE Web Service Client".
- packaging and deploying Web services, see Chapter 19, "Packaging and Deploying Web Services".
- JAR files that are needed to assemble a client, see Appendix A, "Web Service Client APIs and JARs".
- Web services interoperability, see "Ensuring Interoperable Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- using quality of service features in Web service clients, see "Managing Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- adding security to a Web service, see the *Oracle Application Server Web Services Security Guide*.
- how to write clients to access Web services secured on the transport level, see "Adding Transport-level Security for Web Services Based on EJBs" and "Accessing Web Services Secured on the Transport Level" in the *Oracle Application Server Web Services Security Guide*.
- adding reliability to a Web service, see "Ensuring Web Service Reliability" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- adding an auditing and logging configuration to a Web service, see "Auditing and Logging Messages" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- processing nonstandard data types, see "Custom Serialization of Java Value Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- using the JAX-RPC mapping file and its contents, see "JAX-RPC Mapping File Descriptor" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- data types supported by OracleAS Web Services, see "Mapping Java Types to XML and WSDL Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- using the Web Service Invocation Framework, see "Using Web Services Invocation Framework" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- Oracle JDeveloper tool support for Web service development, see the Oracle JDeveloper on-line help.

19

Packaging and Deploying Web Services

This chapter describes the packaging of Web service files and deployment support offered by Oracle Application Server Web Services. Except for a few details, Web services are packaged and deployed in the same manner as any other J2EE application.

This chapter has the following sections.

- Understanding Web Service Packaging
- Packaging Structure for Web Service Applications
- Descriptions of Packaged Files
- Tool Support for Packaging
- Understanding Web Service Deployment
- Tool Support for Deployment

> **Note:** This chapter does not describe deployment itself. The deployment of Web modules and EJBs is covered in detail in the *Oracle Containers for J2EE Deployment Guide*.

Understanding Web Service Packaging

Web service files can be assembled and packaged by using either Oracle JDeveloper or WebServicesAssembler. The tools ensure that the correct files and deployment descriptors are included for the packaged application. Options in Oracle JDeveloper wizards and arguments to WebServicesAssembler commands allow you to configure values in the deployment descriptors that the Web service will use at runtime. The files are packaged into a deployable EAR file according to the rules outlined in the Enterprise Web Services 1.1 specification.

You can also assemble and package a Web service application by hand. Although this chapter describes Web service package structure and contents, it does not describe the details of manual assembly.

Deployment can be performed by using `admin_client.jar` on the command line, Ant tasks, or by using the Oracle JDeveloper or Application Server Control tools.

Table 19–1 summarizes the Oracle tools that can perform Web service packaging and deployment.

Table 19–1 Packaging and Deployment Support Offered by Oracle Tools

	WebServices-Assembler	Oracle JDeveloper	Application Server Control	admin_client.jar	Ant Tasks
Packaging	Yes	Yes	No	No	No
Deployment	No	Yes	Yes	Yes	Yes

As a final step, you can publish your deployed Web service to the Universal Description, Discovery, and Integration (UDDI) registry.

Note:

- Discussion of UDDI is beyond the scope of this documentation. For more information about working with UDDI, see the following Web site.

  http://www.uddi.org/specification.html

- For more information on Oracle's UDDI product, Oracle Application Server UDDI Registry, see the following Web site:

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

Packaging Structure for Web Service Applications

The Web service files which are intended for deployment to OC4J are included in a component deployment module. This component deployment module can be a JAR file for an EJB or a WAR file for Java classes. The component deployment module is then stored in a deployable EAR. The package structure for Web service files in an EAR file follows the rules defined by the Enterprise Web Services 1.1 specification. The following sections describe the packaging structure for Web services based on Java classes and EJBs.

- Packaging for a Web Service Based on Java Classes
- Packaging for a Web Service Based on EJBs

Packaging for a Web Service Based on Java Classes

For a Web service based on Java classes, the EAR file contains a WAR file. The deployment and class files reside under the `WEB-INF` directory in the WAR. The `WEB-INF` directory contains the `web.xml`, `webservices.xml`, `oracle-webservices.xml`, and the JAX-RPC mapping file. It also contains a `classes` directory for the class files, a `lib` directory for the JAR files, and a `wsdl` directory for the WSDL.

The `META-INF` directory contains the `MANIFEST.MF` manifest file which defines extension and package related data and the `application.xml` file which specifies the components of a J2EE application.

Example 19–1 illustrates the packaging structure for a Web service based on Java classes.

Example 19–1 Packaging Structure for a Web Service Based on Java Classes

```
<serviceName>.ear        contains
      META-INF/
         |--MANIFEST.MF
```

```
   |--application.xml

<serviceName>.war  contains
    WEB-INF/
       |--web.xml
       |--webservices.xml
       |--oracle-webservices.xml
       |--<mapping file>
       |--wsdl/
            |--<serviceName>.wsdl
       |--classes/
            |--class files
       |--lib/
            |--*jar files
```

Packaging for a Web Service Based on EJBs

The packaging structure for a Web service based on EJBs is similar to Java classes, except the EAR file contains a `META-INF` directory for the manifest and a JAR file. Within the JAR file is another `META-INF` directory which contains the deployment and class files for the EJB. The `META-INF` directory within the JAR contains the `ejb-jar.xml`, `webservices.xml`, `oracle-webservices.xml`, and the JAX-RPC mapping file. It also contains the class files and a `wsdl` directory for the WSDL. Example 19–2 illustrates the packaging structure for a Web service based on EJBs.

Example 19–2 Packaging Structure for a Web Service Based on EJBs

```
<serviceName>.ear  contains
       META-INF/
          |--MANIFEST.MF
          |--application.xml

       <serviceName>.jar contains
            class files
            META-INF/
                |--ejb-jar.xml
                |--webservices.xml
                |--oracle-webservices.xml
                |--<mapping file>
                |--wsdl/
                     |--<serviceName>.wsdl
```

Descriptions of Packaged Files

The following list describes the files that are packaged for deployment.

- `application.xml`—describes all of the WARs and EJB JARs in the EAR. It specifies the components of a J2EE application, such as EJB and Web modules, can specify additional configuration for the application as well. This descriptor must be included in the `/META-INF` directory of the application's EAR file. The `application.xml` file is defined by the `application_1_4.xsd` schema located at the following Web site.

  `http://java.sun.com/xml/ns/j2ee/application_1_4.xsd`

- `ejb-jar.xml`—the deployment descriptor for an EJB component. It defines the specific structural characteristics and dependencies of the Enterprise JavaBeans within a JAR, and provides instructions for the EJB container about how the beans expect to interact with the container.

There is a relationship between the contents of `webservices.xml` and `ejb-jar.xml` to identify the EJB exposed as a Web service. Figure 19–1 illustrates this relationship.

The `ejb-jar.xml` file is defined by the schema located at the following Web site.

`http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd`

- *<mapping file>*`.xml` (for example, *serviceName*`_java-wsdl-mapping.xml`)—the JAX-RPC mapping file that maps Java interfaces, methods, and parameters to the WSDL. For more information on this file, see *JAX-RPC Mapping File Descriptor* in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- `oracle-webservices.xml`—a deployment descriptor that defines deployment properties specific to a Web service application running on OracleAS Web Services. For more information on the contents of this file, see Appendix C, "oracle-webservices.xml Deployment Descriptor Schema".
- `web.xml`—this deployment descriptor is defined by the Java Servlet 2.4 specification. This deployment descriptor can be used to deploy a Web application on any J2EE-compliant application server. For more information on the `web.xml` file, see the Java Servlet 2.4 specification at the following Web site.

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

  There is a relationship between the contents of `webservices.xml` and `web.xml` to identify the servlet exposed as a Web service. Figure 19–2 illustrates this relationship.

  The `web.xml` file is defined by the `web-app_2_4.xsd` schema located at the following Web site:

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

- `webservices.xml`—a standard configuration file for a Web Service application packaged within a component deployment module. It defines the Web service endpoint, associated configuration files, WSDL information, and JAX-RPC mapping data. It provides the location of the WSDL file (`<wsdl-file>`), the mapping file (`<jaxrpc-mapping-file>`), the `<port-component>` corresponding to the ports in the WSDL, the Java service endpoint interface (`<service-endpoint-interface>`), the Java representation of the WSDL, and the servlet name (`<servlet-link>`) or EJB name (`<ejb-link>`).

  There is a relationship between the contents of `webservices.xml` and the `ejb-jar.xml`, `oracle-webservices.xml`, and `web.xml` files. These relationships identify the component being exposed as a Web service and to pass metadata between files. "Relationships Between Deployment Descriptor Files" provides more information on these relationships.

  The `webservices.xml` file is defined by the `j2ee_web_services_1_1.xsd` schema located at the following Web site:

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

- WSDL file—describes the interface of the Web service and the format of the messages used to invoke it. If an archive containing a Web service does not include a Web Services Description Language (WSDL) document, OracleAS Web Services will generate a WSDL document at deployment time. The WSDL is defined by the specification located at the following Web site.

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

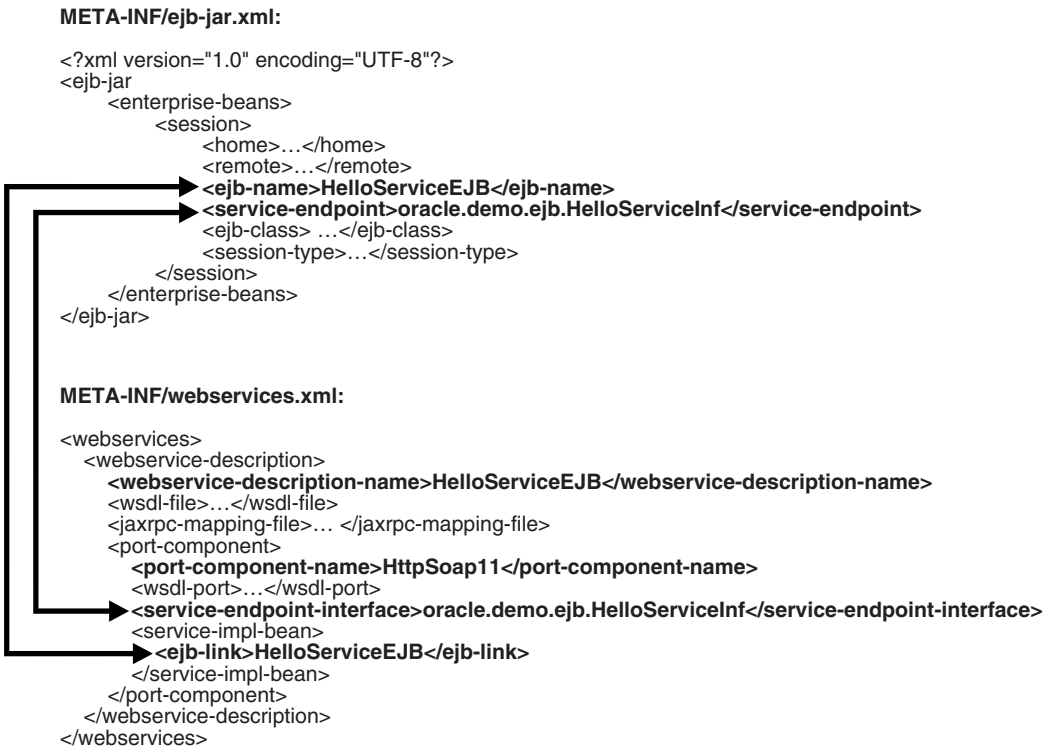
Relationships Between Deployment Descriptor Files

This section illustrates the relationships between the `webservices.xml` file and the `ejb-jar.xml`, `oracle-webservices.xml`, and `web.xml` files.

Relationships Between webservices.xml and ejb-jar.xml

Figure 19–1 illustrates the relationship between the contents of `webservices.xml` and `ejb-jar.xml` for Web services based on EJBs. The `<ejb-link>` element in `webservices.xml` provides the same value as `<ejb-name>` in `ejb-jar.xml`. This mapping identifies the EJB that is to be exposed as a Web service and hence over HTTP (SOAP over HTTP). The `<service-endpoint-interface>` element in `webservices.xml` provides the same value as `<service-endpoint>` element in `ejb-jar.xml`. This serves as an additional uniqueness constraint on the EJB.

Figure 19–1 Relationship Between webservices.xml and ejb-jar.xml



Relationships Between webservices.xml and oracle-webservices.xml

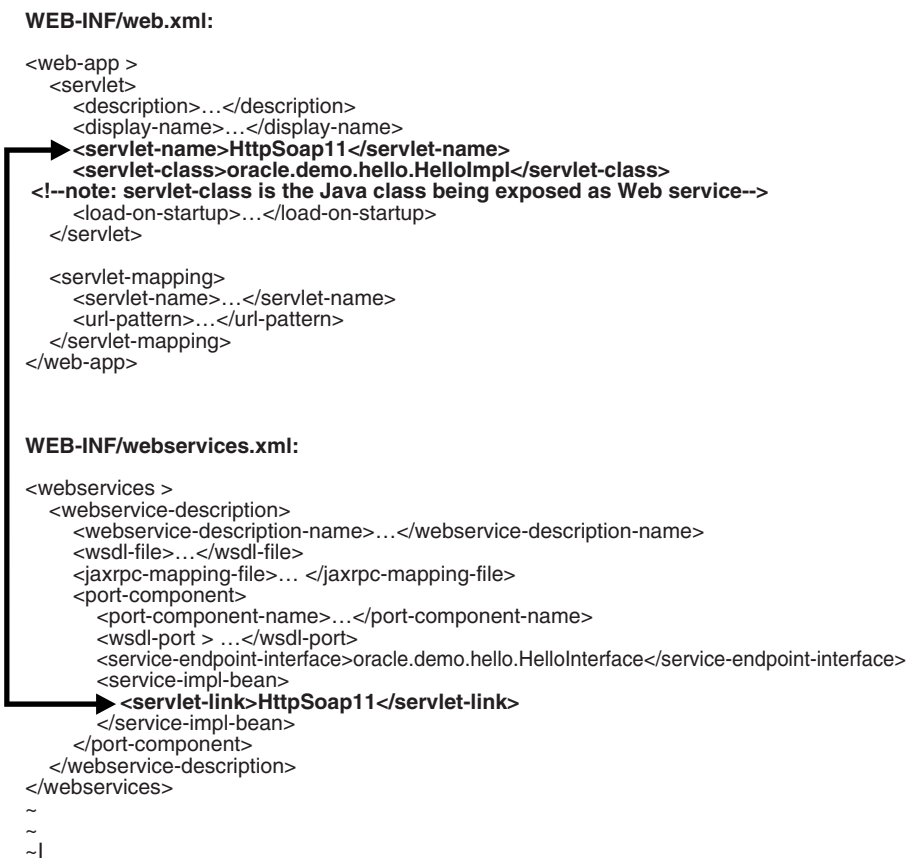
Figure 19–2 illustrates the relationship between the contents of the J2EE standard `webservices.xml` deployment descriptor and the `oracle-webservices.xml` proprietary deployment descriptor. This relationship allows metadata to be mapped between files. The `<port-component-name>` element in `webservices.xml` provides the same value as the `name` attribute in `<port-component name="...">` in `oracle-webservices.xml`. The `<webservice-description-name>` element in `webservices.xml` provides the same value as the `name` attribute in `<webservice-description name="...">` in `oracle-webservices.xml`.

Figure 19–2 Relationship Between webservices.xml and oracle-webservices.xml

Relationships Between webservices.xml and web.xml

Figure 19–3 illustrates the relationship between the contents of `webservices.xml` and `web.xml` to identify the servlet that is to be exposed as a Web service The `<servlet-link>` element in `webservices.xml` provides the same value as the `<servlet-name>` present in `web.xml`. This allows the Web service implementation to be exposed as a servlet and hence over HTTP (SOAP over HTTP). This generalized approach allows application packages to be portable across J2EE containers implementing JAX-RPC and the Enterprise Web Services 1.1 specification. An application adhering to these standards can be deployed to any container that complies with the Enterprise Web Services 1.1 specification and can be invoked using any Web service client.

Figure 19–3 Relationship Between webservices.xml and web.xml



Tool Support for Packaging

This section describes the tool support for packaging Web service files offered by OracleAS Web Services. Packaging can be performed by WebServicesAssembler and by Oracle JDeveloper.

- Packaging Support with WebServicesAssembler
- Packaging Support with Oracle JDeveloper

Packaging Support with WebServicesAssembler

This section describes the packaging support offered by WebServicesAssembler. The commands that assemble a Web service also package it in a deployable EAR file. In the course of assembling the files, many of these commands also create the deployment descriptors. The deployment descriptors contain the declarative data required to deploy the components as well as the assembly instructions that describe how the components are composed into an application.

- WebServicesAssembler Commands that Perform Packaging
- Managing Deployment Descriptors

WebServicesAssembler Commands that Perform Packaging

WebServicesAssembler provides several commands that assemble all of the files needed for a Web service.

- aqAssemble
- assemble
- corbaAssemble
- dbJavaAssemble
- ejbAssemble
- jmsAssemble
- plsqlAssemble
- sqlAssemble

WebServicesAssembler handles the generation of all relevant deployment descriptors and maps the proprietary configuration needed by the applications into Oracle specific deployment files. It also packages all of the relevant files so that they can be deployed to an application server.

These commands support arguments that let you package or save the generated files in a number of ways. The `ear` argument saves the files as a deployable EAR file. The `war` argument saves the files as a WAR file. The files are also be saved un-archived, in a directory that contains the contents of a WAR.

See Also:

- "Web Service Assembly Commands" on page 18-4 for more information on the commands listed in this section.
- "ear" on page 18-44 and "war" on page 18-51 for more information on these arguments.

Managing Deployment Descriptors

A number of WebServicesAssembler commands create deployment descriptors in the course of assembling a Web service. WebServicesAssembler does not perform deployment, but arguments to WebServicesAssembler commands will allow you to set values in the deployment descriptors.

- Commands that Create Deployment Descriptors
- Arguments that Affect Deployment Descriptor Contents

Commands that Create Deployment Descriptors Several of the WebServicesAssembler commands generate the `application.xml`, `web.xml`, `webservices.xml`, and `oracle-webservices.xml` deployment descriptors as part of their output. The commands that generate these files are:

- aqAssemble
- assemble
- corbaAssemble
- dbJavaAssemble
- ejbAssemble
- jmsAssemble
- plsqlAssemble
- sqlAssemble
- topDownAssemble

- genApplicationDescriptor (generates `application.xml` only)
- genDDs

Arguments that Affect Deployment Descriptor Contents This section lists deployment descriptor files and identifies the WebServicesAssembler arguments that can affect their content.

application.xml:

- context—when used for web applications, adds the `<context-root>` element.

oracle-webservices.xml:

- callScope—adds the `<param name="scope">call</param>` element.
- context—when used for EJB 2.1, adds the `<context-root>` element.
- dataSource—adds the following `param name` attribute to the `<implementor>` element. In the following example, *dataSource value* is the value specified for the `dataSource` argument.

  `<param name="databaseJndiName">`*dataSource value*`</param>`
- ddFileName—when used with the `*Assemble` commands, management and custom serialization information will be copied from the specified file to the `oracle-webservices.xml` file in the archive.
- mtomSupport—adds the boolean `<mtom-support>` subelement to the `<port-component>` element.
- restSupport—adds a boolean `<rest-support>` subelement to the `<port-component>` element and the `<provider-port>` element.
- session—adds the `<param name="scope">session</param>` element.
- timeout—adds the `<param name="session-timeout">`*integer*`</param>` element.
- uri—when used for EJB 2.1, adds the `<endpoint-address-uri>` element.
- useDimeEncoding—adds the `<use-dime-encoding>` element.

***serviceName*-java-wsdl-mapping.xml:**

In this file name, ***serviceName*** indicates the name of the Web service (which is specified by the `serviceName` argument).

- interfaceName—when used with the commands to assemble a Web service from database resources (`plsqlAssemble`, `sqlAssemble`, `dbJavaAssemble`, or `aqAssemble`) adds the `<service-endpoint-interface>` element.

web.xml:

- className—adds the `<servlet-class>` element.
- recoverable—adds a boolean `<distributable>` element.
- uri—when used for web applications, adds the `<url-pattern>` element.

webservices.xml:

- ejbName—adds the `<ejb-link>` element to `webservices.xml` for EJB 2.1 only.
- interfaceName—adds the `<service-endpoint-interface>` element.
- mappingFileName—adds the `<jaxrpc-mapping-file>` element. Note that the location and name of the file may be changed when put in the deployment

descriptor. The contents of the file may be modified before being placed in the archive if some mappings were not defined in the original file.

- `<handler>` tags—the data in handler tags are added to `webservices.xml`. These tags can be used only in Ant tasks.

Packaging Support with Oracle JDeveloper

Oracle JDeveloper provides wizards that can package your Web service application for deployment. For more information on the support offered by Oracle JDeveloper, see the following topics in the Oracle JDeveloper on-line help.

- *About J2EE Archive Formats*

  Lists the archive types and their associated module types and contents, as supported by Oracle JDeveloper.

- *Configuring Applications for Deployment*

  Provides links to topics dealing with the configuration and packaging of deployment descriptors, client applications, EJBs, and applets.

- *Configuring EJBs for Deployment*

  Takes you through the steps of creating a EJB JAR File deployment profile and adding an `ejb-jar.xml` deployment descriptor.

- *Configuring a Client Application for Deployment*

  Takes you through the steps of creating a client JAR file deployment profile and creating the `application-client.xml` deployment descriptor file.

- *Configuring an Applet for Deployment*

  Takes you through the steps of creating a WAR File deployment profile and adding a `web.xml` deployment descriptor.

Understanding Web Service Deployment

Deployment is the process which transfers application files to the server where the application will run. The deployment of Web modules and EJBs is covered in detail in the following chapters of the *Oracle Containers for J2EE Deployment Guide*.

- For guidelines on deploying WAR files into OracleAS Web Services, see "Deploying Web Modules".
- For guidelines on deploying EJB archives, see "Deploying Enterprise JavaBeans".
- For guidelines on deploying applications from the command line, see "Deploying Applications with admin_client.jar".
- For guidelines on deploying applications from Ant tasks, see "Deploying with the OC4J Ant Tasks".

Web services can be deployed by using Oracle JDeveloper or Application Server Control. You can also deploy Web services by using Ant tasks or by using `admin_client.jar` on the command line.

See Also:

"Tool Support for Deployment" for more information on using these tools for deployment.

Tool Support for Deployment

This section describes the tool support for deployment offered by OracleAS Web Services. Deployment can be performed through the command line, Oracle JDeveloper, and Application Server Control.

- Command Line Support for Deployment
- Ant Task Support for Deployment
- Deployment Support with Oracle JDeveloper
- Deployment Support with Application Server Control

Command Line Support for Deployment

The `admin_client.jar` command-line utility provided with OC4J can be used to deploy Web services packaged within an EAR file. You may want to use this utility if you plan to script the application deployment process. However, deploying standalone modules, such as a Web module packaged in a WAR file, is not supported using `admin_client.jar`.

See Also:

"Deploying Applications with admin_client.jar" in the *Oracle Containers for J2EE Deployment Guide* for instructions on deploying applications with this tool.

A Sample Deployment Using admin_client.jar

An EAR file containing files for a Web service can be deployed in the same way as other J2EE applications. The following is a sample deployment command.

```
java -jar <oc4jHome>/j2ee/home/admin_client.jar
deployer:oc4j:<oc4jHost>:<oc4jOrmiPort> <adminId> <adminPassword>
           -deploy
           -file dist/hello.ear
           -deploymentName hello
           -bindWebApp default-web-site
```

The following list describes the parameters in this code example.

- *<oc4jHome>*—The directory containing the OC4J installation.
- *<oc4jHost>*:*<oc4jOrmiPort>*—The host name and port of the OC4J server to which you are deploying the EAR file or J2EE application.
- *<adminId>*—The user name for the OC4J instance. The user assigns this value when OC4J is installed.
- *<adminPassword>*—The password for the OC4J instance. The user assigns this value when OC4J is installed.
- `default-web-site` —The Web site to which the application will be bound. This is usually `default-web-site`. To configure Web sites, see the `server.xml` file in *<oc4jHome>*`/j2ee/home/config`.

Ant Task Support for Deployment

OracleAS Web Services provides a set of Ant tasks for deploying and undeploying J2EE applications and modules to an OC4J instance. "Deploying with the OC4J Ant Tasks" in the *Oracle Containers for J2EE Deployment Guide* describes the Ant tasks and

provides guidelines for integrating the tasks into your application build process. This chapter includes the following topics.

- "Incorporating the OC4J Ant Tasks into Your Environment" outlines the procedure for incorporating the OC4J Ant tasks into the build environment.
- "Invoking the OC4J Ant Tasks" includes descriptions of how to invoke the following Ant tasks:
  - `bindAllWebApps`—The `bindAllWebApps` task binds the Web modules within a previously deployed EAR to a specified Web site.
  - `bindWebApp`—The `bindWebApp` task binds the application to the Web site that will be used to access it.
  - `deploy`—The `deploy` task deploys a J2EE application or module packaged in an archive.
  - `deployerUri`—The `deployerUri` task specifies the target OC4J instance or instances for the task.
  - `modifySharedLibrary`—The `modifySharedLibrary` task enables you to make changes to an existing shared library installed in an OC4J instance or in a group of instances across an Oracle Application Server cluster.
  - `publishSharedLibrary`—The `publishSharedLibrary` task installs a shared library in an OC4J instance or in a group of instances across an Oracle Application Server cluster.
  - `removeSharedLibrary`—The `removeSharedLibrary` task removes a shared library from an OC4J instance or from a group of instances across an Oracle Application Server cluster.
  - `start`, `stop`—The `start` and `stop` tasks start, stop, or restart an application and its child applications as part of a deployment operation on a specific OC4J instance.
  - `undeploy`—The `undeploy` task removes the specified application or module from the OC4J instance. This task also unbinds the application from the Web site automatically.
  - `updateEjbModule`—The `updateEjbModule` task allows incremental or partial redeployment of EJB modules within an application running in an OC4J instance or in a group of instances across an Oracle Application Server cluster.

Deployment Support with Oracle JDeveloper

The following list describes the topics available in the Oracle JDeveloper online help for deploying Web services. For more information on each of these topics, see the Oracle JDeveloper on-line help

- *Simple JAR Deployment*

  Takes you through the steps of deploying your application to an executable JAR file, or a JAR file on your file system.

- *Deploying Web Services to Embedded OC4J*

  Takes you through the steps of deploying your Web service to Oracle Application Server or an embedded instance of OC4J that runs on your local server.

- *Deploying Web Services to External OC4J*

Takes you through the steps of deploying your Web service to Oracle Application Server or an external instance of OC4J that runs on a remote server.

- *Deploying Secure Oracle Application Server Web Services*

  Takes you through the steps of deploying a J2EE 1.4 Web service that uses security. The steps describe how to bundle the keystore with the Web service for deployment.

Deployment Support with Application Server Control

The following list describes the topics available in the Application Server Control on-line help for deploying Web services. For more information on each of these topics, see the Application Server Control on-line help

- *Deploy: Deployment Settings Page*

  The deployment plan page provided with the Application Server Control Console includes the ability to set values in the OracleAS Web Services deployment descriptor (`oracle-webservices.xml`) at deployment time.

- *Deploy: Application Attributes Page*

  Describes the deployment attributes that you can configure by using Application Sever Control.

- *Deploying an Application*

  Takes you through the steps of deploying an application.

- *Redeploying and Undeploying Applications*

  Takes you through the steps of redeploying or undeploying an application.

Limitations

See "Packaging and Deploying Web Services" on page F-12.

Additional Information

For more information on:

- assembling Web services from a WSDL, see Chapter 6, "Assembling a Web Service from WSDL".
- assembling stateful Web services, see Chapter 7, "Assembling a Web Service with Java Classes".
- assembling Web services from EJBs, see Chapter 8, "Assembling a Web Service with EJBs".
- assembling Web services from a JMS queue or topic, see Chapter 9, "Assembling Web Services with JMS Destinations".
- assembling Web services from database resources, see Chapter 10, "Assembling Database Web Services".
- assembling Web services with J2SE 5.0 Annotations, see Chapter 11, "Assembling Web Services with Annotations".
- building J2EE clients, see Chapter 14, "Assembling a J2EE Web Service Client".
- building J2SE clients, see Chapter 15, "Assembling a J2SE Web Service Client".

- using the WebServicesAssembler tool to assemble Web services, see Chapter 18, "Using WebServicesAssembler".
- using the JAX-RPC mapping file and its contents, see "JAX-RPC Mapping File Descriptor" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- the contents of `wsmgmt.xml` management policy file, see "Understanding the Web Services Management Schema" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- working with DIME attachments, see "Working with Attachments" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- adding transport-level security to Web services based on EJBs, see "Adding Transport-level Security to a Web Service" and "Accessing Web Services Secured on the Transport Level" in the *Oracle Application Server Web Services Security Guide*.
- the Oracle Web Services Manager tool, see the *Oracle Web Services Manager Administrator's Guide*.
- using quality of service features in Web service clients, see "Managing Web Services" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- adding security to a Web service, see the *Oracle Application Server Web Services Security Guide*.
- adding reliability to a Web service, see "Ensuring Web Service Reliability" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
- adding an auditing and logging configuration to a Web service, see "Auditing and Logging Messages" in the *Oracle Application Server Advanced Web Services Developer's Guide*.

A

Web Service Client APIs and JARs

This appendix contains high-level descriptions of the Web services client API packages. It also identifies the JAR files that are required to run a Web service client.

This appendix contains these sections:

- Web Services API Packages
- Setting the Classpath for a Web Service Proxy

Web Services API Packages

Table A–1 lists the public Oracle APIs available in the current release of Oracle Application Server Web Services. The `provider` package can be used only in the OC4J container. The other APIs can be used in the container or in a Web services client.

See Also:

For more information on the APIs listed in this table, see:

- *Oracle Application Server Web Services Java API Reference*
- the Oracle Technology Network

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

Table A–1 Client API Packages

Package Name	Description
oracle.webservices	Contains the Oracle specific extension classes and interfaces that are common for all modules.
oracle.webservices.attachments	Contains classes and interfaces to support streaming attachments. For more information on using this package, see "Working with Message Attachments" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
oracle.webservices.databinding	Contains the `SOAPElementSerializer` interface which you can use for custom serialization. For more information on using the functionality provided by the `databinding` package, see "Custom Serialization of Java Value Types" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
oracle.webservices.management-portability	Web services platform portability layer.

Table A–1 (Cont.) Client API Packages

Package Name	Description
oracle.webservices.provider	Contains the interfaces and classes needed to implement provider-based endpoints. For more information on using the `provider` package, see "Using Web Service Providers" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
oracle.webservices.reliability	Contains the interfaces and classes needed to implement reliable SOAP message exchange using WS-Reliability. For more information see "Dynamically Configuring Client Side Reliability" in the *Oracle Application Server Advanced Web Services Developer's Guide*.
oracle.webservices.security.callback	Provides classes to support Web Services security call back.
oracle.webservices.soap	Contains the Oracle extension classes and interfaces to support SOAP 1.2. For more information on this API, see "Understanding the OraSAAJ APIs" on page 5-12.
oracle.webservices.transport	Contains the classes and interfaces to support multiple transport bindings. "Writing Client Code to Support JMS Transport" in the *Oracle Application Server Advanced Web Services Developer's Guide* provides information on how classes in this package are used in JMS transport.
oracle.webservices.wsdl	Contains the Oracle specific interface to support SOAP 1.2 operations in WSDL 1.1. For more information on this interface, see Appendix B, "Oracle Implementation of the WSDL 1.1 API".

Setting the Classpath for a Web Service Proxy

When you build a Web service client, you must enter the correct classpath to run it. The following sections list the JAR files that can be included on the classpath. The tables in this section use the *OC4J_HOME* environment variable to specify the location where the Oracle Application Server or the standalone OC4J is installed.

- Simplifying the Classpath with wsclient_extended.jar
- Classpath Components for Clients using a Client-Side Proxy
- WS-Security-Related Client JAR Files
- Reliability-Related Client JAR File
- JMS Transport-Related Client JAR File

Simplifying the Classpath with wsclient_extended.jar

To simplify your configuration of J2SE environments, OracleAS Web Services provides a `wsclient_extended.jar` file that contains all the classes necessary to compile and run a Web service client. This file can be listed on the classpath as an alternative to listing individual JARs.

The `wsclient_extended.jar` file contains these class files:

- core Web service client class files in the individual JAR files listed in Table A–2
- OC4J Security-related class files in the JARs listed in Table A–3
- WS-Security-related class files in the JARs listed in Table A–4
- WS-Reliability-related class files in the JARs listed in Table A–5

- JMS transport-related class files in the JAR listed in Table A–6

You do not have to include any of these JAR files on the classpath if you run the client with the `wsclient_extended.jar` file.

The `wsclient_extended.jar` file is available as a separate download from the Oracle Technology network:

`http://download.oracle.com/otn/java/oc4j/10131/wsclient_extended.zip`

The file can also be installed from the OC4J companion CD. In this case, you will find it in the *ORACLE_HOME*`/webservices/lib` directory.

Classpath Components for Clients using a Client-Side Proxy

Table A–2 lists the JAR files that can be included in the classpath for a J2SE Web services client.

Note:

- The `wsclient.jar` listed in this table contains only the core Web service client classes. At compilation and runtime, the Web service client requires many other classes such as those for the XML parser, the WSDL parser, and quality of service (QOS). All of these required classes are packaged into `wsclient_extended.jar` to simplify classpath set up.
- Not all JAR files are required in all cases.

Table A–2 Classpath Components for a Client Using a Client-Side Proxy

Component JAR	Description
ORACLE_HOME/diagnostics/lib/ojdl.jar and *ORACLE_HOME*/diagnostics/lib/ojdl2.jar	ODL implements APIs to be used by Oracle products to emit error diagnostics and a LogLoader tool that collects error diagnostic logs for analysis.
ORACLE_HOME/j2ee/home/oc4j-api.jar	Contains the public OC4J APIs.
ORACLE_HOME/j2ee/home/oc4jclient.jar	Contains the files needed by the OC4J client.
ORACLE_HOME/j2ee/home/lib/activation.jar	Usually, this component is available in the JRE. If it is not, then include it in the classpath. This JAR is needed only for processing attachments.
ORACLE_HOME/j2ee/home/lib/adminclient.jar	Contains J2EE APIs for platform management and application deployment.
OC4J_HOME/j2ee/home/lib/ejb.jar	Contains class files for Enterprise Java Beans.
OC4J_HOME/j2ee/home/lib/http_client.jar	Contains the Oracle HTTP client transport implementation.
OC4J_HOME/j2ee/home/lib/javax77.jar	Contains the J2EE management specification (JSR 77) APIs.
OC4J_HOME/j2ee/home/lib/jax-qname-namespace.jar	Contains the QName definition.
OC4J_HOME/j2ee/home/lib/jmxri.xml	Contains the JMX APIs.
OC4J_HOME/j2ee/home/lib/jmx_remote_api.jar	Contains the JMX Remote APIs.
OC4J_HOME/j2ee/home/lib/mail.jar	Contains the JavaMail API and all service providers. Usually, this component is available in the JRE. If it is not, then include it in the classpath.

Table A–2 (Cont.) Classpath Components for a Client Using a Client-Side Proxy

Component JAR	Description
OC4J_HOME/j2ee/home/lib/oc4j-schemas.jar	Contains the OracleAS Web Services public schemas.
OC4J_HOME/j2ee/home/lib/servlet.jar	Contains the servlet implementation.
OC4J_HOME/lib/dms.jar	Contains the DMS implementation for Oracle Diagnostics and Monitoring.
OC4J_HOME/lib/xml.jar	Contains the JAXB implementation.
OC4J_HOME/lib/xmlparserv2.jar	Contains the Oracle XML parser JAR.
OC4J_HOME/lib/xsu12.jar	Contains the Oracle XDK SQL utility.
OC4J_HOME/webservices/lib/commons-logging.jar	Contains a logging library package.
OC4J_HOME/webservices/lib/jaxrpc-api.jar	Contains the JAX-RPC API, including `javax.xml.rpc` and `java.xml.namespace`.
OC4J_HOME/webservices/lib/orasaaj.jar	Contains the Oracle implementation of the SAAJ API.
OC4J_HOME/webservices/lib/orawsdl.jar	Contains the Oracle implementation (OraWSDL) of the Java APIs for the WSDL specification (JSR 110).
OC4J_HOME/webservices/lib/relaxngDatatype.jar	Contains the RELAX NG data type library shared by several JAX* technologies.
OC4J_HOME/webservices/lib/saaj-api.jar	Contains the SAAJ API 1.1 for processing messages with attachments.
OC4J_HOME/webservices/lib/wsclient.jar	Contains the classes that are required by the Web service client runtime.
OC4J_HOME/webservices/lib/wsdl-api.jar	Contains the Java APIs for WSDL (JSR 110).
OC4J_HOME/webservices/lib/wsif.jar	Contains the Oracle WSIF implementation.
OC4J_HOME/webservices/lib/xsdlib.jar	Contains the XML Schema type library shared by several JAX* technologies.

OC4J Security-Related Client JAR Files

Table A–3 lists the JAR files that must be included in the classpath when a J2SE Web service client supports OC4J Security. The `wsclient_extended.jar` file contains the OC4J Security-related class files in the JARs listed in this table. There is no need to include these JARs if you run the client with the `wsclient_extended.jar` file.

Table A–3 OC4J Security CLASSPATH Components for a Client Using a Client-Side Proxy

Component Name	Description
OC4J_HOME/j2ee/home/lib/jta.jar	Contains JTA (Java Transaction API) Specification APIs.
OC4J_HOME/j2ee/home/lib/jaas.jar	Contains the Java Authorization and Authentication Specification (JAAS) APIs.
OC4J_HOME/j2ee/home/lib/jazn.jar	Contains the JAZN (Oracle JAAS provider) implementation.
OC4J_HOME/j2ee/home/lib/jazncore.jar	Contains the JAZN (Oracle JAAS provider) implementation.

WS-Security-Related Client JAR Files

Table A–4 lists the JAR files that must be included in the classpath when a J2SE Web service client supports WS-Security. The `wsclient_extended.jar` file contains the WS-Security-related class files in the JARs listed in this table. There is no need to include these JARs if you run the client with the `wsclient_extended.jar` file.

Table A–4 WS-Security CLASSPATH Components for a Client Using a Client-Side Proxy

Component Name	Description
OC4J_HOME/jlib/javax-ssl-1_1.jar	Contains transport-level security support.
OC4J_HOME/jlib/jaxen.jar	Contains the classes that define Jaxen—a Java XPath Engine capable of evaluating XPath expressions across multiple modes (such as dom4j, JDOM, and so on).
OC4J_HOME/jlib/ojpse.jar	Contains the core encryption implementation.
OC4J_HOME/jlib/oraclepki.jar	Contains the Oracle orapki keytool utility.
OC4J_HOME/jlib/osdt_core.jar	Contains the Oracle Security Developer's Toolkit (OSDT) APIs.
OC4J_HOME/jlib/osdt_cert.jar	Contains the Oracle Security Developer's Toolkit cryptography APIs.
OC4J_HOME/jlib/osdt_saml.jar	Contains Oracle Security Developer's Toolkit Security Assertion Markup Language (SAML) APIs.
OC4J_HOME/jlib/osdt_wss.jar	Contains Oracle Security Developer's Toolkit Web services security (WS-Security) APIs.
OC4J_HOME/jlib/osdt_xmlsec.jar	Contains Oracle Security Developer's Toolkit XML signing and encryption APIs.
OC4J_HOME/webservices/lib/wssecurity.jar	Contains the WS-Security APIs.

Reliability-Related Client JAR File

Table A–5 lists the JAR files that must be included in the classpath when a J2SE Web service client supports WS-Reliability. The `wsclient_extended.jar` file contains the WS-Reliability-related class files in the JAR listed in this table. There is no need to include this JAR if you run the client with the `wsclient_extended.jar` file.

Table A–5 Reliability CLASSPATH Components for a Client Using a Client-Side Proxy

Component Name	Description
OC4J_HOME/webservices/lib/orawsrm.jar	Contains Web service Reliability (WS-Reliability) APIs and their implementation.

JMS Transport-Related Client JAR File

Table A–6 lists the JAR file that must be included in the classpath when a J2SE Web service client supports JMS as a transport mechanism. The `wsclient_extended.jar` file contains the JMS transport-related class files in the JAR listed in this table. There is no need to include this JAR if you run the client with the `wsclient_extended.jar` file.

Table A–6 CLASSPATH Components for a Client Using JMS as a Transport Mechanism

Component Name	Description
J2EE_HOME/j2ee/home/lib/jms.jar	Contains the JMS APIs.

Database Web Services-Related Client JAR Files

The JAR files listed in Table A–7 must be included in the classpath when a J2SE Web service client wants to invoke a service that uses parameters or returns in SQL/XML format, or are of type `WebRowSet`. The client must also include these libraries when invoking a database Web service using a native WSIF binding.

Note: The class files in the JARs listed in this table are **not** included in the `wsclient_extended.jar` file. If the client needs the functionality provided by these files, then they must be listed explicitly on the classpath.

Table A–7 Database-Related CLASSPATH Components for a Client Using a Client-Side Proxy

Component Name	Description
ORACLE_HOME/rdbms/jlib/aqapi.jar	Contains the Oracle Advanced Queueing API
ORACLE_HOME/jdbc/lib/ocrs12.jar	Contains the Oracle WebRowSet implementation
ORACLE_HOME/jdbc/lib/ojdbc14dms.jar	Contains the Oracle JDBC driver
OC4J_HOME/lib/xsu12.jar	Contains the Oracle SQL/XML format implementation

Sample Classpath Commands

Example A–1 provides sample Windows platform `set CLASSPATH` commands for all of the Oracle Application Server Web Services client JAR files. The classpath on the UNIX platform would be set in a similar manner.

Example A–1 set CLASSPATH Commands for the Windows Platform

```
set CLASSPATH=%ORACLE_HOME%\j2ee\home\oc4jclient.jar;%CLASSPATH%
set CLASSPATH=%ORACLE_HOME%\j2ee\home\lib\adminclient.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\j2ee\home\lib\ejb.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\j2ee\home\lib\mail.jar;%CLASSPATH%
set CLASSPATH=%ORACLE_HOME%\j2ee\home\lib\activation.jar;%CLASSPATH%
set CLASSPATH=%J2EE_HOME%\j2ee\home\lib\jms.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\j2ee\home\lib\http_client.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\j2ee\home\lib\jaas.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\j2ee\home\lib\jazn.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\j2ee\home\lib\javax77.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\j2ee\home\lib\jax-qname-namespace.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\j2ee\home\lib\jta.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\j2ee\home\lib\jazncore.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\j2ee\home\lib\servlet.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\j2ee\home\lib\jmx_remote_api.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\j2ee\home\lib\jmxri.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\j2ee\home\lib\oc4j-schemas.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\lib\xmlparserv2.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\xdk\lib\xml.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\webservices\lib\orawsrm.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\webservices\lib\xsdlib.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\webservices\lib\relaxngDatatype.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\lib\dms.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\webservices\lib\commons-logging.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\webservices\lib\jaxrpc-api.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\webservices\lib\wsclient.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\webservices\lib\wsdl-api.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\webservices\lib\wsif.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\webservices\lib\wssecurity.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\webservices\lib\saaj-api.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\webservices\lib\orasaaj.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\webservices\lib\orawsdl.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\jlib\osdt_wss.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\jlib\osdt_xmlsec.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\jlib\ojpse.jar;%CLASSPATH%
```

```
set CLASSPATH=%OC4J_HOME%\jlib\osdt_saml.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\jlib\oraclepki.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\jlib\osdt_core.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\jlib\osdt_cert.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\jlib\jaxen.jar;%CLASSPATH%
set CLASSPATH=%OC4J_HOME%\jlib\javax-ssl-1_1.jar;%CLASSPATH%
```

B

Oracle Implementation of the WSDL 1.1 API

This appendix describes the Oracle implementation of the Java APIs for WSDL version 1.1 (OraWSDL). These APIs allow you to read, modify, write, create, and re-organize WSDL documents in memory.

The key Java API for WSDL is the `javax.wsdl.factory.WSDLFactory` class. This abstract class enables applications to obtain a WSDL factory capable of producing new definitions, new `WSDLReaders`, and new `WSDLWriters`.

See Also:

The Java APIs for WSDL version 1.1 at the following Web site:

`http://www.jcp.org/en/jsr/detail?id=110`

Understanding the OraWSDL APIs

The OracleAS Web Services implementation of the `javax.wsdl.factory.WSDLFactory` class is `oracle.webservices.wsdl.WSDLFactoryImpl`. Use this class to create new WSDL factory instances. As the abstract `WSDLFactory` class requires, OracleAS Web Services provides extensions to support XMLSchema, SOAP, HTTP, and MIME.

The `WSDL_READ_TIMEOUT` property can be used only with the HTTP or HTTPS protocols. This property specifies the maximum amount of time, in seconds, that the `WSDLReader` implementation waits to receive a response to a request for a remote WSDL definition.

Here are a few examples of where you can use the functionality provided by the `WSDLFactory` class.

- writing a WSDL manipulation tool
- programmatically analyzing a WSDL
- writing WSIF provider

Example B–1 provides sample code that uses the Java APIs for WSDL to get and manipulate a WSDL file. The samples illustrate the following tasks.

- getting a WSDL factory instance
- creating a WSDL file reader and registering the standard extensions
- setting a timeout for reading the WSDL definition
- reading a WSDL file
- getting information from a WSDL file

In the lines of code that get a WSDL factory, note that the `WSDLFactory.newInstance` method uses the Oracle `WSDLFactoryImpl` class to create the WSDL factory instance.

Example B–1 Using the Java APIs for WSDL to Manipulate a WSDL

```
   ...
//--Get the WSDLFactory. You must specifiy Oracle's implementation class name
       WSDLFactory wsdlFactory = WSDLFactory.newInstance("oracle.webservices.wsdl.
WSDLFactoryImpl");
   ...
//--Create a reader and register the standard extensions
       WSDLReader wsdlReader = wsdlFactory.newWSDLReader();
       ExtensionRegistry extensionRegistry = wsdlFactory.
newPopulatedExtensionRegistry();
       wsdlReader.setExtensionRegistry(extensionRegistry);
   ...
 //--Set a sixty-second timeout for reading the WSDL definition
        System.setProperty(oracle.webservices.wsdl.WSDLFactoryImpl.WSDL_READ_
TIMEOUT, "60" );
   ...
//--Read a WSDL file, including any imports
       Definition def = wsdlReader.readWSDL("http://some.com/someservice?WSDL");
   ...
//--You can now explore the WSDL definition, for example,
       Map services = def.getServices();
       String targetNamespace = def.getTargetNamespace();
   ...
```

The code in Example B–2 demonstrates how you can get the WSDL by using methods in the `javax.xml.rpc.server.ServletEndpointContext` and `javax.servlet.ServletContext` classes. The WSDL is obtained and stored as an `InputStream`. This technique works not only for the WSDL, but for any resource.

The `// code to handle the input stream` section indicates where you can insert OraWSDL API code similar to the samples in Example B–1 to get and manipulate the WSDL file.

Example B–2 Getting the WSDL as a Resource

```
public class HelloImpl  implements javax.xml.rpc.server.ServiceLifecycle {
    // default constructor
    public HelloImpl() {
}
    public init(Object context){
        javax.xml.rpc.server.ServletEndpointContext ctx = (javax.xml.rpc.server.
ServletEndpointContext) context;
        javax.servlet.ServletContext servletContext= ctx.getServletContext();
        //we can read any other resource the same way.
        java.io.InputStream wsdlDocument = servletContext.
getResourceAsStream("/WEB-INF/wsdl/MyWsdl.wsdl")) ;

       // code to handle the input stream
      ...
    }

   //empty implementation
    public void destroy() {}

    // sayHello method
```

```
    public String sayHello(String name) {
        return ("Hello " + name + "!");
    }
}
```

C

oracle-webservices.xml Deployment Descriptor Schema

The `oracle-webservices.xml` deployment descriptor is used in conjunction with the standard `webservices.xml`. It contains deployment and run-time information that is specific to OracleAS Web Services. For example, it contains the context-URI, the Web service end-point address, and so on. The values of some elements in the file can be changed by WebServicesAssembler command line arguments and Ant tasks.

All of the elements in `oracle-webservices.xml` are optional. If the file is not provided, the application will still deploy and run, with appropriate default values for the unspecified elements.

The `oracle-webservices.xml` deployment descriptor contains the Web services management information for security, reliability, auditing, and logging. On deployment, the file is parsed and its object representation is cached. The Web service management information is extracted from the file and saved as `wsmgmt.xml` in the OC4J container.

Although you could manually create `oracle-webservices.xml` by examining the schema, you will typically use the version of the file created by WebServicesAssembler. You can edit this file to produce the functionality you want. The schema is defined by the `oracle-webservices-10_0.xsd` file and can be found in the `oc4j-schemas.jar` file.

OC4J_HOME`\j2ee\home\lib\oc4j-schemas.jar`

Here, *OC4J_HOME* represents the directory where you installed Oracle Containers for J2EE (OC4J).

See Also:

- "Arguments that Affect Deployment Descriptor Contents" on page 19-9 for a list of these arguments.
- "Understanding the Web Services Management Schema" in the *Oracle Application Server Advanced Web Services Developer's Guide*. for a description of the management configuration elements

Hierarchy of XML Elements in oracle-webservices.xml

Here is an overview of the element hierarchy for the `oracle-webservices.xml` file.

```
<oracle-webservices>
   <web-site>
   <context-root>
   <webservice-description>
```

```
      <expose-wsdl>
      <expose-testpage>
      <resolve-relative-imports>
      <download-external-imports>
      <port-component>
         <endpoint-address-uri>
         <ejb-transport-security-constraint>
            <wsdl-url>
            <soap-port>
            <role-name>
            <transport-guarantee>
<!-- For a description of transport-level security constraints for EJBs, see "Securing EJB-Based Web Services at
the Transport Level" on page C-15. -->
         <implementor>
            <param>
         <runtime>
            <owsm>
<!-- For a description of the Oracle Web Services Manager (<owsm>) element, see the Oracle Web Services Manager Administrator's
Guide -->
            <security>
<!-- For a description and listing of security elements, see the Oracle Application Server Web Services Security Guide. -->
            <reliability>
               <repository>
 <!-- For a description and listing of port-level reliability elements, see "Port-Level Reliability Elements on the Server" in the Oracle
Application Server Advanced Web Services Developer's Guide. -->
            <logging>
         <operations>
            <operation>
               <runtime>
                  <security>
<!-- For a description and listing of security elements, see the Oracle Application Server Web Services Security Guide. -->
                   <reliability>
                     <duplication-elimination-required>
                     <guaranteed-delivery-required>
<!-- For a listing and description of operation-level reliability elements, see "Operation Level Reliability Elements on the Server" in the
Oracle Application Server Advanced Web Services Developer's Guide. -->
                  <auditing>
<!-- For a listing and description of operation-level auditing elements, see "Server-Side Auditing Configuration Elements" in the Oracle
Application Server Advanced Web Services Developer's Guide. -->
                  <logging>
<!-- For a listing and description of operation-level logging elements, see "Operation Level Logging Elements on the Server" in the
Oracle Application Server Advanced Web Services Developer's Guide. -->
   <provider-description>
      <provider-description-name>
      <wsdl-file>
      <wsdl-service-name>
      <property>
      <provider-port>
         <provider-name>
         <wsdl-port>
         <expose-testpage>
         <expose-wsdl>
         <implementation-class>
         <servlet-link>
         <max-request-size>
         <property>
         <policy>
            <runtime>
            <operations>
               <operation>
```

```
<!-- Here you can enter a management configuration for a particular operation -->
   <ejb-transport-login-config>
      <auth-method>
      <realm-name>
<!-- For a listing and description of elements that define transport-level  security for EJBs, see "Securing EJB-Based Web
Services at the Transport Level" on page C-15. -->
```

Elements and Attributes of oracle-webservices.xml

This section is an alphabetical dictionary of elements in the oracle-webservices.xml file. See the preceding section, "Hierarchy of XML Elements in oracle-webservices.xml" on page C-1, if you are interested in the hierarchy. See "oracle-webservices.xml File Listing" on page C-16 if you are interested in seeing a sample file.

<auth-method>

Parent element: <ejb-transport-login-config>

Child element: None

Required? Optional

This subelement is used to configure the authentication mechanism for an EJB application. As a prerequisite to gaining access to any Web resources which are protected by an authorization constraint, a user must have authenticated using the configured mechanism. Legal values for this subelement are `BASIC`, `DIGEST`, `CLIENT-CERT`, or a vendor-specific single sign-on authentication scheme. See "Securing EJB-Based Web Services at the Transport Level" on page C-15.

<context-root>

Parent element: <oracle-webservices>

Child element: None

Required? Only for version 2.1 EJB Web services

The value of this element is type `string`. Default: the EJB archive file name without the `.jar` extension.

This subelement specifies the root context of the exposed Web service. It is required only for a version 2.1 EJB exposed as a Web service. If `context-root` is not specified, it defaults to the EJB archive file name without the `.jar` extension. For example, if the EJB archive file is named `foo-ejb.jar`, then the context root will be `/foo-ejb`.

For Java-class Web services, the context root is specified inside `application.xml`.

The *Oracle Containers for J2EE Configuration and Administration Guide* provides more information on the `<context-root>` element.

<download-external-imports>

Parent element: <webservice-description>

Child element: None

Required? Optional

This boolean element specifies whether the relative imports should be downloaded and resolved to absolute URLs. Default is `false`.

Note: If `download-external-imports` is set to `true`, then `resolve-relative-imports` is automatically set to `true`.

<ejb-transport-login-config>

Parent element: <oracle-webservices>

Child elements: <auth-method>, <realm-name>

Required? Optional

This element is used to configure the transport-level authentication method and the realm name that should be used for this EJB application. The URL of the EJB application exposed as a Web service is indicated by the `<endpoint-address-uri>` element in the port component. For more information, see "Securing EJB-Based Web Services at the Transport Level" on page C-15.

<ejb-transport-security-constraint>

Parent element: <port-component>

Child elements: <role-name>, <soap-port>, <transport-guarantee>, <wsdl-url>

Required? Optional

Defines transport-level security for a Web service based on EJBs. See "Securing EJB-Based Web Services at the Transport Level" on page C-15 for more information on how to use this element.

<endpoint-address-uri>

Parent element: <port-component>

Child element: None

Required? Only for Web services based on version 2.1 EJBs

This subelement, needed only for an EJB 2.1 Web service, specifies the sub-context of the HTTP URL at which this EJB is exposed as a Web service. If none is provided, it defaults to the `port-component` name. Two transport-level elements are provided to secure this URI. For more information on these elements, see "<ejb-transport-security-constraint>" and "<ejb-transport-login-config>". See also, "Securing EJB-Based Web Services at the Transport Level" on page C-15.

For a Web module, (Web services derived from Java classes), this information is already present in the `web.xml` file and is not required.

<expose-testpage>

Parent element: <webservice-description>

Child element: None

Required? Optional

This boolean element specifies whether the test-page should be exposed. Default is `true`.

Parent element: <provider-port>

Child elements: None

Required? Optional

This boolean element specifies whether the test-page should be exposed. Default is `true`.

<expose-wsdl>

Parent element: <webservice-description>

Child element: None

Required? Optional

This boolean element specifies whether the WSDL should be exposed. Default is `true`.

Parent element: <provider-port>

Child elements: None

Required? Optional

This boolean element specifies whether the WSDL should be exposed. Default is `true`.

<implementation-class>

Parent element: <provider-port>

Child elements: None

Required? Required

Specifies the name of the class implementing the `oracle.webservices.provider.Provider` interface.

<implementor>

Parent element: <port-component>

Child element: <param>

Required? Optional

This subelement captures information that is generated at deployment time about OC4J's proprietary Web services. This subelement has a `param` subelement.

Table C–1 <implementor> Attributes

Name	Description
type	Value: implementorTypeAttr Default: n/a (Required) For example, database.

<jms-address>

Parent element: <port-component>

Child element: None

Required? Optional

The address element used for providing JMS destination information. This is used when JMS transport is specified to send SOAP messages over JMS.

Table C–2 <jms-address> Attributes

Name	Description
jndiConnectionFactoryName	Value: string Default: n/a (Required) 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.
jndiDestinationName	Value: string Default: n/a (Required) 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.

<max-request-size>

Parent element: <port-component>

Child element: None

Required? Optional

This element enables you to configure a maximum size, in bytes, for a message passed to the Web service. If the Web service reads a message that passes this number of bytes, then the transmission will fail and the connection will close.

The value of the element is a `long` integer. If a non-positive value is assigned, then it is assumed there is no limit. The default, `-1` means unlimited. There is no limit as to the size.

Parent element: <provider-port>

Child elements: None

Required? 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.

<operation>

Parent element: <operations>

Child elements: <runtime>

Required? Optional

Specifies a quality of service configuration for a particular operation provided by the referenced Web service. The configuration appears within this element's `<runtime>` subelement.

Table C–3 <operation> Attributes

Name	Description
input	Value: string Default: n/a Contains the input name of the operation from the WSDL.
name	Value: string Default: n/a (Required) associates the contained quality of service configuration to a specific operation. The value of the attribute must match the operation name from the WSDL.
output	Value: string Default: n/a Contains the output name of the operation from the WSDL.

<operations>

Parent element: <policy>

Child elements: <operation>

Required? Optional

Contains a sequence of elements, one for each operation. The `<operation>` subelement describes the management policies applied to an individual operation.

<oracle-webservices>

Parent element: n/a (root)

Child elements: <context-root>, <ejb-transport-login-config>, <web-site>, <webservice-description>

Required? Required

The <oracle-webservices> element captures information local to the Oracle container for Web services.

Table C–4 <oracle-webservices> Attributes

Name	Description
noNamespaceSchemaLocation	Value: URI Default: n/a This attribute is the "standard" way of telling the parser which schema the XML document should adhere to.

<param>

Parent element: <port-component>

Child element: None

Required? Optional

Specifies the JNDI location of the data source at runtime.

<policy>

Parent element: <provider-port>

Child elements: None

Required? Optional

Defines the Web service management policies for the Provider.

<port-component>

Parent element: <webservice-description>

Child elements: <ejb-transport-security-constraint>, <endpoint-address-uri>, <implementor>, <jms-address>, <max-request-size>, <rest-support>, <runtime>, <use-dime-encoding>

Required? Optional

The <port-component> element is used as a reference to map to similar elements in the standard deployment descriptor webservices.xml with a -name appended to them. The elements contain information pertaining to a particular port.

Table C–5 <port-component> Attributes

Name	Description
name	Value: string Default: n/a (Required) Maps to the name element in webservices.xml.

<property>

Parent element: <provider-description>

Child elements: None

Required? Optional

Specifies globally-defined properties; that is, properties that are available to all of the defined Provider ports. This element has a required string `name` attribute.

Parent element: <provider-port>

Child elements: None

Required? 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

<provider-description>

Parent element: <oracle-webservices>

Child elements: <property>, <provider-description-name>, <provider-port>, <wsdl-file>, <wsdl-service-name>

Required? Optional

This element serves to make the Web service Provider-aware, and identifies a collection of Provider ports. For more information on Web service Providers, see "Using Web Service Providers" in the *Oracle Application Server Advanced Web Services Developer's Guide*

<provider-description-name>

Parent element: <provider-description>

Child elements: None

Required? Required

This string element specifies a name for the Provider description.

<provider-name>

Parent element: <provider-port>

Child elements: None

Required? Required

The name of the Provider. The string `<provider-name>` within the `<provider-port>` must be unique within the deployment descriptor.

<provider-port>

Parent element: <provider-description>

Child elements: <expose-testpage>, <expose-wsdl>, <implementation-class>, <max-request-size>, <policy>, <property>, <provider-name>, <servlet-link>, <wsdl-port>

Required? 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.

<realm-name>

Parent element: <ejb-transport-login-config>

Child element: None

Required? Optional

This subelement specifies the realm name to use in HTTP Basic authorization for an EJB exposed as Web services. For more information, see "Securing EJB-Based Web Services at the Transport Level" on page C-15.

<resolve-relative-imports>

Parent element: <webservice-description>

Child element: None

Required? Optional

This boolean element specifies whether you want to resolve from relative imports to absolute URLs. Default is `false`.

<rest-support>

Parent element: <port-component>

Child elements: None

Required? Optional

The boolean rest element indicates whether this port supports REST-style `GET` and `POST` requests and responses. Default is `false`.

<role-name>

Parent element: <ejb-transport-security-constraint>

Child element: None

Required? Optional

This `string` element identifies the name of a security role. The name must conform to the lexical rules for a token. A default value is not defined.

The `role-name` used here must correspond to either:

- the role name of one of the security role elements defined for this EJB application, or
- the reserved role-name "`*`" that indicates all roles in the EJB application.

If both "`*`" and role names are entered in this subelement, the container interprets this as all roles. If no roles are defined, then no user is allowed access to the portion of the Web application described by the containing security constraint. The container matches role names in a case-sensitive manner.

See "Securing EJB-Based Web Services at the Transport Level" on page C-15 for more information on how to use this element.

<runtime>

Parent element: <port-component>

Child element: None

Required? Optional

Denotes the start of the Web services management information that will be read at runtime. For more information, see "Understanding the Web Services Management Schema" in the *Oracle Application Server Advanced Web Services Developer's Guide*.

Parent element: <policy>

Child elements: None

Required? Optional

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.

<servlet-link>

Parent element: <provider-port>

Child elements: None

Required? Required

Associates a `provider-port` value with a servlet endpoint defined in the `/WEB-INF/web.xml` file.

<soap-port>

Parent element: <ejb-transport-security-constraint>

Child element: None

Required? Optional

If present, this element specifies that the security constraints must apply only to the SOAP port. A default is not defined. See "Securing EJB-Based Web Services at the Transport Level" on page C-15 for more information on how to use this element.

<transport-guarantee>

Parent element: <ejb-transport-security-constraint>

Child element: None

Required? Optional

This element specifies constraints on the access to data transmitted between the client and server. This subelement can have one of the following values:

- `NONE`—the application does not require any transport guarantees.
- `INTEGRAL`—the application requires that the data sent between the client and server must be sent in such a way that it cannot be changed in transit.
- `CONFIDENTIAL`—the application requires that the data be transmitted in a fashion that prevents other entities from observing the contents of the transmission.

In most cases, the presence of the `INTEGRAL` or `CONFIDENTIAL` flag will indicate that SSL must be used.

See "Securing EJB-Based Web Services at the Transport Level" on page C-15 for more information on how to use this element.

<use-dime-encoding>

Parent element: <port-component>

Child element: None

Required? Optional

If set to `true`, any SOAP responses returned from this service that have attachments will be encoded in DIME. If `false` (default), any SOAP responses with attachments will be returned in MIME encoding. See "Working with DIME Attachments" in the *Oracle Application Server Advanced Web Services Developer's Guide* for more information on DIME-encoded attachments.

<web-site>

Parent element: <oracle-webservices>

Child elements: None

Required: Optional

Default: the location where the service endpoint interface is installed

This optional element provides a name for the host and the port name which will be substituted inside the updated WSDL port location.

For example, you might need to enter this subelement in the `oracle-webservices.xml` file if you are accessing the Web service by `port-component-link` resolution, or if you are publishing the WSDL to a location specified by `wsdl-publish-location`.

If this element is not used, the host and port values of the HTTP request used to get the WSDL will be substituted.

Table C–6 <web-site> Attributes

Name	Description
host	Value: string Default: n/a (required) The name for the host which will be substituted inside the updated WSDL port location.
port	Value: string Default: n/a (required) The name for the port which will be substituted inside the updated WSDL port location

<webservice-description>

Parent element: <oracle-webservices>

Child elements: <download-external-imports>, <expose-testpage>, <expose-wsdl>, <port-component>, <resolve-relative-imports>, <wsdl-file>, <wsdl-publish-location>

Required? Optional

This element extends the `<webservice-description>` element in the standard deployment descriptor, `webservices.xml`.

Table C–7 <webservice-description> Attributes

Name	Description
name	Value: string Default: n/a (required) Maps to the `name` element in `webservices.xml`.

<wsdl-file>

Parent element: <provider-description>

Child elements: None

Required? Optional

This string element specifies the location of an associated WSDL for the Provider.

Parent element: <webservice-description>

Child element: None

Required? Optional

This element of type `wsdl-file` is generated at deployment time and cannot be specified by an end-user. It specifies the location of the final updated WSDL.

Table C–8 <wsdl-file> Attributes

Name	Description
final-location	Value: anyUri Default: n/a (Required) Specifies the location of the final updated WSDL.

<wsdl-port>

Parent element: <provider-port>

Child elements: None

Required? 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.

<wsdl-publish-location>

Parent element: <webservice-description>

Child element: None

Required? Optional

This element is used to specify the location where the final WSDL and its dependent files (imports) can be placed. It is of type anyUri and the value should be of the form `file:/`*location*`/`. A default is not defined.

<wsdl-service-name>

Parent element: <provider-description>

Child elements: None

Required? 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

<wsdl-url>

Parent element: <ejb-transport-security-constraint>

Child elements: None

Required? Optional

If present, this element specifies that the security constraints must apply only to the WSDL URL. A default value is not defined. See "Securing EJB-Based Web Services at the Transport Level" on page C-15 for more information on how to use this element.

Securing EJB-Based Web Services at the Transport Level

The `oracle-webservices.xml` deployment descriptor provides two elements, `<ejb-transport-security-constraint>` and `<ejb-transport-login-config>` that allow you to define transport-level security for Web services that are based on EJBs.

See Also:

"Adding Transport-level Security for Web Services Based on EJBs" and "Accessing Web Services Secured on the Transport Level" in the *Oracle Application Server Web Services Security Guide* for more information on this topic.

The `<ejb-transport-security-constraint>` element is used to associate transport-level security constraints for a version 2.1 EJB exposed as Web service. The URL of the EJB exposed as a Web service is indicated by the `<endpoint-address-uri>` element in the port component. The `<ejb-transport-security-constraint>` element has the following subelements.

- `<role-name>`—Identifies the name of a security role. For more information, see "<role-name>" on page C-10.
- `<soap-port>`—Specifies that the security constraints must apply only to the SOAP port. For more information, see "<soap-port>" on page C-11
- `<transport-guarantee>`—Specifies constraints on the access to data transmitted between the client and server. For more information, see "<transport-guarantee>" on page C-12.
- `<wsdl-url>`—Specifies that the security constraints must apply only to the WSDL URL. For more information, see "<wsdl-url>" on page C-14.

The subelements `<wsdl-url>` and `<soap-port>` are identifiers that let you choose whether the security constraints will apply to a WSDL URL or to a SOAP port. If `<wsdl-url>` and `<soap-port>` are both present or both absent in `<ejb-transport-security-constraint>` then the security constraints will apply to both the WSDL and the SOAP port.

The `<ejb-transport-login-config>` element is used to configure the transport-level authentication method and the realm name that should be used for this EJB application. The URL of the EJB application exposed as a Web service is indicated by the `<endpoint-address-uri>` element in the port component.

- `<auth-method>`—Configures the authentication mechanism for an EJB application. For more information, see "<auth-method>" on page C-3.
- `<realm-name>`—Specifies the realm name to use in HTTP Basic authorization for an EJB exposed as Web services. For more information, see "<realm-name>" on page C-10.

oracle-webservices.xml File Listing

Example C–1 provides a listing of a template for a `oracle-webservices.xml` deployment descriptor. Note that this template file also includes the Web services management elements. These elements are described in the sections indicated in the file.

Example C–1 Sample oracle-webservices.xml File

```
<?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-webs
ervices-10_0.xsd"
   deployment-version="String" deployment-time="String" schema-major-version="10"
   schema-minor-version="0">
   <web-site host="String" port="String"/>
   <context-root>String</context-root>
   <webservice-description name="String">
      <expose-wsdl>true</expose-wsdl>
      <expose-testpage>true</expose-testpage>
      <resolve-relative-imports>false</resolve-relative-imports>
      <download-external-imports>false</download-external-imports>
      <port-component name="String">
         <endpoint-address-uri>String</endpoint-address-uri>
         <ejb-transport-security-constraint>
            <wsdl-url/>
            <soap-port/>
            <role-name>Manager</role-name>
            <role-name>Administrator</role-name>
            <transport-guarantee>NONE</transport-guarantee>
         </ejb-transport-security-constraint>
<!-- For a listing and description of elements that define transport-level security
constraints for EJBs, see "Securing EJB-Based Web Services at the Transport Level" on
page C-15. -->
         <implementor type="database">
            <param name="String">String</param>
         </implementor>
         <runtime enabled="String">
            <owsm/>
<!-- For a description of the element for the Oracle Web Services Manager (<owsm>)
see the Oracle Web Services Manager Administrator's Guide -->
            <security>
<!-- For a description and listing of security elements, see the Oracle Application Server
Web Services Security Guide. -->
            </security>
            <reliability>
               <repository jndiLocation="..." name="..." type="..."/>
<!-- For a description and listing of port-level reliability elements, see "Port-Level
Reliability Elements on the Server" in the Oracle Application Server Advanced Web
Services Developer's Guide. -->
            </reliability>
            <logging/>
         </runtime>
         <operations>
            <operation name="String" input="String">
               <runtime>
```

```
                <security>
```
<!-- For a description and listing of security elements, see the *Oracle Application Server Web Services Security Guide*. -->

```
                </security>
                <reliability>
                   <duplication-elimination-required/>
                   <guaranteed-delivery-required/>
```
<!-- For a listing and description of operation-level reliability elements, see "Operation Level Reliability Elements on the Server" in the *Oracle Application Server Advanced Web Services Developer's Guide*. -->

```
                </reliability>
                <auditing request="false" response="false" fault="false"/>
```
<!-- For a listing and description of operation-level auditing elements, see "Server-Side Auditing Configuration Elements" in the *Oracle Application Server Advanced Web Services Developer's Guide*. -->

```
                <logging>
```
<!-- For a listing and description of operation-level logging elements, see "Operation Level Logging Elements on the Server" in the *Oracle Application Server Advanced Web Services Developer's Guide*. -->

```
                </logging>
             </runtime>
          </operation>
       </operations>
    </port-component>
 </webservice-description>
 <ejb-transport-login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>sec-ejb</realm-name>
 </ejb-transport-login-config>
```
<!-- For a listing and description of elements that define transport-level security for EJBs, see "Securing EJB-Based Web Services at the Transport Level" on page C-15 . -->

```
</oracle-webservices>
```

D

service-ref-mapping Schema

This appendix describes the `service-ref-mapping` schema. This schema is defined by the `service-ref-mapping-10_0.xsd` file and can be found in the `oc4j-schemas.jar` file.

OC4J_HOME`\j2ee\home\lib\oc4j-schemas.jar`

This schema defines elements that define OC4J-specific runtime and deployment-time generated settings for a Web service reference. You can use the elements defined by this schema to configure the following information:

- configure the client to consume stateful Web Services
- configure the client to make JMS transport calls
- configure security, logging, and auditing quality of service (QOS) features for the corresponding Web service.

The `service-ref-mapping-10_0.xsd` is imported into the `orion-web`, `orion-ejb-jar`, and `orion-application-client` XSDs.

The `<service-ref-mapping>` elements can appear as subelements of the `<orion-web-app>` element in the `orion-web.xml`, `orion-ejb-jar.xml`, or `orion-application-client.xml` proprietary deployment descriptor files. Currently, there is no tool support, such as Oracle JDeveloper wizards, for providing values to `service-ref-mapping-10_0.xsd` on adding them to the deployment descriptors. You must refer to the schema and enter the values for `<service-ref-mapping>` elements into the appropriate XML file by hand.

Hierarchy of service-ref-mapping Schema

Here is an overview of the element hierarchy for the `service-ref-mapping` schema.

```
<service-ref-mapping>
  <service-impl-class>
  <wsdl-file>
  <wsdl-location>
  <service-qname>
  <stub-property>
   <name>
   <value>
  <call-property>
   <name>
   <value>
  <port-info>
    <wsdl-port>
```

```
<service-endpoint-interface>
<stub-property>
  <name>
  <value>
<call-property>
  <name>
  <value>
<runtime>
  ...
```

Under <runtime>, you can add client-side quality of service (Web Service Management) features. This can include a reliabilty and security configuration.

```
  ...
<operations>
  <operation>
     <runtime>
       ...
```

Under <runtime>, you can add client-side quality of service (Web Service Management) features. This can include an auditing, reliabilty, and security configuration.

```
          ...
```

Elements and Attributes of service-ref-mapping Schema

This section is an alphabetical dictionary of elements of the service-ref-mapping schema. See the preceding section, "Hierarchy of service-ref-mapping Schema", if you are interested in the hierarchy.

The element descriptions in the section apply to `service-ref-mapping` elements that are inserted into an `orion-web.xml`, `orion-ejb-jar.xml`, or `orion-application-client.xml` proprietary deployment descriptor file.

<call-property>

Parent element: <service-ref-mapping> and <port-info>

Child elements: <name>, <value>

Required? Optional

The `<call-property>` element can be a subelement of both `<service-ref-mapping>` and `<port-info>`. When `<call-property>` is used as a subelement of `<service-ref-mapping>`, it defines the call property values applicable to all ports. This is a convenient way to specify a property without specifying the port name.

When `<call-property>` is used as a subelement of `<port-info>`, it defines the call property values applicable to the specific port defined by the `<port-info>`.

If call property values are specified for a particular port inside the `<port-info>` tag, then they override the `<call-property>` element values set under `<service-ref-mapping>`.

<name>

Parent element: <call-property> and <stub-property>

Child elements: None

Required? Required

The `<name>` element can appear as a subelement of either `<call-property>` or `<stub-property>`. It 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`.

The `<value>` element assigns a value to the property. For more information on this element, see "<value>" on page D-6.

<operation>

Parent element: <operations>

Child elements: <runtime>

Required? Optional

Specifies client-side quality of service configuration for a particular operation provided by the referenced Web service. The configuration appears within this element's `<runtime>` subelement.

Table D–1 <operation> Attributes

Name	Description
inputName	Value: string Default: n/a This optional attribute contains the input name of the operation from the WSDL. It is required only if the `name` attribute cannot be used to uniquely identify the operation.
name	Value: string Default: n/a (Required) Associates the contained quality of service configuration to a specific operation. The value of the attribute must match the operation name from the WSDL.
outputName	Value: string Default: n/a This optional attribute contains the output name of the operation from the WSDL. It is required only if the `name` and `input` attributes cannot be used to uniquely identify the operation.

<operations>

Parent element: <port-info>

Child elements: <operation>

Required? Optional

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.

<port-info>

Parent element: <service-ref-mapping>

Child elements: <call-property>, <operations>, <runtime>, <service-endpoint-interface>, <stub-property>, <wsdl-port>

Required? Optional

Defines a port within a service-reference. This element provides all of the information for a port within a service reference. You can specify either `<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.

This element also contains subelements that let you specify quality of service features that are available for the port and its operations.

<runtime>

Parent element: <port-info> and <operation>

Child elements: None

Required? Optional

The `<runtime>` element can be used as a subelement of the `<port-info>` and `<operation>` elements. When the `<runtime>` element is used as a subelement to `<port-info>`, it 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.

When the `<runtime>` element is used as a subelement to `<operation>`, it contains client-side quality of service configuration for individual operations within the port. Each child element contains configuration for one of the quality of services features (security, reliability, and/or auditing).

<service-endpoint-interface>

Parent element: <port-info>

Child elements: None

Required? Optional

Specifies the fully-qualified path to the service endpoint interface of a WSDL port. The container uses this port for container-managed port selection. The value of this element is a string.

<service-impl-class>

Parent element: <service-ref-mapping>

Child elements: None

Required? Optional

Defines a deployment-time generated name of a `Service` implementation.

<service-qname>

Parent element: <service-ref-mapping>

Child elements: None

Required? Optional

Derived at deployment-time, this element contains the `QName` of the Web service.

<service-ref-mapping>

Parent element: root element

Child elements: <call-property>, <port-info>, <service-impl-class>, <stub-property>, <service-qname>, <wsdl-file>, <wsdl-location>

Required? Optional

The `<service-ref-mapping>` element is used in conjunction with the `<service-ref>` element that appears in the standard deployment descriptors `web.xml`, `ejb-jar.xml`, and `application-client.xml`. The `<service-ref>` element contains the information that lets you employ an EJB, JSP, or servlet as a Web service client that can invoke a remote Web service.

Note that whenever a `<service-ref>` element can appear in a `web.xml`, `ejb-jar.xml`, or `application-client.xml` file, a corresponding `<service-ref-mapping>` element can appear in an `orion-web.xml`, `orion-ejb-jar.xml`, or `orion-application-client.xml` file.

In its simplest case, the `<service-ref-mapping>` element contains only deployment information. Do not add run-time or quality of service elements if you do not want your client to be managed. A managed client is more expensive in terms of performance.

<stub-property>

Parent element: <service-ref-mapping> and <port-info>

Child elements: <name>, <value>

Required? Optional

The `<stub-property>` element can be a subelement of both `<service-ref-mapping>` and `<port-info>`. When `<stub-property>` is used as a subelement of `<service-ref-mapping>`, it defines the stub property values applicable to all ports. This is a convenient way to specify a property without specifying the port name.

When `<stub-property>` is used as a subelement of `<port-info>`, it defines the stub property values applicable to the specific port defined by the `<port-info>`.

If stub property values are specified for a particular port inside the <port-info> tag, then they override the <stub-property> element values set under <service-ref-mapping>.

<value>

Parent element: <call-property> and <stub-property>

Child elements: None

Required? Required

The <value> element can appear as a subelement of either <call-property> or <stub-property>. It 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.

The <name> element assigns the name of the JAX-RPC property. For more information on this element, see "<name>" on page D-2.

<wsdl-file>

Parent element: <service-ref-mapping>

Child elements: None

Required? Optional

Defines a deployment-time generated name for the WSDL file.

Table D–2 <wsdl-file> Attributes

Name	Description
final-location	Value: anyURI Default: n/a (Required) Points to the copy of the WSDL document associated with the service-ref in the standard deployment descriptor.

<wsdl-location>

Parent element: <service-ref-mapping>

Child elements: None

Required? Optional

Contains a valid URL pointing to a WSDL document. If a URL is specified, then the WSDL document at this URL will be used during deployment instead of the WSDL document associated with the service-ref in the standard deployment descriptor. Sample values for <wsdl-location> include:
http://hostname:port/myservice/myport?WSDL and
file:/home/user1/myfinalwsdl.wsdl.

Table D–3 <wsdl-location> Attributes

Name	Description
wsdl-override-last-modified	Value: `string` Default: n/a This optional string value is generated at deployment time and lists the time when the WSDL file was last modified.

<wsdl-port>

Parent element: <port-info>

Child elements: None

Required? Optional

Specifies the name of a port in the WSDL that the container will use for container-managed port selection.

In container-managed port selection, the container manages calls to the instance directly, and the client requests a generic port that might be used to access multiple different instances.

Table D–4 <wsdl-port> Attributes

Name	Description
localpart	Value: string Default: n/a (Required) The local part of a WSDL name. For example, `authenticateHeader`.
namespaceURI	Value: anyURI Default: n/a (Required) The namespace URI of a WSDL. For example, `http://oracle.j2ee.ws/Header`

service-ref-mapping File Listing

For a sample `service-ref-mapping` fragment that can be used in a proprietary deployment descriptor, see "How to Add OC4J-Specific Platform Information for Deployment and Runtime" on page 14-7.

E

Error Message Prefixes

Table E–1 displays the prefixes for error messages that can occur in different feature areas of OracleAS Web Services. The variable *n* represents an integer. Note that message prefixes have changed between the current release and earlier releases.

Table E–1 Error Message Prefixes for OracleAS Web Services Features

Prefixes for Version 10.1.3.1	Prefixes for Versions 10.1.3 and Earlier	OracleAS Web Services Feature Area
OWS-00*nnn*	J2EE-WSM*nnnnn*	Web services common stack messages
OWS-03*nnn*	J2EE-WSC*nnnnn*	Web services client-related messages
OWS-04*nnn*	J2EE-WSS*nnnnn*	Web services server-side messages
OWS-08*nnn*	J2EE-WSD*nnnnn*	Web services deployment-related messages
OWS-11*nnn*	J2EE-WSW*nnnnn*	Oracle implementation of the WSDL 1.1 API (OraWSDL)-related messages
OWS-12*nnn*	J2EE-WSJ*nnnnn*	SOAP with Attachments API for Java (SAAJ)-related messages
OWS-15*nnn*	J2EE-WSIF*nnnnn*	Web Services Invocation Framework (WSIF)-related messages
OWS-20*nnn*	J2EE-WSE*nnnnn*	Web services security-related messages
OWS-25*nnn*	J2EE-WSL*nnnnn*	Web services reliability-related messages
OWS-30*nnn*	J2EE-WSA*nnnnn*	WebServicesAssembler-related messages
OWS-40*nnn*	J2EE-WSR*nnnnn*	JAX-R-related messages

F

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 `Sample` 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; }
}
```

Deserialization Errors Occur when Processing Responses or Requests

This error can occur when OracleAS Web Services attempts to perform schema validation during the deserialization step. The cause of the error is a mismatch between the schema definition found in the `wsdl:types` element and the format of the SOAP payload on the wire.

The error can occur when a client proxy, assembled with `genProxy`, attempts to deserialize a response from the server. It can occur on the server when it attempts to deserialize the request.

The following example illustrates a mismatch between the order of the elements on the wire and in the WSDL. As the deserialization code processes `ParentItemId` on the wire, it skips over the `Quantity` and `PrePick` elements from the schema. Processing fails when encountering `PrePick` on the wire, as this element is not expected after `ParentItemId`.

Wire format:

```
...
<ListOfProduct>
 <Product>
   <ParentItemId></ParentItemId>
   <PrePick>Y</PrePick>
   <Quantity>1</Quantity>
...
```

WSDL definition:

```
...
<xsd:complexType name="Product">
  <xsd:sequence>
    <xsd:element name="Quantity" maxOccurs="1" minOccurs="0"
                 type="xsd:string"></xsd:element>
```

```
    <xsd:element name="PrePick" maxOccurs="1" minOccurs="0"
                 type="xsdLocal11:string1"></xsd:element>
   <xsd:element name="ParentItemId" maxOccurs="1" minOccurs="0"
                type="xsdLocal11:string30"></xsd:element>
...
```

The following is an example deserialization error that can be thrown in this situation:

```
unexpected element name:
expected={http://myCompany.com/Catalog/Data/CategoryProduct}Description,
actual={http://myCompany.com/Catalog/Data/CategoryProduct}PrePick.
at oracle.j2ee.ws.common.util.exception.JAXRPCExceptionBase.<init>(JAXRPCExceptio
nBase.java:93)
at oracle.j2ee.ws.common.util.exception.JAXRPCExceptionBase.<init>(JAXRPCExceptio
nBase.java:111)
at oracle.j2ee.ws.common.encoding.DeserializationException.<init>(Deserialization
Exception.java:54)
at com.myCompany.catalog.data.categoryproduct.runtime.Product_
LiteralSerializer.doDeserialize(Product_Literal Serializer.java:471).
```

The following are possible ways to work around the error:

- Change the service implementation so that the wire format matches the format described in the WSDL.

  You can employ this work around only if you have control over the service implementation and are working in a top down Web service assembly scenario. The mismatch between the schema definition and SOAP payload may have been caused by errors in formatting the response when you implemented the contract. This scenario is possible if you are using the SAAJ APIs (that is, you are either assembling the Web service top down with `dataBinding=false`, or writing custom code using the JAX-RPC handler chain or the Provider APIs).

- Edit the WSDL so that the schema definition matches the wire format.

  You can employ this work around on the server side if you own the service and made a mistake in authoring the XML schema definition. This work around is also possible on the client side when you do not own (or cannot change) the service implementation.

- Generate the Web service with `dataBinding=false` to avoid the XML to Java layer. Every schema type will be represented as a `SOAPElement`. Then, use the SAAJ APIs to write custom code to process the raw XML.

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 `int` 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
```

Get NodeLists by Using getFirstChild and getNextSibling Instead of getChildNode

You may see a performance degradation when iterating over a `NodeList` obtained by using `node.getChildNode`. This degradation will only be significant for `NodeLists` with very long lengths.

Instead of using the `NodeList` obtained by `node.getChildNodes`, the current Oracle XDK implementation offers an optimization of navigating a list of child nodes by using `node.getFirstChild` and looping over `node.getNextSibling`. The following code sample illustrates this technique.

```
Node n = ...;
if (n.hasChildNodes()) {
   for(Node nd=n.getFirstChild(); nd!=null; nd=nd.getNextSibling()){
         nd.getValue(); // do something with nd
      }
}
```

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.

Schema Features Limitations

Schema Features that are Mapped to a SOAPElement

If any of the following schema features are encountered in the WSDL, they will be mapped to a `SOAPElement`.

- any model group with multiple `xsd:any` elements
- `xsd:choice` elements
- mixed content
- substitution groups
- a type with multiple `xsd:any` attributes

If you are assembling Web Services top down or assembling Web service proxies, WebServicesAssembler cannot consume WSDLs that contain the `xsd:choice` or `xsd:group` XML types. If you want to consume a WSDL that contains these XML types, set the WebServicesAssembler `dataBinding` argument to `false` and code the `SOAPElement` so that the payload conforms to the schema definition in the WSDL file.

RPC Encoded Does Not Support Complex Types With Attributes

If the schema contains a binding with an RPC-encoded message format and WebServicesAssembler encounters a complexType with attributes, then it will throw an "unsupported type encountered" error message.

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 10.1.3.1 and 10.1.2

Note the following differences between Oracle Web Services release 10.1.2 (and earlier) and release 10.1.3.1.

- In release 10.1.2 there was no requirement to extend `RemoteInterface` or for methods to throw `RemoteException`. This is now required in release 10.1.3.1, unless you are using the J2SE 5.0 Web Service annotations feature.
- In release 10.1.2 it was possible to publish a class by itself without providing an interface. In release 10.1.3.1 you must provide an interface to publish a class, unless you are using the J2SE 5.0 Web Service annotations feature.

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 10.1.3.1 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 10.1.3.1. 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 10.1.3.1, 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 Also:

"assemble" on page 18-7 and "genWsdl" on page 18-28 for more information on using these commands to generate WSDLs for use with J2SE 5.0 Web Service annotations.

Annotated Classes Must be Listed in the classpath Argument for the WebServicesAssembler assemble Command

If you are using the WebServicesAssembler `assemble` command to assemble an annotated class into a Web service, then the class must be placed in the both the `input` and the `classpath` arguments. The `classpath` should not be needed, as files listed in the `input` argument are implied to be part of the classpath for non-annotated classes.

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, typically to 2000 characters or less. Try to keep the size of the URL small by using a limited number of parameters and short parameter values and names.

- 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.

Testing Web Service Deployment

The Web Service Test Page has the following limitations:

- The Web Service Test 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 Test 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 Test Page. For example, services that use JMS as a transport cannot be tested by using the Test 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.

See Also:

"debug" on page 18-43 for more information on this argument.

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`.

Web Service and Message Parts are Generated into Separate Directories

It is possible, either intentionally or unintentionally, to generate the Web service and the message parts into separate directories. One of the ways this can happen is by setting the `targetNamespace` argument incorrectly.

The value of `targetNamespace` can be either a conforming HTTP URL, a non-conforming HTTP URL, or even a URI. The value of `targetNamespace` maps to a package name in the following ways:

- When the target namespace is a conforming HTTP URL, then the order of the elements in it will be reversed when mapping to a Java package name. For example:

  The target namespace `http://hello.demo.oracle.com` will map to the package name `com.oracle.demo.hello`.

- When the target namespace is a non-conforming HTTP URL, then the order of the elements in it will be unchanged. For example:

  The target namespace `http://hello.demo.oracle` will map to the package name `hello.demo.oracle`.

- When the target namespace is a URN, then the value is mapped into a single identifier, with components separated by underscores ("_"). For example:

  The target namespace `urn:oracle.demo.hello` will map to the identifier `oracle_demo_hello`.

For example, assume you are assembling a Web service with a package path `oracle/demo/hello`. If you set the `targetNamespace` argument to `http://hello.demo.oracle` (a non-conforming URL), then the target namespace in the WSDL will be `http://hello.demo.oracle`, but the message parts will still use the namespace `oracle.demo.hello` (from the service package name). If you then generate a proxy with this WSDL, then the service will be generated into `hello.demo.oracle` and the message parts will be generated into `oracle.demo.hello`.

In this case, you should use `http://oracle.demo.hello` as the value of the `targetNamespace` argument. As an alternative, if the Java code resides in `com/oracle/demo/hello`, then use `http://hello.demo.oracle.com` as the value of the `targetNamespace` argument. This ensures that the generated code will be placed in the expected Java package.

See Also:

- "targetNamespace" on page 18-66 for more information on how this argument responds to different URI values.
- "packageName" on page 18-48 for more information on the behavior of this argument.

genInterface, genProxy, and topDownAssemble Cannot Process WSDL Files Correctly that Contain Multibyte Characters in Element Names

The `genInterface`, genProxy, and topDownAssemble commands cannot process a WSDL file correctly if it uses multibyte characters in element names. The commands will produce Java source files, but they will not compile.

To work around this limitation, set the `dataBinding` argument to `false` when generating the interface.

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.

See Also:

"Understanding the Packaging Structure for J2EE Clients" on page 14-17 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.

OWS-04005 Error Returned when Generated Client and Service Classes Appear in the Same EAR File

If you generate client and Web service classes into the same directory, WebServicesAssembler might package client classes in the Web service EAR file. When you deploy the EAR file to the server, the deployment will fail with an `OWS-04005` error from OracleAS Web Services:

```
OWS-04005 An error occurred for port: ...
```

For example, using Application Server Control to deploy an EAR file that contains generated client classes might return an error similar to the following:

```
ERROR OWS-04005 An error occurred for port: {http://ws.myservice.org/}HttpSoap11:
no serializer is registered for (class org.myservice.ws.MyWebService_sayHello_
ResponseStruct, {http://ws.myservice.org/}sayHelloResponse)
```

If an attempt to invoke this service in the Web Services Test Page, the page will respond with a diagnostic message. For example, a response message similar to the following can be displayed on the Test Page:

```
<env:Envelope  xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
<env:Header/>
<env:Body>
 <env:Fault
   xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <faultcode>env:Server</faultcode>
  <faultstring>no serializer is registered for (class
org.rajkovic.ws.MyWebService_sayHello_ResponseStruct,
{http://ws.myService.org/}sayHelloResponse)</faultstring>
  <faultactor/>
 </env:Fault>
</env:Body>
</env:Envelope>
```

To work around this problem, generate your client and Web service classes into separate directories.

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.

AXIS Platform Cannot Deserialize "0" as the Value of a URI Response

According to the JAX-RPC specification, the `xsd:anyURI` type must be mapped to the `java.net.URI` class in applications that are intended to run on J2SE 1.4 or later. For compatibility with pre-1.4 environments, JAX-RPC implementations are allowed to map this type to `java.lang.String`.

The AXIS platform, however, maps `xsd:anyURI` to the proprietary class `org.apache.axis.types.URI`. This can cause problems if "0" is passed to instantiate a new URI.

For example, when the AXIS platform consumes the value "0" in the following code sample, the URI-parser assigns "0" to the port and calls the constructor for instantiation.

```
java.net.URIret = new java.net.URI("0"); // <wsdl:part
name="inUri" type="xsd:anyURI"/>
return ret;
```

This situation will cause the `setPort` method on AXIS platform to throw an error such as the following:

```
org.apache.axis.types.URI$MalformedURIException: Port cannot be set when host is
null!
```

As a work around, use `xsd:string` instead of `xsd:anyURI` on the AXIS platform.

.NET Clients Cannot Correctly Deserialize Multi-dimensional soapenc:array Responses from OracleAS Web Services

.NET clients incorrectly represent the multi-dimensional SOAP-encoded array (`soapenc:array`) as an array of arrays. When these clients receive a response from OracleAS Web Services that contains a `soapenc:array`, they return an error similar to the following:

```
Unhandled Exception: System.InvalidOperationException: There is an error in XML
document (2, 713). ---> System.ArgumentException: SOAP-ENC:arrayType with
multdimensional array found at <ArrayOfArrayOf_xsd_anyType
xmlns='http://tips.cf'>.
```

To work around this limitation in .NET clients, do not use multi-dimensional `soapenc:array` if interoperability with .NET is required. Use single dimensional arrays, or an array of arrays instead.

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 subelement 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 subelement of `TypeB` is serialized inside the custom serializer for `TypeA`. The custom serializer of `TypeA` must handle the subelement 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.

Changing JMS Properties

If you need to change the properties of your JMS binding after Web service deployment, you must manually edit the WSDL document to enter the correct values for the JMS address (`<jms:address>`) and JMS property value (`<jms:propertyValue>`) elements. After editing the WSDL document, redeploy the service.

Do not try to solve the problem by editing the JMS values in the `oracle-webservices.xml` file. The JMS part of this file is not reloaded when the service is redeployed. If you want to use the `oracle-webservices.xml` file to resolve the problem, then you must use WebServicesAssembler to regenerate the service.

See Also:

The JMS address and JMS property value WSDL elements are described in "WSDL Extensions for JMS Transport" in the *Oracle Application Server Advanced Web Services Developer's Guide*.

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 Web Service Providers

Duplicate MBeans Appear for Web Service Provider Operations

If you are using a dynamic provider endpoint and notice duplicate MBeans appearing for an operation in Application Server Control, then ensure that the `inputName` and

`outputName` attributes are specified for the operation in the `oracle-webvservices.xml` file.

G

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

*Assemble commands, behavior, 18-4
<auth-method> element, C-3
<call-property> element, 14-9, 14-10, D-2
<context-root> element, 13-2, 18-63, 19-9, C-3
<distributable> element, 18-52, 19-9
<download-external-imports> element, C-3
<ejb-link> element, 18-45, 19-5, 19-9
<ejb-name> element, 8-4, 18-45, 19-5
<ejb-transport-login-config> element, 8-9, C-4, C-15
<ejb-transport-security-constraint> element, 8-9, C-4
<endpoint-address-uri> element, 18-64, 19-9, C-4, C-15
<expose-testpage> element, C-5
<expose-wsdl> element, C-5
<handler> Ant tag, 16-3, 18-81, 19-10
<handler> Ant task, 16-3
<handler> element, 14-12
<handler-class> element, 16-4, 16-5, 18-82
<handler-name> element, 16-4, 16-5, 18-82
<implementation-class> element, C-5
<implementor> element, 18-54, 19-9, C-5
<init-param> element, 16-4, 18-82
<jaxrpc-mapping-file> element, 14-5, 14-18, 14-20, 18-47, 19-4, 19-9
<jms-address> element, C-6
<mapping file>.xml file, described, 19-4
<max-request-size> element, C-6
<mtom-support> element, 19-9
<name> element, 14-12
<operation> element, 14-11, C-7, D-3
<operations> element, 14-10, C-7, D-3
<orion-web-app> element, 14-7, D-1
<param name="databaseJndiName"> element, 19-9
<param name="scope"> element, 18-52, 19-9
<param name="session-timeout"> element, 18-52, 19-9
<param="scope"> element, 18-52
<policy> element, C-8
<port name> element, 15-3
<port> Ant tag, 18-78
<port>, porttype Ant subtag, 18-80
<port-component> element, 12-9, 16-3, 16-5, 19-4, 19-9, C-8
<port-component-link> element, 14-5, 14-6
<port-component-name> element, 19-5
<port-component-ref> element, 14-5, 14-6
<port-info> element, 14-9, D-4
<port-name> element, 16-5
<porttype> Ant tag, 18-22, 18-80
<portType> element, 15-3, 18-49
<property> element, C-9
<provider-description> element, C-9
<provider-description-name> element, C-9
<provider-name> element, C-9
<provider-port> element, 19-9, C-10
<proxy> Ant tag, 18-77
<realm-name> element, C-10
<rest-support> element, 12-9, 19-9, C-10
<role-name> element, C-10
<runtime> element, 14-10, 14-11, C-11, D-4
<service> element, 18-50
<service-endpoint> element, 8-3, 19-5
<service-endpoint-interface> element, 14-5, 14-10, 14-18, 14-20, 18-47, 19-5, 19-9, D-4
<service-impl-class> element, 14-9, D-4
<service-interface> element, 14-5, 14-18, 14-20
<service-qname> element, 14-5, 14-9, D-5
<service-ref> element, 14-3, 14-4, 14-8, 14-20, 16-4, 16-5, D-5
<service-ref-mapping> element, 14-7, 14-14
<service-ref-name> element, 14-5, 14-20
<servlet-class> element, 18-43, 19-9
<servlet-link> element, 19-4, 19-6, C-11
<servlet-name> element, 19-6
<soap-header> element, 18-82
<soap-port> element, C-11
<soap-role> element, 18-82
<stub-property> element, 14-9, 14-11, 14-14, D-5
<transport-guarantee> element, C-12
<url-pattern> element, 13-2, 18-64, 19-9
<use-dime-encoding> element, 18-50, 19-9, C-12
<value> element, 14-12
<webservice-description> element, C-13
<webservice-description-name> element, 19-5
<web-site> element, C-12
<WSDL_port_name>Client.java utility client class, 15-1
<wsdl-file> element, 14-5, 14-9, 14-18, 14-20, 19-4, C-13, D-6
<wsdl-location> element, 14-10, D-6

<wsdl-port> element, 14-11, C-14, D-7
<wsdl-service-name> element, C-14
<wsdl-url> element, C-14

A

admin_client.jar file, 19-1, 19-11
analyze command, 18-24
annotations
  developing a Web service, 11-1
  J2SE 5.0 Web Services Annotations, 11-13
    advantages and disadvantages, 11-1
    and EJB version 3.0, 11-2
    annotating and compiling a Java file, 11-11
    annotating and compiling a version 3.0 EJB, 11-12
    Deployment tag, 11-7
    limitations, F-8
    sample Java file with annotations, 11-14
    specification, 11-1, 11-2
    support for REST services, 12-2
    supported Oracle proprietary annotation tags, 11-6
    supported tags, 11-2
    using to assemble a Web service, 11-2
Ant
  setting up for WebServicesAssembler, 4-3
  setting up version 1.5.2 with a previous installation, 4-5
  setting up version 1.6.5, 4-3
  setting up version 1.6.5 with a previous installation, 4-4
Ant tasks
  assembling a J2SE client proxy, 8-6
  assembling a stateful Web service from Java classes, 7-10
  assembling a Web Service from a PL/SQL package, 10-9
  assembling a Web service from a SQL query or DML statement, 10-16
  assembling a Web service from an EJB, 8-4
  assembling a Web service from an Oracle Streams AQ, 10-24
  assembling a Web service from JMS destinations, 9-4
  assembling server-side Java classes as a Web service, 10-33
  configuring a port type, 18-80
  configuring handlers, 18-81
  configuring ports, 18-78
  deployment, 19-11
  for configuration and scripting, 1-6
  generating a Web service top down, 6-4
  generating handler information into a proxy, 18-78
  proxy generation, 18-77
  using multiple instances of an argument, 18-76
  using oracle namespace, 4-6
Apache software
  license, G-2
APIs
  OraSAAJ 1.2, 5-12, 5-13
  packages for J2SE Web service clients, A-1
appendToExistingDDs argument, 18-62
  interactions with ddFileName argument, 18-62
application client module
  deploying and running, 14-5
Application Server Control
  support for deployment, 19-13
application-client.xml deployment descriptor, 16-5
application.xml file, 18-38
application.xml file, described, 19-3
appName argument, 10-7, 10-24, 18-42
aqAssemble command, 18-5
aqAssemble command example, 10-23
aqConnectionFactoryLocation argument, 10-32, 18-54
aqConnectionLocation argument, 10-31, 18-54
architecture of Web services, 3-1
archive
  assigning multiple services, 18-85
    limitations, 18-87
assemble command, 18-7
assemble command example, 7-4, 7-9
assemble command, and J2SE 5.0 Annotations, 18-8
attachment data types, 5-11
attachments package, A-1

B

bindingName argument, 18-42
bottom up assembly
  REST services, 12-6
bottom up Web service assembly
  Java classes, 7-1
  using SOAP 1.2 messages, 5-14

C

Call interface, 15-2
call scope, 7-13
callback package, A-2
call-in
  database, 10-1
  optional WebServicesAssembler arguments, 10-8
  required WebServicesAssembler arguments, 10-7
  SQL to XML type mappings, 10-4
  Web service life cycle, 10-7
call-out
  database, 10-2
  optional JPublisher options
    dir, 10-37
    httpproxy, 10-37
    proxyopts, 10-37
    sysuser, 10-37
  required JPublisher options
    proxywsdl, 10-37
    user, 10-37
  XML to SQL type mappings, 10-6
call-outs

requirements for Web services, 10-37
callScope argument, 18-52
CHUNK_SIZE property, 14-16
chunked data transfer, 14-15
CHUNK_SIZE property, 14-16
DO_NOT_CHUNK property, 14-16
class, handler Ant tag attribute, 18-81
classFileName argument, 18-43
classFileName, porttype Ant attribute, 18-80
className
attribute for wsifEjbPort argument, 11-10
className argument, 18-28, 18-43
className, porttype Ant attribute, 18-80
classpath
components for a J2SE client proxy, A-3
Database Web services-related client JARs, A-5
JMS transport-related client JARs, A-5
OC4J security-related client JARs, A-4
sample commands, A-6
setting for a client proxy, A-2
WS-Reliability-related client JARs, A-5
WS-Security-related client JARs, A-4
classpath argument, 18-43
client
Web services client in the database, 10-36
client code
for accessing an AQ Queue exposed as a Web service, 10-31
client handlers
configuring in an Ant task, 18-81
client stub code
generated by JPublisher, 10-36
client utility class, 15-5
client utility class file, 15-3
ClientConstants.COOKIE_MAP property, 15-8
clients
J2EE
<service-ref> element, 14-3
about, 14-1
accessing a Web service, 14-6
adding client information to deployment descriptors, 14-4
adding handlers, 14-12
adding OC4J-specific information, 14-7, D-1
application client module, 14-5
assembling, 14-2
EJB packaging, 14-19
for stateful web services, 14-13
JMS transport calls, 14-15
managed, 14-8, D-5
packaging, 14-17
schema, 14-4
servlet packaging, 14-17
Web application packaging, 14-17
writing client code, 14-12
J2SE
dynamic invocation interface (DII) clients, 15-2
setting cookies, 15-8
static stub clients, 15-1
tool support, 15-15
writing client applications, 15-5
managed, 14-8, D-5
thread usage, F-9
compatibility with previous Oracle Application Server versions, 1-9
context argument, 10-8, 17-3, 18-62
context-root, URI component, 13-2
Cookie class, 15-8
cookie, setting in a client stub, 15-8
corbaAssemble command, 18-10
corbanameURL argument, 18-53
corbaObjectPath argument, 18-53
correlation-ID
JMS message property, 9-2, F-7
createOneWayOperations argument, 18-65
custom type mapping framework for serialization, 1-6
custom-type-mappings.xml file, 18-36

D

data binding, described, 3-3
data transfer, chunked, 14-15
database connection, establishing, 18-75
database requirements for OracleAS Web Services, 4-6
database resources
assembling as Web services, 10-1
assembling as Web services with JDeveloper, 10-40
limitations on exposing as a Web service, F-7
database Web services, 1-6
Database Web services-related client JARs, A-5
dataBinding argument, 18-69
databinding package, A-1
dataSource argument, 10-8, 10-32, 18-54, 18-75
dbConnection argument, 10-8, 18-55, 18-75
dbJavaAssemble command, 10-32, 18-12
dbJavaAssemble command example, 10-32
dbJavaClassname argument, 18-55
dbUser argument, 10-8, 18-55, 18-75
ddFileName Argument
interactions with appendToExistingDDs argument, 18-62
ddFileName argument, 18-27, 18-62, 18-63
debug argument, 10-8, 18-43
defining
Java classes for Web services, 7-8
Java interfaces for Web services, 7-7
deliveryMode argument, 18-58
deployment
Ant task support, 19-11
Application Server Control support, 19-13
command line example, 19-11
command line support, 19-11
JDeveloper support, 19-12
testing, 13-1
testing REST services, 12-4, 12-8
tool support, 19-11

Deployment annotation tag, 11-7
overriding, 11-13
deployment descriptor
application-client.xml, 16-5
ejb-jar.xml, 16-5
webservices.xml, 8-4, 16-2, 16-3
web.xml, 6-5, 16-5
deployment descriptors
adding J2EE client information, 14-4
additions for REST services, 12-9
managing with WebServicesAssembler, 19-8
relationship with EJB application client EAR files, 14-20
relationship with servlet client EAR files, 14-18
relationship with Web application client EAR files, 14-18
deployment descriptors, relationships between, 19-5
deployment resources, 19-10
Deployment tag, 11-13
deployment-cache.jar file, 14-5
designing message formats, 5-9
Determines, 18-66
DIME attachment support, 1-7
dir
JPublisher option for call-outs, 10-37
DO_NOT_CHUNK property, 14-16
document style
message format, 5-2
documentation roadmap, 2-1
document-literal message format, 5-3
request message, 5-3
sample messages, 5-3
dynamic invocation interface (DII) clients, 15-2

E

EAR archive
adding a file, 18-84
assigning multiple services, 18-85
ear argument, 7-3, 8-3, 10-8, 18-44
interactions with output and war arguments, 18-44, 18-48, 18-51
EJB
writing, 8-7
writing as a service endpoint interface, 8-6
EJB application client EAR files, relationship with deployment descriptors, 14-20
EJB version 3.0
and J2SE 5.0 Web Services Annotations, 11-2
annotating and compiling with J2SE 5.0 Web Services Annotations, 11-12
ejbAssemble command, 18-14
ejbAssemble command example, 8-3
ejb-jar.xml deployment descriptor, 16-5
ejb-jar.xml file, described, 19-3
ejbName argument, 18-45
EJBs
adding transport level security, 8-9
assembling a Web service, 8-3
packaging for a Web service, 19-3
requirements for Web Services, 8-6
support for version 2.0, 8-2
emptySoapAction argument, 18-45
encoded use
message formats, 5-2
encodingStyle SOAP body attribute, 5-2
endpoint implementation, described, 3-3
endpoint scope, 7-13
endpointAddress argument, 18-61
errors
and Java classes, 7-8
exposing as a Web service, 10-15

F

failonerror argument, 18-84
fetchWsdl command, 18-25
fetchWsdlImports argument, 18-64
final-location attribute, 14-9, D-6
fragment, 12-4

G

genApplicationDescriptor command, 18-38
genConcreteWsdl command., 18-26
genDDs command, 18-38
GenericHandler class, 16-2
genInterface command, 6-6, 7-6, 7-11, 8-5, 9-6, 10-11, 10-17, 10-25, 10-34, 14-2, 18-31
genInterface command example, 6-3, 12-2
genJmsPropertyHeader argument, 18-59, F-7
genJUnitTest argument, 18-61
genProxy command, 8-5, 9-6, 10-11, 10-17, 10-25, 10-34, 15-2, 18-34
genProxy command example, 6-6, 7-6, 7-11, 8-5, 9-6, 10-11, 10-17, 10-25, 10-34
genQos argument, 18-65
genQosWsdl command, 18-27
genValueTypes command, 18-36
genWsdl command, 18-28
J2SE 5.0 Annotations, 18-29
GET requests, for REST services, 12-9

H

handler
generating into a proxy, 18-78
handler Ant tag
attributes and child tags, 18-81
class attribute, 18-81
initparam child tag, 18-82
name attribute, 18-82
soapheader child tag, 18-82
soaprole attribute, 18-82
handler chain, processing, 16-1
Handler interface, 16-1, 17-3
implementing, 16-2
handleRequest message, 16-1
handlers
adding to deployment descriptors, 14-12
configuring in an Ant task, 18-81

configuring multiple handlers in an Ant task, 18-84
processing SOAP headers, 17-3
header processing
limitations, F-9
with handlers, 17-3
with parameter mapping, 17-1
with the ServiceLifecycle interface, 17-4
help argument, 18-46
help command, 18-40
host, URI component, 13-2
HTTPClient.Cookie class, 15-8
httpproxy
JPublisher option for call-outs, 10-37
httpProxyHost argument, 18-64
httpProxyPort argument, 18-64

I

idlFile argument, 18-53
idlInterfacename argument, 18-53
idljPath argument, 18-53
IDL-to-Java compiler (idlj), 18-10
If, 18-61
importAbstractWsdl argument, 18-64
IN OUT PL/SQL parameter, 10-10
IN PL/SQL parameter, 10-10
mapping to XML INOUT parameter, 10-13
initialContextFactory
attribute for wsifEjbPort argument, 11-10
initialContextFactory argument, 18-46
initparam, handler Ant child tag, 18-82
INOUT parameter
limitations on SQL types, F-7
INOUT PL/SQL parameter
mapping to XML IN OUT parameter, 10-13
INOUT PL/SQL parameter, XML mapping for, 10-5
input argument, 18-46, 18-76
inputName attribute, 14-11, D-3
installing OC4J, 4-1
interfaceFileName argument, 18-47
interfaceName argument, 18-28, 18-47
interfaceName, porttype Ant attribute, 18-80

J

J2EE client
<service-ref> element, 14-3
about, 14-1
accessing a Web service, 14-6
adding client information to deployment descriptors, 14-4
adding JAX-RPC handlers, 14-12
adding OC4J-specific information, 14-7, D-1
application client module, 14-5
assembling, 14-2
EJB packaging, 14-19
for stateful web services, 14-13
JMS transport calls, 14-15
managed, 14-8, D-5
packaging, 14-17
registering message handlers, 16-4
schema, 14-4
servlet packaging, 14-17
Web application packaging, 14-17
writing client code, 14-12
J2SE 5.0 Annotations, and assemble command, 18-8
J2SE 5.0 Annotations, and genWsdl command, 18-29
J2SE client
registering message handlers, 16-6
J2SE clients
API packages, A-1
Database Web services-related client JARs, A-5
JMS transport-related client JARs, A-5
OC4J security-related client JARs, A-4
possible classpath components, A-3
sample classpath commands, A-6
setting the classpath, A-2
wsclient_extended.jar file, A-2
WS-Reliability-related client JARs, A-5
WS-Security-related client JARs, A-4
Java 2 Enterprise Edition (J2EE) supported standards, 1-2
Java class-based Web services
writing, 7-1
Java classes
and unsupported types, 7-8
assembling stateful Web services, 7-9
assembling stateless Web services, 7-3
defining, 7-8, 7-14
errors, 7-8
packaging for a Web service, 19-2
return values, 7-8
Java interfaces
defining, 7-7, 7-13
requirements, 7-1
Java Management Extensions, 3-4
Java type support for RPC-encoded message format, 5-6
java.rmi.Remote, 8-6
java.rmi.Remote class, 7-1
java.rmi.RemoteException, 8-6
java.rmi.RemoteException class, 7-1
java.util.Map class, 15-9
javax.jms.ObjectMessage, 9-2
javax.servlet.ServletContext, B-2
javax.wsdl.factory.WSDLFactory class, WSDLFactory class, B-1
javax.xml.rpc.Call interface, 15-2
javax.xml.rpc.handler.GenericHandler class, 16-2
javax.xml.rpc.handler.Handler interface, 16-1, 17-3
implementing, 16-2
javax.xml.rpc.holders package, 7-2
javax.xml.rpc.server.ServiceLifecycle interface, 17-4
javax.xml.rpc.server.ServletEndpointContext, B-2
javax.xml.rpc.Service class, 15-2, 15-5
javax.xml.rpc.service.endpoint.address, 14-15
javax.xml.rpc.session.maintain property, 14-14
javax.xml.rpc.Stub interface, 15-1
javax.xml.soap.AttachmentPart, 10-19

JAX-RPC handlers, described, 3-3
JAX-RPC holders, for accessing IN OUT PL/SQL parameters, 10-14
JAX-RPC mapping file, type-mapping.xml, 6-4
jaxrpc-mappings.xml file, 18-36
JDBC web row set format, 10-20
JDeveloper
  packaging Web services, 19-10
  support for assembling Web services from database resources, 10-40
  support for assembling Web services with Java classes, 7-14
  support for deployment, 19-12
  support for J2SE clients, 15-15
JMS destinations
  assembling Web services, 9-1
JMS endpoint Web service
  limitations, F-7
  receive operation, 9-2
  send operation, 9-1
JMS endpoint Web services
  message processing, 9-7
  reply messages, 9-7
JMS message headers
  JMSDeliveryMode, 9-7
  JMSExpiration, 9-7
  JMSPriority, 9-7
  JMSReplyTo, 9-7
  JMSType, 9-7
JMS message property
  correlation-ID, 9-2, F-7
  message-ID, 9-2, F-7
  reply-to-destination, 9-2, F-7
JMS queue
  accessing an Oracle AQ queue, 10-31
JMS transport, 1-8
JMS transport calls, from J2EE clients, 14-15
JMS transport-related client JARs, A-5
jmsAssemble command, 18-16
jmsAssemble command example, 9-3
JMSDeliveryMode, JMS message header, 9-7
JMSExpiration, JMS message header, 9-7
JMSPriority, JMS message header, 9-7
JMSReplyTo, JMS message header, 9-7
JMSType, JMS message header, 9-7
jmsTypeHeader argument, 18-59
JMX, 3-4
jndiName
  attribute for wsifEjbPort argument, 11-10
jndiName argument, 18-14, 18-47
jndiProviderURL
  attribute for wsifEjbPort argument, 11-10
jndiProviderURL argument, 18-47
JPublisher, 10-3
  for generating client stub code, 10-36
JPublisher options
  dir, 10-37
  httpproxy, 10-37
  numbertypes, 10-6
  proxyopts, 10-37
  proxywsdl, 10-37
  sysuser, 10-37
  user, 10-37
jpubProp argument, 10-6, 10-8, 18-55

L

limitations
  for Web service management, F-14
  for WSIF, F-16
  packaging, F-12
linkReceiveWithReplyTo argument, 18-59
literal use
  message formats, 5-2
logging and auditing support, 1-8

M

managed client, 14-8, D-5
management framework, 1-4
management policy enforcement, 3-2
mapHeadersToParameters argument, 17-2, 18-69
mapping type namespaces, 18-72
mappingFileName argument, 18-47
MBeans, 1-4
message formats
  changing, 5-11
  designing, 5-9
  document style, 5-2
  document-literal format, 5-3
  encoded use, 5-2
  literal use, 5-2
  recommendations, 5-11
  relationship to wire format, 5-1
  RPC style, 5-2
  RPC-encoded format, 5-4
  RPC-literal format, 5-8
  SOAP 1.2 support, 5-11
message formats, summarized, 5-1
message handler
  server-side configuration, 16-2
message handlers, 16-1
  registering, 16-3
  registering with J2EE clients, 16-4
  registering with J2SE clients, 16-6
message processing
  for a JMS endpoint Web service, 9-7
message processing components, 3-1
message-ID
  JMS message property, 9-2, F-7
method parameters
  representing in the WSDL, 18-87
MIME attachment support, 1-7
mtomSupport argument, 18-68

N

name
  attribute for wsifEjbPort argument, 11-10
name attribute, 14-11, C-7, D-3
name, handler Ant tag attribute, 18-82

namespace, specifying, 18-75
.NET, 7-1, 7-3, 8-2
new features
Ant tasks for configuration and scripting, 1-6
custom type mapping framework for serialization, 1-6
database Web services, 1-6
DIME attachment support, 1-7
J2SE 5.0 Web Service annotations (Web Services Metadata for the Java Platform), 1-5
JMS transport, 1-8
MBeans support, 1-4
message delivery quality of service (QOS), 1-7
MIME attachment support, 1-7
SOAP header support, 1-6
Web Service Invocation Framework (WSIF) support, 1-8
Web service logging and auditing support, 1-8
Web service provider support, 1-8
Web Service Reliability (WS-Reliability) support, 1-7
Web services management framework, 1-4
Web Services-Security (WS-Security) support, 1-4
nonstandard datatypes
and top down Web service assembly, 6-2
numbertypes
JPublisher option, 10-6

O

OC4J
setting up your environment, 4-1
OC4J security-related client JARs, A-4
OC4J Standalone Environment, defined, 1-11
OC4J, installing, 4-1
OC4J_REPLY_TO_FACTORY_NAME property, 9-7
operations
generated from server-side Java classes, 10-35
Oracle Application Server Environment, defined, 1-11
Oracle Application Server new features, 1-4
Oracle AQ queue
accessing via a JMS queue instance, 10-31
accessing via a Web service client, 10-31
sample queue and topic declaration, 10-26
sample Web service generated from an AQ queue, 10-26
sample Web service generated from an AQ topic, 10-28
Oracle HTTP Server
third party licenses, G-1
oracle namespace, using for Ant tasks, 4-6
Oracle Streams AQ
exposing as a Web service, 10-22
oracle.jdbc.rowset.OracleWebRowSet, 10-20
OracleWebRowSet format, 10-5, 10-19
oracle.webservices package, A-1
oracle.webservices.annotations.Deployment class, 11-7
oracle.webservices.attachments package, A-1
oracle.webservices.ClientConstants.COOKIE_MAP property, 15-8
oracle.webservices.databinding package, A-1
oracle.webservices.management-portability package, A-1
oracle.webservices.provider package, A-2
oracle.webservices.reliability package, A-2
oracle.webservices.security.callback package, A-2
oracle.webservices.soap package, 5-12, 5-13, A-2
oracle.webservices.transport package, A-2
oracle.webservices.transport.ReplyToFactoryName, 14-15
oracle.webservices.transport.ReplyToQueueName, 14-15
oracle.webservices.wsdl package, A-2
oracle.webservices.wsdl.WSDLFactoryImpl, B-1
oracle-webservices.xml deployment descriptor
described, C-1
listing, C-16
oracle-webservices.xml deployment descriptor, described, 19-4
oracle-werbservices.xml deployment descriptor, 8-9
OraSAAJ APIs, 5-12, 5-13
ORBInitialHost argument, 18-53
ORBInitialPort argument, 18-53
ORBInitRef argument, 18-53
OUT parameter
limitations on SQL types, F-7
OUT PL/SQL parameter, 10-10
OUT PL/SQL parameter, XML mapping for, 10-5
output argument, 7-3, 8-3, 10-8, 18-48
interactions with ear and war arguments, 18-44, 18-48, 18-51
outputName attribute, 14-11, D-3
overwriteBeans argument, 18-69

P

packageName argument, 18-48
packaging
limitations, F-12
WebServicesAssembler tool, 19-7
with JDeveloper, 19-10
with WebServicesAssembler, 19-7
packaging structure
Web service based on EJBs, 19-3
Web service based on Java classes, 19-2
packaging, available tools, 19-1
parameter mapping
processing SOAP headers, 17-1
payloadBindingClassName argument, 18-59
platforms, supported, 4-1
PL/SQL functions, mapping to Web service operations, 10-13
PL/SQL package, sample, 10-12
PL/SQL packages, exposing as Web services, 10-9
plsqlAssemble command, 18-17
plsqlAssemble command example, 10-9
port
configuring in an Ant task, 18-78

generating into a proxy, 18-78
port type
configuring in an Ant task, 18-80
port, URI component, 13-2
portability package, A-1
PortComponentLinkResolver property, 14-6
portName argument, 10-8, 18-48
portNameType argument, 18-49
POST requests, for REST services, 12-11
priority argument, 18-59
protocol handlers, described, 3-2
provider package, A-2
proxy
including handler information, 18-78
including port information, 18-78
proxy generation from Ant tasks, 18-77
proxyopts
JPublisher option for call-outs, 10-37
proxywsdl
JPublisher option for call-outs, 10-37

Q

quality of service (QOS), 1-7
quoting symbols for sqlstatement argument, 18-56

R

receive operation
JMS endpoint Web service, 9-2
receiveConnectionFactoryLocation argument, 18-59
receiveQueueLocation argument, 18-59
receiveTimeout argument, 18-60
receiveTopicLocation argument, 18-60
recoverable argument, 7-11, 18-52
REF CURSOR parameter, F-7
Java mapping for, 10-5
reliability package, A-2
reply messages
for a JMS endpoint Web service, 9-7
replyToConnectionFactoryLocation argument, 9-7, 18-60
reply-to-destination
JMS message property, 9-2, F-7
replyToQueueLocation argument, 9-7, 18-60
replyToTopicLocation argument, 9-7, 18-60
responses, for REST services, 12-11
REST GET URL, invoking, 13-14
REST POST request, invoking, 13-14
REST services
accessing operations, 12-4, 12-8
additions to deployment descriptors, 12-9
assembling, 12-2
bottom up assembly, 12-6
defined, 12-2
HTTP GET requests, 12-9
HTTP POST requests, 12-11
J2SE 5.0 Annotation support, 12-2
REST responses, 12-11
testing deployment, 12-4, 12-8
tool support, 12-12
top down assembly, 12-2
REST Web Services
invoking a GET URL, 13-14
invoking an XML REST POST request, 13-14
restSupport argument, 12-3, 12-7, 18-49
return values
for Java classes, 7-8
root package name, specifying, 18-75
RPC style
message format, 5-2
RPC-encoded message format, 5-4
Oracle specific Java type support, 5-6
request message, 5-4
response message, 5-5
sample messages, 5-4
with and without the xsi:type attribute, 5-5
with xsi:type attribute, 5-6
without xsi:type attribute, 5-6
RPC-literal message format, 5-8
request message, 5-9
sample messages, 5-9

S

SAAJ 1.2 APIs, 5-12, 5-13
SAAJ API, 5-12
schema argument, 18-49, 18-76
schemas
J2EE client, 14-4
service-ref-mapping-10_0.xsd, 14-8, D-1
scope, for stateful Java implementations, 7-13
searchSchema argument, 18-50
send operation
JMS endpoint Web service, 9-1
sendConnectionFactoryLocation argument, 18-60
sendQueueLocation argument, 18-61
sendTopicLocation argument, 18-61
server-side Java classes
exposing as a Web service, 10-32
generating Web service operations, 10-35
sample classes, 10-35
supported data and return types, 10-32
server.xml file, 8-5
Web site configuration file, 7-11
Service class, 15-2, 15-5
service endpoint interface, writing an EJB, 8-6
service operations, mapping SQL queries into, 10-18
service, URI component, 13-2
ServiceLifecycle interface, 3-4
processing SOAP headers, 17-4
serviceName argument, 10-8, 18-50
service-ref-mapping-10_0.xsd schema, 14-8, D-1
services
assigning multiple services to an archive, 18-85
limitations on assigning multiple services to a WAR, 18-87
servlet client EAR files, relationship with deployment descriptors, 14-18
ServletContext class, B-2

ServletEndpointContext class, B-2
session argument, 7-11, 18-52
session scope, 7-13
SESSION_MAINTAIN_PROPERTY property, 15-6, 15-9
SESSION_MAINTAIN_PROPERTY runtime property, 7-12, 14-14
setting up your environment, 4-1
Simple Object Access Protocol (SOAP) 1.1 and 1.2, supported standard, 1-2
singleService argument, 18-66
SOAP 1.2 messages
  in bottom up Web service assembly, 5-14
  in top down Web service assembly, 5-15
  message format support, 5-11
SOAP header support, 1-6
soap package, A-2
SOAP with Attachments API (SAAJ), 5-12
SOAPAction header, use with REST services, 12-11
soapheader handler Ant child tag, 18-82
soaprole, handler Ant attribute, 18-82
soapVersion argument, 5-14, 18-66
specifying a namespace, 18-75
sql argument, 10-24, 18-55
SQL DML, 10-15
SQL numeric types
  changing SQL to XML type mapping, 10-6
SQL query
  exposing as a Web service, 10-15
  mapping to service operations, 10-18
SQL statements, samples, 10-18
SQL*PLUS, command for loading a wrapper package, 10-11
sqlAssemble command, 10-15, 18-19
sqlAssemble command example, 10-16
sqlstatement argument, 10-15, 18-56, 18-76
  valid quoting symbols, 18-56
sqlTimeout argument, 18-57
staging directory structure, 18-4
stateful Java implementations, supported scope, 7-13
stateful Web services
  and interoperability, 7-9
  assembling with Java classes, 7-9
  defining Java classes, 7-13, 7-14
  writing Java classes, 7-13
stateless Web services
  assembling with an EJB session bean, 8-3
  assembling with Java classes, 7-4
  defining Java classes, 7-8
  defining Java interfaces, 7-7
  writing Java classes, 7-7
static stub clients, 15-1
strictJaxrpcValidation argument, 18-50
Stub interface, 15-1
style argument, 5-9, 10-8, 10-10, 18-68
supported platforms, 4-1
supported standards
  Java 2 Enterprise Edition (J2EE), 1-2
  Simple Object Access Protocol (SOAP) 1.1 and 1.2, 1-2
  Web Service Description Language (WSDL) 1.1, 1-3
  Web Service Inspection Language (WSIL) 1.0, 1-3
  Web Service Reliability (WS-Reliability), 1-7
  Web Service-Interoperability Basic Profile (WS-I) 1.1, 1-3
  Web Services-Security (WS-Security), 1-4
  WSIL (Web Service Inspection Language) 1.0, 1-3
system-application.xml file, 14-6
sysUser argument, 10-33, 10-38, 18-57
sysuser, JPublisher option for call-outs, 10-37

T

targetNamespace argument, 18-66
Test Page for Web Service testing, 13-1
testing deployment, 13-1
third party licenses, G-1
timeout argument, 7-11, 18-52
timeToLive argument, 18-61
tool support
  deployment, 19-11
  exposing EJBs as a Web service, 8-9
  for J2SE clients, 15-15
  for Web service packaging, 19-1
    JDeveloper, 19-10
    WebServicesAssembler, 19-7
  REST services, 12-12
top down assembly
  REST services, 12-2
top down Web service assembly, 6-1
  and nonstandard types, 6-2
  defined, 6-1
  limitations, F-5
  using SOAP 1.2 messages, 5-15
topDownAssemble command, 18-22
topDownAssemble command example, 6-4, 12-3
topicDurableSubscriptionName argument, 18-61
transport level security, for EJBs, 8-9
transport package, A-2
type mappings
  changing mappings for SQL numeric types, 10-6
  for Web service call-ins, 10-4
  for Web service call-outs, 10-6
type namespaces, mapping, 18-72
type-mapping.xml, JAX-RPC mapping file, 6-4
typeNamespace argument, 18-67, 18-75

U

unsupported types
  and Java classes, 7-8
unwrapParameters argument, 6-2, 18-22, 18-69, F-15
uri argument, 10-8, 18-63
URI components, 13-2
use argument, 5-9, 10-8, 18-68
useDataSource argument, 18-57
useDimeEncoding argument, 18-50
user, JPublisher option for call-outs, 10-37

userThreads option (Oracle Application Server), F-9

V

valueTypeClassName argument, 18-69, 18-76
valueTypePackagePrefix argument, 18-70, 18-75
version command, 18-41

W

WAR archive
  adding a file, 18-84
  assigning multiple services, 18-85
  limitations on assigning multiple services, 18-87
war argument, 18-51
  interactions with output and ear arguments, 18-44, 18-48, 18-51
Web application client EAR files, relationship with deployment descriptors, 14-18
Web Service Description Language (WSDL) 1.1, supported standard, 1-3
Web Service Home Page
  limitations, F-9
  obtaining a WSDL file, 13-18
Web Service Inspection Language (WSIL)
  disabling, 1-3
Web Service Inspection Language (WSIL) 1.0 standard, 1-3
Web Service Invocation Framework (WSIF) support, 1-8
Web service management
  limitations, F-14
  policy enforcement, 3-2
Web service provider support, 1-8
Web Service Reliability support, 1-7
Web Service Test Page, 13-1
  described, 13-6
  for REST Services, 12-9
  for REST Web Services, 13-14
  using, 13-6
Web Service Test page
  using, 13-1
Web Service-Interoperability Basic Profile (WS-I) 1.1 standard, 1-3
Web services
  architecture, 3-1
  assembling from database resources, 10-1
  assembling from DML statements, 10-15
  assembling from Oracle Streams AQ, 10-22
  assembling from PL/SQL packages, 10-9
  assembling from server-side Java classes, 10-32
  assembling from SQL queries, 10-15
  assembling with JMS destinations, 9-1
  defined, 1-1
  deployment resources, 19-10
  development life cycle, 3-4
  EJBs
    assembling, 8-1
    stateless, 8-1
  Java classes
    assembling, 7-1
    limitations, F-6
    stateful, 7-1
    stateless, 7-1
    tool support, 7-14
  message flow, 3-1
  operations generated from server-side Java classes, 10-35
  packaging, 19-1
  processing components, 3-1
  top down assembly, 6-1
Web Services-Security (WS-Security) supported standards, 1-4
WebRowSet format, F-7
webservices package, A-1
WebServicesAssembler
  limitations, F-10
  setting up Ant, 4-3
WebServicesAssembler arguments
  appendToExistingDDs, 18-62
  appName, 18-42
  aqConnectionFactoryLocation, 18-54
  aqConnectionLocation, 18-54
  bindingName, 18-42
  callScope, 18-52
  classFileName, 18-43
  className, 18-28, 18-43
  classpath, 18-43
  context, 17-3, 18-62
  corbanameURL, 18-53
  corbaObjectPath, 18-53
  createOneWayOperations, 18-65
  dataBinding, 18-69
  dataSource, 18-54, 18-75
  dbConnection, 18-55, 18-75
  dbJavaClassName, 18-55
  dbUser, 18-55, 18-75
  ddFileName, 18-27, 18-62, 18-63
  debug, 18-43
  deliveryMode, 18-58
  ear, 18-44
  ejbName, 18-45
  emptySoapAction, 18-45
  endpointAddress, 18-61
  failonerror, 18-84
  fetchWsdlImports, 18-64
  genJmsPropertyHeader, 18-59
  genJUnitTest, 18-61
  genQos, 18-65
  help, 18-46
  httpProxyHost, 18-64
  httpProxyPort, 18-64
  idlFile, 18-53
  idlInterfaceName, 18-53
  idljPath, 18-53
  importAbstractWsdl, 18-64
  initialContextFactory, 18-46
  input, 18-46, 18-76
  interfaceFileName, 18-47
  interfaceName, 18-28, 18-47

jmsTypeHeader, 18-59
jndiName, 18-14, 18-47
jndiProviderURL, 18-47
jpubProp, 18-55
linkReceiveWithReplyTo, 18-59
mapHeadersToParameters, 17-2, 18-69
mappingFileName, 18-47
mtomSupport, 18-68
optional for Web service call-in, 10-8
ORBInitialHost, 18-53
ORBInitialPort, 18-53
ORBInitRef, 18-53
output, 18-48
overwriteBeans, 18-69
packageName, 18-48
payloadBindingClassName, 18-59
portName, 18-48
portNameType, 18-49
priority, 18-59
receiveConnectionFactoryLocation, 18-59
receiveQueueLocation, 18-59
receiveTimeout, 18-60
receiveTopicLocation, 18-60
recoverable, 18-52
replyToConnectionFactoryLocation, 18-60
replyToQueueLocation, 18-60
replyToTopicLocation, 18-60
required for Web service call-in, 10-7
restSupport, 18-49
schema, 18-49, 18-76
searchSchema, 18-50
sendConnectionFactoryLocation, 18-60
sendQueueLocation, 18-61
sendTopicLocation, 18-61
serviceName, 18-50
session, 18-52
singleService, 18-66
soapVersion, 18-66
sql, 18-55
sqlstatement, 18-56, 18-76
sqlTimeout, 18-57
strictJaxrpcValidation, 18-50
style, 18-68
sysUser, 18-57
targetNamespace, 18-66
timeout, 18-52
timeToLive, 18-61
topicDurableSubscriptionName, 18-61
typeNamespace, 18-67, 18-75
unwrapParameters, 18-22, 18-69
uri, 18-63
use, 18-68
useDataSource, 18-57
useDimeEncoding, 18-50
valueTypeClassName, 18-69, 18-76
valueTypePackagePrefix, 18-70, 18-75
war, 18-51
wsdl, 18-65
wsdlTimeout, 18-67
wsifDbBinding, 18-5, 18-18, 18-20, 18-29, 18-57
wsifDbPort, 18-57
wsifEjbBinding, 18-14, 18-29, 18-71
wsifJavaBinding, 18-7, 18-29, 18-71
wsifJDbBinding, 18-12
WebServicesAssembler commands
analyze, 18-24
aqAssemble, 18-5
assemble, 18-7
corbaAssemble, 18-10
dbJavaAssemble, 18-12
ejbAssemble, 18-14
fetchWsdl, 18-25
genApplicationDescriptor, 18-38
genConcreteWsdl, 18-26
genDDs, 18-38
genInterface, 18-31
genProxy, 18-34
genQosWsdl, 18-27
genValueTypes, 18-36
genWsdl, 18-28
help, 18-40
jmsAssemble, 18-16
plsqlAssemble, 18-17
sqlAssemble, 18-19
topDownAssemble, 18-22
version, 18-41
WebServicesAssembler tool
Ant task support, 18-2
bottom up assembly support, 18-1
command line syntax, 18-3
deployment support, 18-2
managing deployment descriptors, 19-8
packaging commands, 19-7
packaging Web services, 19-7
top down assembly support, 18-1
XML schema driven assembly support, 18-2
WebServicesAssembler tool, described, 18-1
webservices.xml configuration file, described, 19-4
webservices.xml deployment descriptor, 6-5, 8-4, 16-2, 16-3
web.xml deployment descriptor, 6-5, 16-5
web.xml deployment descriptor, described, 19-4
wsclient_extended.jar client class file, A-2
contents, A-2
wsclient_extended.jar file, 6-7, 7-7, 7-12, 8-6, 9-6, 10-12, 10-18, 10-25, 10-35, 15-3
WSDL API
creating WSDL factory instances, B-1
creating WSDL file reader, B-1
extracting information, B-1
getting a WSDL factory instance, B-1
getting the WSDL as a resource, B-2
reading a WSDL file, B-1
setting a timeout, B-1
wsdl argument, 18-65
WSDL factory instances, creating, B-1
WSDL file
obtaining directly, 13-18
representing Java method parameters, 18-87
WSDL file, described, 19-4

wsdl package, A-2
WSDL_READ_TIMEOUT property, B-1
WSDLFactoryImpl class, B-1
wsdl-override-last-modified attribute, 14-10, D-7
wsdlTimeout argument, 18-67
WSIF
  limitations, F-16
wsifDbBinding argument, 18-5, 18-12, 18-18, 18-20, 18-29, 18-57
wsifDbPort argument, 18-57
wsifEjbBinding argument, 18-14, 18-29, 18-71
wsifJavaBinding argument, 18-7, 18-29, 18-71
WSIL (Web Service Inspection Language) 1.0
  standard, 1-3
wsmgmt.xml management policy file, F-14
WS-Reliability-related client JARs, A-5
WS-Security-related client JARs, A-4

X

XDB rowset format, 10-5, 10-19, 10-20, F-7
XMLprocessing, described, 3-2
XMLType SQL type, mapping to XML any, 10-15
xpath queries, F-15
xsi:type attribute, 5-6